AFRL-IF-RS-TR-2003-198
Final Technical Report
August 2003

DATA INTENSIVE SYSTEMS (DIS) BENCHMARK PERFORMANCE SUMMARY

Titan Corporation

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. J201

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2003-198 has been reviewed and is approved for publication.

APPROVED: /s/
CHRISTOPHER J. FLYNN
Project Engineer

FOR THE DIRECTOR: /s/
JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

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, gathering and maintaining the data needed, and completing and reviewing this 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 Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	AUGUST 2003	Final Jun 99 – Dec 02

4. TITLE AND SUBTITLE
DATA INTENSIVE SYSTEMS (DIS) BENCHMARK PERFORMANCE SUMMARY

5. FUNDING NUMBERS
C - F30602-99-C-0153
PE - 62301E
PR - H307
TA - DI
WU - SB

6. AUTHOR(S)
Joseph Musmanno

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Titan Corporation
Aerospace Electronics Division
470 Totten Pond Road
Waltham Massachusetts 02451

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington Virginia 22203-1714

AFRL/IFTC
26 Electronics Parkway
Rome New York 13441-4514

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2003-198

11. SUPPLEMENTARY NOTES
AFRL Project Engineer: Christopher J. Flynn/IFTC/(315) 330-3249/ Christopher.Flynn@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***
Peak processor performance increases at a rate of 60% per year, but memory access speeds increase at a rate of only 7% per year. Computing-system designers compensate for the resulting divergence by incorporating caches or latency-hiding measures into their designs. However, elements such as larger caches, prefetching, and multithreading do not address the needs of data-intensive DoD applications, which consequently operate at rates far below the peak processor capacity. As the mismatch between processor and memory grows the number of applications unable to operate at peak rates increases. The DARPA Data Intensive Systems Program was created to address this problem. A variety of novel architectures or enhancements were developed under this program to increase the effective performance-as opposed to the rated peak performance of systems running data-starved applications. Under this project, a DIS Benchmark Suite was developed to measure the performance of the prototypical systems. Additionally, the DIS Stressmark Suite was developed to assist performance measurement during the development process. Participating teams were expected to utilize these tools and supply their measurements. In this report the benchmarking tools are introduced, the reported results are summarized, and an objective analyses of the results is provided.

14. SUBJECT TERMS
Data Intensive Systems, Computer Benchmarks, Processor in Memory, Adaptive Cache Management, Memory Wall

15. NUMBER OF PAGES
144

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

Abstract

Peak processor performance increases at a rate of 60% per year, but memory access speeds increase at a rate of only 7% per year. Computing-system designers compensate for the resulting divergence by incorporating caches or latency-hiding measures into their designs. However, elements such as larger caches, prefetching, and multithreading do not address the needs of data-intensive DoD applications, which consequently operate at rates far below the peak processor capacity. As the mismatch between processor and memory grows, the number of applications unable to operate at peak rates increases.

The DARPA *Data Intensive Systems* Program was created to address this problem. A variety of novel architectures or enhancements were developed under this program to increase the *effective* performance—as opposed to the rated peak performance—of systems running data-starved applications. Under this program, we developed the *DIS Benchmark Suite* to measure the performance of the prototypical systems. Additionally, the *DIS Stressmark Suite* was developed to assist performance measurement during the development process. Participating teams were expected to utilize these tools and supply their measurements.

In this report, we introduce the benchmarking tools, summarize the reported results, and perform objective analyses of the results. We also compile normalized data to support basic comparisons of the approaches.

1 Introduction

The speed of modern processors is growing at approximately 60% per year, while the speed of memory increases only about 7% per year. The resultant growing mismatch is making it increasingly difficult to fully exploit the power of new processors. Architectural mechanisms, such as out-of-order instruction execution and non-blocking caches, can hide the memory latency of programs that have high degrees of spatial and temporal locality. However, many important defense applications employ large data sets accessed non-contiguously. These applications cannot take full advantage of typical memory-access optimizations, and consequently perform at substantially below peak rates due to data starvation.

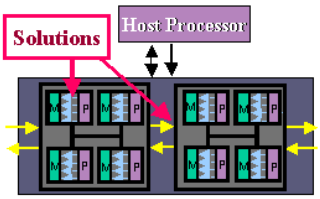
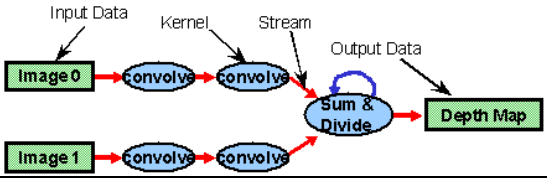
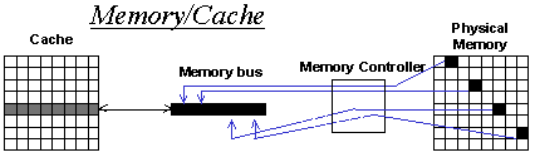
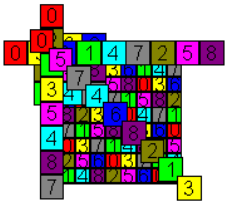
The DARPA Information Processing Technology Office (IPTO)[1] created the Data-Intensive Systems (DIS) research program[2] to develop new data-access architectures in a direct response to data-starved Defense applications. These architectures would benefit any application required to access or manipulate data in any manner not consistent with regular, ordered data access patterns and local working set models assumed by today's conventional architectures. Such applications include model-based Automatic Target Recognition (ATR), synthetic aperture radar (SAR) codes, large scale dynamic databases/battlefield integration, dynamic sensor-based processing, high-speed cryptanalysis, high speed distributed interactive and data intensive simulations, data-oriented problems characterized by pointer-based and other highly irregular data structures, security-sensitive applications (where protection and validation strategies are central), and real-time visualization.

The DIS mission was *to explore new memory architectures concepts, techniques, and implementations that reduce data bandwidth and data access latency limitations for defense applications*. The program attempted to reduce data access limitations by pursuing novel approaches in two categories:

1) processing of data *in situ*, including Processor-in-Memory (PIM) nodes and computational streaming; and
2) adaptive cache management.

[1] http://www.darpa.mil/ipto

[2] http://www.darpa.mil/ipto/research/dis

In-situ processing: place logic within memory/data stream to maximize effective memory bandwidth.	Adaptive cache management: develop mechanisms that empower applications to directly manage the flow and placement of data throughout the memory hierarchy.
Processor in Memory (PIM)  **Data Flow (Stream) Processing**  	❖*Adaptive Virtual Memory/Cache*  ❖*Algorithm, Compile & Data Placement* **Perfect Latin Squares: conflict-free access** 

As part of the DIS effort, the Aerospace Electronics Division[3] of Titan Corporation[4] was charged with benchmarking and evaluating the DIS technology. A suite of benchmarks and stressmarks were created, along with appropriate test data and metrics. DIS participants were requested to utilize this suite, and report results. This document is the summary and analysis of those results.

[3] Formerly Atlantic Aerospace Electronics Corporation. http://www.aaec.com

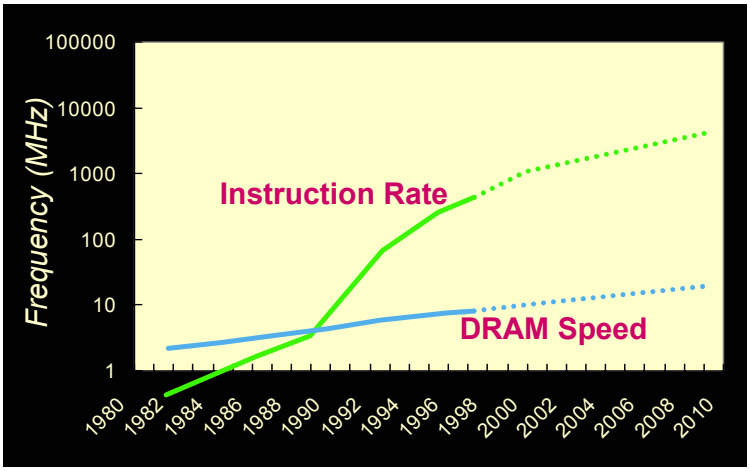
[4] http://www.titan.com

1.1 DIS Program Summary

In this section, the DIS program is summarized briefly, to set the background and scope of this report. The text of the following three subsections is quoted directly from the program web pages,[5] where the reader is referred for detail on individual projects.

1.1.1 Background

"Peak processor performance increases are at a rate of 60% per year. This is a necessary trend that enables high throughput Defense applications to be hosted in compact, COTS-based architectures. Unfortunately, the access latency of the memory chips and the memory bandwidth of the processors they feed are not keeping pace. Memory access latency is improving at only 7% per year, resulting in data-starved processing and an inability to utilize processor advances or reach peak system processing performance. High bandwidth and dynamically accessed data base systems have reached a "memory wall" in conventional architectures that result in a memory/CPU 'impedance mismatch'—the rate that data can be supplied to a 'processor' does not match the rate at which the 'processor' can perform useful work on that data. Thus, the performance of many Defense applications will be constrained not by the processing speed or memory size of their computers, but rather by the ability of their memory systems to deliver the data.



"Consider Object-Level Change Detection (OLCD), which contains a COTS Object-Oriented Database (OODB) of 2 Gbytes. To meet its performance goals, the OODB is distributed over four processors. These processors are idle 98% of the time while their memories struggle to keep up with patterns of data access for which they were not designed. To overcome this problem, Defense users of object-oriented databases and other data-intensive applications can build special purpose devices (SPDs) designed to optimally serve their specific task. However, such systems are expensive to design and build, they are even more expensive to program (as there is no COTS software), and they present a long-term maintenance burden. These solutions very rapidly fall behind their commercial counterparts.

"In addition, the continued scaling of VLSI technology is forcing a dramatic change in the nature of electronic computing. There are three aspects of scaling that are driving this change. First, the decreasing cost of computation is pushing it into many new applications, ones that are off the desktop and in embedded systems dealing with real-time data. Second, the cost of on-chip wires is growing in significance, making today's architectures that use a global resource model harder and harder to build. Third, as chip complexity grows, building custom silicon solutions for each application space will become less and less cost effective since the NRE costs of the design will continue to grow. Cost effective chips will have to be sold in large volumes.

"The above issues are motivating the development of new data-access architectures […]"

[5] http://www.darpa.mil/ipto/research/dis

1.1.2 Application Scope

Data-Intensive Applications Are	Data-Intensive Applications Are NOT
Large high-rate data streams (data-rate limited)	Applications with small working sets of contiguous data
Distributed data access (non-contiguous data)	
Dynamic data accesses (lack of predictability)	

1.1.3 Goals

The goal and derived objectives of the DIS program were given as follows.

Goal

> Demonstrate data placement, advanced data caching, and memory/data architecture concepts with the potential to realize greater than one-order of magnitude (10X) improvement in run-time performance for data-starved applications.

Objectives

- Develop new data-access architectures, data flow and placement concepts, and the associated chip-level technologies necessary to respond to data-starved defense applications.
- Enable the full use of increasing processing element capabilities and reduce the under-utilization of system resources due to restricted data flow and high latency that many data rich applications encounter with current memory implementations.

Memory Architecture Challenges

- Maximize effective memory bandwidth
- Design, develop, fabricate prototype Processor in Memory (PIM) devices to optimize memory bandwidth
- Design, develop, fabricate prototype streaming data processing devices to optimize processing of high rate, stream data applications
- Demonstrate potential technologies for selected data-intensive applications

Data Placement/Movement Challenges

- Develop data placement techniques that significantly improve data availability
- Develop augmented and adaptive cache techniques and implementations that optimize data movement and effective utilization
- Develop algorithmic approaches to utilize architectures and techniques being developed
- Supporting infrastructures:
- Develop evaluating/validating benchmarks and stressmarks

1.1.4 Program Timeline

When created, the program included two major milestones, situated at the ends of two phases of investigation. The milestones involved required demonstrations of performance, and came to be known as the *mid-term*, and *final* examinations. They had the following minima defined for success:

Mid-term	• In Situ Processing: Working memory chips capable of 1M ray-patch intersection computations per second each, along with simulation results justifying the anticipation of system capacity to meet the final goals. • Cache Management: 20x performance improvement in data-starved applications performance.
Final	• X-Patch challenge: 64 looks at a T-72 in under one minute (~100M ray-patch intersection calculations per second. • 20x performance improvement in Method-of-Moments Finite Multipole Method. • 100x performance improvement in other data-starved defense applications.

The DIS Benchmark Suite was developed in support of these goals.

During the initial phase, it became clear that the mid-term goal of working chips was overly ambitious; many teams cited difficulty with fabrication. Additionally, program needs changed. Consequently, program goals were modified. The first phase was extended, and the second phase was cancelled.

Some participants also needed smaller, simpler benchmark algorithms for use during testing and development. The DIS Stressmark Suite was created to support this need. It was not intended as a replacement of the Benchmark Suite.

1.2 Benchmarking Project Summary

As part of the DIS program, we created a suite of tools for evaluating the performance of new data-intensive systems. This section introduces the goals and scope of the benchmarking effort. Sections 2.5 and 2.6 give more information about the benchmark suite.

1.2.1 Goals

The primary goal of this effort was the development of a benchmark suite that can be used to quantify the performance gains likely to be achieved for defense computer programs when implemented using approaches and architectures developed under the DIS program.

Any benchmark specification dealing with early research into new systems must remain architecture-neutral. In support of this goal, the benchmark specifications were essentially only the mathematical description of problems' solutions. Of course, due to years of development in the context of Von Neumann computer architectures, many known optimizations were utilized, and an attempt was made to provide or reference these, so that participants charged with implementing the benchmarks would not be faced with having to independently rediscover the optimizations.

Benchmarks that focus on the measurement of relative performance frequently involve implementation only of specific, isolated functions, resulting in accurate measurement of peak performance. This level of performance is rarely realizable in general application, so benchmarks that include the processes of data movement and preparation were desirable for a more generalized measurement of real performance. Considering the variety of architectures under scrutiny in the DIS program, it would have been dangerous to presume that these "overhead" functions diminish in

proportional resource consumption as data sets grow larger. Therefore, avoidance of isolated tasks as benchmarks was a goal of this program; rather, performance related to the interactions between program components was to be included in the measurements.

Weems, et al,[6] while reviewing lessons from prior benchmark efforts, pointed out:

> "Having a known, correct solution for a benchmark is essential, since it is difficult to compare the performance of architectures that produce different results. For example, suppose architecture A performs a task in half the time of B, but A uses integer arithmetic while B uses floating-point, and they obtain different results. Is A really twice as powerful as B?"

Therefore, a complete solution with test data sets was considered one of the essential components of the distribution of the benchmark specification.

Although there are sometimes competing ideas about how to best solve a particular problem, the goal of a benchmark is not specifically to solve a problem, but rather to test the performance of different machines doing comparable work. Since DIS architectures were likely to vary greatly, significant latitude was allowed in the implementation of a solution to benchmark problems. However, participants were cautioned to remain cognizant of the fact that ultimately, the measurements taken must be meaningful in the context of defense problems, and specifically in the context of *relative* gain. So, it was not a goal of the benchmark effort to develop the best solutions for the most difficult problems; rather, it was a goal to employ pertinent solutions to problems expected to benefit from DIS research, and allow enough flexibility to maximize individual performance, yet remain consistent and comparable.

While benchmarks that are too simplistic do not offer valuable results, those that are too complex are never implemented, at least in a meaningful way. Resources are limited, so ease of implementation is a factor of consideration. It was a goal of this program to develop benchmark programs that required relatively little source code during implementation, yet still offer meaningful results.

Often, developed high-performance systems remain under-utilized due to the esoteric or difficult nature of their programming. Therefore, an important goal of the effort was to evaluate the labor costs associated with use of candidate architectures. The ability to handle existing, 'legacy code' was an important consideration, as is the labor cost to exploit the powerful features of these systems.

A program will generally execute faster when its required data set is small enough to fit in main memory, as opposed to when paging or swapping is required. Likewise, when the data set is small enough to fit in cached memory, it will generally execute faster still. Balancing the competing factors of speed, size, and cost is a major engineering decision, and quantifying the effects of that decision was a goal of this effort.

Finally, in support of the primary goal of being able to quantify performance gains, it was a goal of this effort to remain open to any additional information participants wish to supply that will assist reviewers in making an accurate determination. Minimum participation requirements were specified, but additional results and analyses were solicited.

One of the greatest challenges, of course, was coming up with benchmarks open to algorithmic modification, so that one can take advantage of the architecture's novelty, but sufficiently rigid that the results support useful comparison.

1.2.2 Benchmark Suite

Given the motivation and goals outlined above, the first phase began with determination of the content of the benchmark suite. Understanding the desire to retain an application-oriented focus, algorithms were selected, presented to participants, refined based on participant feedback, and developed. The final suite contained five algorithms from three classes found to provide a representative scope of achievable performance improvement for problems of interest to key DARPA programs.

---

[6] Weems, Riseman, and Hanson, The DARPA Image Understanding Benchmark for Parallel Computers, *Journal of Parallel and Distributed Computing*, 11, 24 January 1991.

Application Domain	Benchmark
Model-Based Image Generation – This class includes generation of synthetic signatures and scenes for targets and terrain based on complex models of objects and sophisticated camera models for various sensor types. Applications include target recognition, real-time scene simulation for visualization or training, and model-driven change detection.	Synthetic Aperture Radar Ray Tracing[7] Method of Moments, Finite Multipole Method[8]
Target Detection – This class includes spatial- and frequency-domain target detection in scenes collected from a wide range of sensor types. Applications include automated exploitation and cueing systems.	Image Understanding Multidimensional Fourier Transform
Database Management – This class includes algorithms for index maintenance, storage management, and content-based query processing. Applications include sensor data archive management and geographic information systems such as the Dynamic Database for Battlefield Situation Awareness.	Data Management

1.2.3 Benchmark Suite Lessons

The DIS Benchmark Suite was created to facilitate the demonstration of comprehensive goals. However, those goals were frequently dependent on chip fabrication cycles. Since the second phase of the program was curtailed, the benchmarks were not fully implemented.

- The application-oriented focus of the Benchmarks was appropriate for implementation after chip development as intended, but participants desired simpler code fragments for use during the early stages of architecture development.
- Mechanically, the I/O required by the benchmarks was cumbersome. It would certainly be a challenge to mimic a data-intensive application that did not require a lot of I/O. After all, reuse of smaller quantities of data would obviate the primary need for new architectures.
- The benchmarks were dependent upon the availability of high-level OS features. This was a justifiable dependence given the program goals, but high-level OS implementations were not available under the curtailed schedule.
- Finally, many of the important metrics (e.g., power, labor) depended upon subjective collection. Ultimately, very few of the teams were able to supply useful information in these areas.

1.2.4 Benchmark Miss Ratios

The AMRM team[9] reported that the DIS Benchmark Suite exhibited low cache-miss ratios. Upon investigation, several problems were found with that finding:

---

[7] Benchmark developed by ERIM, International.

[8] Benchmark developed by the Boeing Corporation.

[9] See Section 12.

- The team only looked at average miss-ratios. However, cache misses come in waves, and a period of many misses cannot be compensated by a miss-free period. Considering the scale of the benchmark applications, the miss-ratios reported are actually quite high.

- Due in part to the above, average miss-ratio is not a good metric. In its report,[10] the AMRM team stated, "[We] show that memory adaptations may be effective in a short period of time, but the performance is not detectable globally. […] we show that, in general, miss rate is not an effective metric."

- In its tests, the team only measured first *n* steps, where *n* was on the order of millions. For these benchmarks, that number of cycles would generally not even represent the entire initialization phase. In other words, the heart of the benchmark was not examined. In later measurements, the team found higher miss-rates when looking at later portions of the execution sequence.

1.2.5 Stressmark Suite

In contrast to the benchmarks, the stressmarks are largely synthetic problems, created expressly for the purpose of exhibiting certain memory access behaviors. The realism of these is necessarily limited, but taken together they can be used to demonstrate mitigation of specific data-intensive problem areas. Where the benchmarks are focused code retaining the context of the enveloping application, the stressmarks are a suite of specific procedures that illustrate DIS attributes, intended to be taken collectively.

The suite includes seven kernels, each composed of just tens of lines of code. The data to be manipulated within the algorithms is all generated randomly, which additionally limits the realism of the benchmark. The suite is shown in the following table.

Stressmark	Characteristic
Pointer	Irregular access to sparse data; pointer-chasing.
Update	Like Pointer, but requiring memory writes.
Neighborhood	Mixed use: dense 2-D FIR kernel, with histogram.
Matrix	Sparse Matrix-Vector Multiply.
Field	Dense, regular access; infrequent writes.
Transitive Closure	Dense, regular access; read-modify-write.
Corner-Turn	Unit- and non-unit-stride access.

The stressmarks are described further in sections 4.11 through 4.18.

[10] Haitao Du, et al, A Quantitative Evaluation of Adaptive Memory Hierarchy, *UC Irvine Technical Report ICS-TR-01-41*, August, 2002.

1.2.6 Stressmark Suite Lessons

While the Stressmark Suite met its goals, the course of the program illustrated the following lessons:

- The Corner-Turn and Matrix stressmarks measure the time required to complete a fixed number of iterations. A more convenient specification would have been to count the iterations completed in a fixed time period.
- Although allowed and recommended, some teams did not generate the (large) initial data outside of their simulations. Consequently, some of the simulations took a very long time to run. The Matrix stressmark, especially, executed in a small fraction of the time needed to generate the input data.
- When stressmarks are to be utilized in simulations, brief execution is valuable. For DIS, some teams did not execute entire stressmarks, choosing instead to abort operation after some number of steps, or one major iteration.
- The AMRM team reported that the stressmarks have no conflict misses—only compulsory and capacity. This is a side-effect of the need for extremely simple kernels requiring practically no input or output. The benchmarks are more realistic in their data usage.
- Though the stressmarks should be valuable for experimentation, the program's emphasis on the stressmarks is neither intentional nor desirable.
- Teams were inconsistent in their use of the stressmarks, and of the reporting of results. A great deal of effort was required to compile results for even cursory comparisons. It is very difficult to strike a balance between implementation flexibility and minimization of reporting variability. Future benchmark developers should consider bifurcating their suite to address these opposing goals.

2 Methods

The DIS benchmarking effort faced major challenges. Consider:

- How does one measure the memory system performance without incidentally measuring the processor? Items such as clock rates and instruction sets bear dramatic effects on the function of the memory system.
- How does one normalize for *radically* differing architectural approaches? Note that when the effort began, suggested approaches covered a spectrum from simple software modifications all the way to multidimensional MIMD array processors with no global addressing.
- How does one continually ensure DoD relevance when projects retain contrary commercial goals?

In this section, we describe the tools used in our attempt to meet those challenges. The general procedures, benchmarks, stressmarks, and data are summarized. For details about these, the reader is referred to *DIS Benchmark Suite Analysis and Specifications*[11] and *DIS Stressmark Suite*[12]. Efficiency is also discussed in the context of derived measurements.

2.1 Specified Procedures

In addition to selecting the benchmark algorithms, certain procedures were developed to assist with the analysis of results submitted by participating teams. These generally applied to both benchmarks and stressmarks, and are introduced in this subsection for the purpose of illustrating the fact that benchmarking is a process, rather than simply the utilization of standard blocks of code.

2.1.1 Benchmark Procedures

A detailed description of the procedures to be used for benchmarking was given with the specifications. Some highlights are noted here.

- Participants were allowed to modify the supplied source code to their pleasure. Modified code was required to be supplied with results, so that competing teams could benefit from algorithmic insights. This requirement was expected to reduce the effects of any given team's access to domain expertise.
- Participants were additionally required to measure performance the benchmarks using 'un-optimized' code. That is, the baseline code provided for each benchmark was to be compiled without any modification, and run 'as-is' to establish performance of the architecture utilizing only automatic optimizations. In addition, participants are encouraged to modify or replace this baseline source code and run the tests again, establishing performance after manual optimizations.
- A detailed set of guidelines regarding the information needed to ensure adequate interpretation of the results was given.

[11] http://www.aaec.com/projectweb/dis/DIS_Benchmarks_V1.pdf

[12] http://www.aaec.com/projectweb/dis/DIS_Stressmarks_V1_0.pdf

- It was noted that the energy spent by implementers laboring in the development of each benchmark implementation is of special interest. Since it was understood to be difficult to measure accurately, participants were asked to candidly report on this subject. A summary of the required skills, labor expended, and problems encountered during the process was identified as necessary to establish the utility of a given design.

- In several cases, common algorithms used in the solution of the selected problems were optimized for use with traditional systems. For example, certain steps in the Method of Moments algorithm are only present to take advantage of unit-stride memory accesses. While these steps were not strictly part of the solution algorithm, it would be onerous to require participants to independently rediscover them. In some cases, it would require implementers to become experts in the application field. So, the algorithmic descriptions are intended to cover the mathematics of the solutions only. Known optimizations were additionally provided for informational purposes, but implementation of these was not required.

- Pseudo-code provided in the algorithmic specifications was intended to provide guidance and clarification of algorithms **only**; it was not intended to represent optimal–*or efficient*–implementations of problem solutions. Similarly, pseudo-code was not intended to represent 'known optimizations' as described above, except when specifically identified as such.

- Participants were expected to use all the supplied data sets. These were provided in a range of sizes, so as to test fixed-system scaling effects resulting from limited-resource optimizations. If particular data sets were unusable for some reason (e.g., the dataset requires more memory than that which is available), the reason was to have been reported.

- There was not to be any recompilation or manipulation of the software or hardware between runs producing final measurements. Recall that quantifying the effects of system design decisions is one of the goals of this effort. Therefore, the environment must be consistent throughout the tests to ensure validity of measurements relative to one another.

- Common data types were specified for use while benchmarking. These were generally based upon IEEE-accepted standards.

- Baseline source code and data sets were provided with each benchmark. Baseline source code was not provided for stressmarks, though many teams used the example code from the specifications as a baseline.

2.1.2 Benchmark Measurement Procedures

The following guidance was given to teams for measurement of performance of a benchmark implementation:

- "Actual platform measurements are preferred over simulated results. It is understood that early iterations through the benchmarking process will necessarily be based on simulation, but these must give way to measurements of actual systems for reliable determinations to be achieved.

- "If simulations are used, a description of the model and tools used, and the bases for the timing values, should be provided.

- "All data sets should be used. They have been provided in a range of sizes, so as to test fixed-system scaling effects resulting from limited-resource optimizations. Should particular data sets be unusable for some reason (e.g., the data set requires more memory than that which is available), the reason should be reported.

- "There may be no recompilation or manipulation of the software or hardware between runs producing final measurements. Recall that quantifying the effects of system design decisions is one of the goals of this effort. Therefore, the environment must be consistent throughout the tests to ensure validity of measurements relative to one another.

- "Tests should be repeated enough times to ensure reproducibility.
- "As the DIS effort is primarily concerned with memory issues, measurement of time to perform I/O operations shall ideally be factored out. However, because the relative need for–and speed of–I/O is determined by the architecture, these times should be measured and included in the report. If possible, the time for these operations should be noted, so they can be excluded when appropriate."

2.1.3 Required Elements of Results Submission

Participants were expected to supply the following items as a result of their tests:

Item	Description
Architecture description	A detailed description of the hardware and software environments utilized during testing should be supplied. The description should be sufficient that strengths and weaknesses of the architecture pertinent to the benchmarks can be understood. Known performance measures such as bisection bandwidth and feature size should be included. Limits of the architecture (e.g., maximum of 32 processors, or maximum clock rate of 100Mhz) should be identified, and if predicted performance is to be considered, it must be justified in the *Comments* section of the report. As it is unwise to compare raw timings, even for similar architectures, without considering the differences in technology between the systems, this description is critical to the process, and should be organized, detailed, and complete.
Source code	If modifications are made to the baseline source code in support of optimized performance, the revised source code used during testing should be supplied, along with corresponding documentation of the changes, and detailed documentation of the code compilation, assembly, and execution.
Implementation documentation	A detailed record of the implementation, including rationale and approach to optimizations, is expected. This is particularly important when deviations from the baseline code are employed, or when problems in implementation are encountered. An accurate account of the labor required to implement each benchmark is required.
Output data	Output data sets should be made available. Any deviations from the output data specification should be explained.
Measurements	Performance figures for each applicable benchmark should be supplied, along with a description of how they were obtained. Any missing measurements should be explained. Metrics in addition to those required by this specification are encouraged, but they must be accompanied by documentation of how they were gathered, and how they are pertinent to the analysis.
Comments	Participants are encouraged to include any other information pertinent to the benchmarking process, including explanations of special circumstances, or recommendations for improving the benchmark. To be considered, theoretical performance of an unbuilt architecture should be given and justified. Particular attention should be given to the scalability of the architecture with respect to each of the benchmarks in the suite. Results from implementations of other benchmarks are welcomed, also, though these should be sufficiently delineated so as not to obscure the data directly relevant to this benchmark.

Specific metrics relating to individual benchmarks were also developed. More information can be found in the benchmark specifications.

2.2 Sources

All results discussed in this document are based on information delivered to us as part of the DIS effort. Where not otherwise cited, information came from project reports, project status reviews, and Principal Investigator meetings.

2.3 Selected Metrics

Work, efficiency, and gain, as used in this document, are defined below.

2.3.1 Work

This effort was strongly oriented toward application performance. A high processing rate was not considered valuable unless it was attainable in an application setting.

Due to this, and the extreme variety of proposed DIS solutions, we elected to measure work at a high level. The amount of work done for a given stressmark was determined by computing the number of abstract elements that would concern the programmer. We specifically avoided counting computations at the level of interest to the chip designer, for those values vary from system to system.

For our purposes, the work units generally would be counted as the number of iterations of an innermost loop given in the C language.

There was some initial objection to this, as some teams observed that a row-oriented access to an array would be counted as the same amount of work as a column-oriented access. We point out that any disparity between the two is architecture-dependent and fully introduced by the inefficiencies of memory systems. The goal of DIS included elimination of this disparity, and therefore it must not be represented in our work metric.

In addition, the amount of work is only utilized as a normalizing metric within the scope of each project separately; therefore, this definition creates no project comparison issues.

2.3.2 Efficiency

For approaches that do not include the modification of the processor or directly related hardware, it is valid to consider efficiency as a fundamental metric. Consider that if the processor were never starved for data, operation could be continuously maintained at peak rates.

For these approaches, the question of performance improvement can be formed by asking, "Has the approach reduced the data starvation of the processor?" Better still would be to approach from the other direction: "What happens if I increase the speed of the processor? How fast can I make it before the number of stalls becomes unacceptable?" Since designers purposely place this point beyond the limits of normal operation, it may not be obvious for a given off-the-shelf system.

Raising the efficiency of a system increases performance somewhat, but the true goal of this action is to allow even faster processors to be utilized. In other words, elimination of all cache misses for a given stressmark cannot be an isolated goal, because it does not necessarily relieve the overall limitations of the architecture.[13]

Ideal performance goals are only measurable using ideal benchmarks. To find efficiency using imperfect measurements, we must approximate ideal by scaling up from problem sizes that should execute at nearly peak rates. This is somewhat dangerous, since it presumes the peak observed processing rate is congruent with the true peak rate.

Nevertheless, some of the analyses in this document include efficiency measures based on computed processing rate (work per unit time), normalized by the peak observed rate for that same platform configuration.

---

[13] This is perhaps especially clear when one recalls that cache misses can be fully eliminated by simply slowing the processor.

2.3.3 Gain

In one form or another, most of the results given in this document have been arranged to show a performance gain versus a baseline. The actual element of performance is identified in each case. Most often, it is processing rate (work per unit time), but the more architecture-specific values of instructions per clock (IPC), and operations per second (OPS) were also used when execution times were not reported.

2.4 Other Assumptions

The summaries here assume correct execution of the stressmarks. In many cases this could not be verified.

Likewise, the timing performed by the participating teams is assumed to be accurate. Unfortunately, the conditions of measurement are not consistent between projects (e.g., some included OS overhead, some utilized rough simulations).

2.5 Benchmark Suite

The canonical description and specifications of the DIS Benchmark Suite can be found in *Data-Intensive Systems Benchmark Suite, Analysis and Specifications*[14]. This document cannot properly be interpreted without those specifications, and no attempt to reproduce them is made here.

However, this section does provide a summary of the benchmark algorithms and data as an introduction and quick reference. Additionally, commentary relating to the intent behind some of the requirements and insight about some of the data sets is provided.

2.5.1 Algorithm Selection

Although many classes of algorithms could benefit from systems with advanced memory or PIM elements, at the beginning of the DIS effort, three classes were identified that would provide a representative scope of achievable performance improvement for problems of interest to key DARPA programs:

- *Model-Based Image Generation* – This class includes generation of synthetic signatures and scenes for targets and terrain based on complex models of objects and sophisticated camera models for various sensor types. Applications include target recognition, real-time scene simulation for visualization or training, and model-driven change detection.

- *Target Detection* – This class includes spatial- and frequency-domain target detection in scenes collected from a wide range of sensor types. Applications include automated exploitation and cueing systems.

- *Database Management*– This class includes algorithms for index maintenance, storage management, and content-based query processing. Applications include sensor data archive management and geographic information systems such as the Dynamic Database for Battlefield Situation Awareness.

From these classes, five algorithms were selected–two from Model-Based Image Generation, two from Target Detection, and one from Database Management.

It deserves mention that at all stages of the program, all DIS participants were solicited for comment and suggestions relating to the benchmarks and methods. When candidate algorithms were tentatively selected, they were presented individually to each team prior to the final selection.

[14] http://www.aaec.com/projectweb/dis/DIS_Benchmarks_V1.pdf

2.5.2 Benchmark Data

Data sets were developed for use with the benchmark algorithms. The purposes of the data sets were:

- to provide uniform input for DIS benchmark tests;
- to provide a range of input so as to stress the memory-handling capabilities of DIS architectures and software;
- to provide correct output for comparative purposes; and
- to act as a minimal acceptance test for programs.

The purposes of the data sets were not:

- to test the robustness of DIS software;
- to debug DIS programs; or
- to precisely mimic real data.

To satisfy the above purposes, the following requirements were applied:

Requirement	Comment
Reflect and conform to purposes, procedures, specifications, and intent set forth in *DIS Benchmark Suite*.	The data must follow from the purposes set forth above and in *DIS Benchmark Suite*.
Valid input only	The purpose of this data set is not to test the robustness of the programs. Input data outside the specified valid range is not desired.
Correct output	"Correct" means as predicted by the specifications (not the in-house software).
Many tests for each benchmark	The benchmark specifications were developed consistent with the goal of performing a great deal of tests with minimal software development.
Each test consists of exactly one input file and exactly one correspondent output file	Formats are defined in the Specifications section of *DIS Benchmark Suite*.
Range of input sizes	Span 5-8 orders of magnitude; smallest set should fit in cache; largest should require > day on workstation.
Range of input 'difficulties'	Try to cover best case, worst case, and average case, if these can be known.
'Difficulties' reflective of real-world	The relative weighting of cases (best, worst, average) should favor typical cases, and not extremes.
Data realism only where needed; always serve the test.	Though real sensor or system data need not be utilized, some attempt must be made to ensure that the test data contains qualities of the real data where these qualities are pertinent to the test.
Each operation of each benchmark must be stressed	It is not necessary that each element be stressed independently of all others. For example, in a chain of filters, each filter should remove something from the data stream; otherwise, the data cannot be used as an acceptance test.

2.6 Method of Moments

The first class of algorithms chosen for inclusion in the DIS benchmark suite are Method of Moments (MoM) algorithms, which are frequency domain techniques for computing the electromagnetic scattering from complex objects. MoM algorithms require the solution of large dense linear systems of equations. Traditionally, MoM algorithms have employed direct linear equation solvers for these systems. The high computational complexity of the direct solver approach has limited MoM algorithms to low frequency problems. Recently, fast solvers have been introduced which have low computational complexity. The potential of these fast solvers to enable MoM algorithms to solve larger problems at higher frequencies is ultimately limited by the speed of main memory. Thus, fast MoM algorithms may benefit from the Data-Intensive Systems research effort.

In MoM algorithms the integral equation form of the Helmholtz equation is discretized by expanding the surface currents induced by the applied excitation in N basis functions. Then N test functions are used to convert the integral equation to a dense linear $N \times N$ system that takes the form $Z \bullet J = V$. Generally, N increases as the square of the frequency, and for typical problems, N is greater than 10,000. In traditional MoM algorithms, which first appeared in the late1960's, the dense linear system $Z \bullet J = V$ is solved by a direct linear equation solution algorithm, which may be composed as an in-core or out-of-core solver. On modern parallel computers, the direct solvers may be extended to work on shared or distributed-memory architectures.

The advantage of MoM algorithms is that they are exact representations of Maxwell's equations and highly accurate simulations are possible. The disadvantage of the traditional MoM algorithms is that the methods are computationally intensive, especially as the frequency goes up. The computational complexity of traditional MoM algorithms includes $O(N^2)$ integral evaluations to compute the matrix Z and $O(N^2)$ arithmetic operations to solve the system $Z \bullet J = V$ for J. The memory requirement for traditional MoM algorithms is $O(N^2)$. For these reasons, the traditional MoM algorithms are generally used only for low frequency problems. Although traditional MoM algorithms have been highly optimized on a variety of high-performance computing machines, the largest problems solved so far are for N on the order of 100,000.

Recently, new fast MoM algorithms based on fast, iterative linear equation solvers have been introduced. The iterative solvers rely on numerically stable and rapidly converging iteration procedures, such as the preconditioned GMRES method.[15] Fast matrix-vector multiply algorithms are used to compute products of the form $Z \bullet X$ used in the iterative procedure. The computational complexity of the fast MoM algorithms is $O(N\log N)$. The memory requirement for the fast MoM algorithms is $O(N)$. This is a remarkable reduction from the $O(N^2)$ computational complexity of the traditional MoM algorithms, and potentially, allows the solution of much larger problems at higher frequencies.

Rohklin[16] has introduced new fast MoM algorithms for the Helmholtz equation, which use iterative linear equation solvers and the fast multipole method (FMM) for fast matrix-vector multiplies. To compute products of the form $Z \bullet X$, the Z matrix is not formed or stored, rather the product $Z \bullet X$ is viewed as a field and approximately evaluated by the FMM. The mathematical formulation of the FMM is based on the theory multipole expansions and involves translation (change of center) of multipole expansions and spherical harmonic filtering. The computational complexity of these new methods is $O(N\log N)$ and the memory requirement is $O(N)$.

Building on the FMM approach, Dembart, Epton and Yip[17] at Boeing have implemented a fast MoM algorithm in a production grade electromagnetics code used by the company for radar cross-section (RCS) studies. Problems for

---

[15] Yousef Saad, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, Boston, MA, 1996.

[16] V. Rokhlin, *Diagonal Forms of Translation Operators for the Helmholtz Equation in Three Dimensions*, Research Report YALEU/DCS/RR-894, Dept. of Comp. Sci., Yale Univ., March, 1992.
R. Coifman, V. Rokhlin and S. Wandzura, "The Fast Multipole Method for the Wave Equation: A Pedestrian Prescription", *IEEE Antennas and Propagation Magazine*, 35, No. 3, June 1993, pp. 7-12.

[17] B. Dembart and E. L. Yip, *A 3-d Fast Multipole Method for Electromagnetics with Multiple Levels*, ISSTECH-97-004, The Boeing Company, December, 1994.

which the number of unknowns is on the order of 10,000,000 have been solved with this code. Boeing's fast solver uses the preconditioned GMRES iterative method, which requires only the calculation of products of the form $Z \bullet X$, combined with a multilevel FMM for fast matrix-vector multiplies.

The potential of the fast MoM algorithms to solve larger problems at higher frequencies, which results from their low computational complexity, is impacted by two memory bottlenecks encountered by fast solvers: low reuse of data and non-unit stride memory access.

We introduce the issue of low reuse of data by first considering the direct solvers used in the traditional MoM algorithms. Since the computational complexity is $O(N^2)$ and the memory requirement is $O(N^2)$ for direct solvers, the ratio of computation to data access is $O(N)$. For typical problems solved by the traditional MoM algorithms, where N is greater than 10,000, data reuse is high. When data reuse is high, cache is an effective tool for enhancing processor performance. The direct solver, in-core or out-of-core, can be organized so that a block of data is placed in cache and then reused from cache. This effective use of cache makes the computer perform as if all the memory is as fast as the cache memory. Similarly, direct solvers can be optimized for shared- or distributed-memory architectures.

For the fast MoM algorithms, where the computational complexity is $O(N\log N)$ and the memory requirement is $O(N)$, the ratio of computation to data access is $O(\log N)$. Indeed, implementation of Rohklin's translation theorems shows that for the translation operations, which are key to the FMM, the ratio of memory access to computation is 3-to-1. Thus, cache cannot be used to enhance processor performance, and the speed of fast MoM algorithms is ultimately limited by the speed of main memory.

In addition to the bottleneck resulting from the low reuse of data, fast solvers based on the FMM face a second memory related bottleneck. The FMM relies on the numerical implementation of spherical harmonic filtering. The filter operates on rectangular arrays of data in three stages. The arrays are accessed first by rows, then by columns, and finally, by rows again. In the second stage, it is necessary to access memory locations that are not consecutive. Thus, the speed of the fast MoM algorithms based on the FMM is ultimately limited by the speed of accessing main memory with non-unit stride.

Fast MoM algorithms, based on efficient iterative linear equation solvers, have the potential to compute the electromagnetic scattering from complex objects at frequencies 10 to 100 times higher than possible with traditional MoM algorithms. As pointed out above, the ultimate potential of these fast MoM algorithms is limited by two memory-related bottlenecks: low reuse of data and non-unit stride. For these reasons we have chosen to use Boeing's fast solver, based the preconditioned GMRES iteration method and the FMM for fast matrix-vector multiplies, as the basis for the *Method of Moments Benchmark.* The key FMM kernels represented in the benchmark are the translation operations and spherical harmonic filtering.

2.6.1 Method of Moments Data

Information was not available for this report.

M. A. Epton. and B. Dembart, "Multipole Translation Theory for the 3-D Laplace and Helmholtz Equations", *SIAM J. Sci. Comput.* 16, No. 4, pp. 865-897, July, 1995.

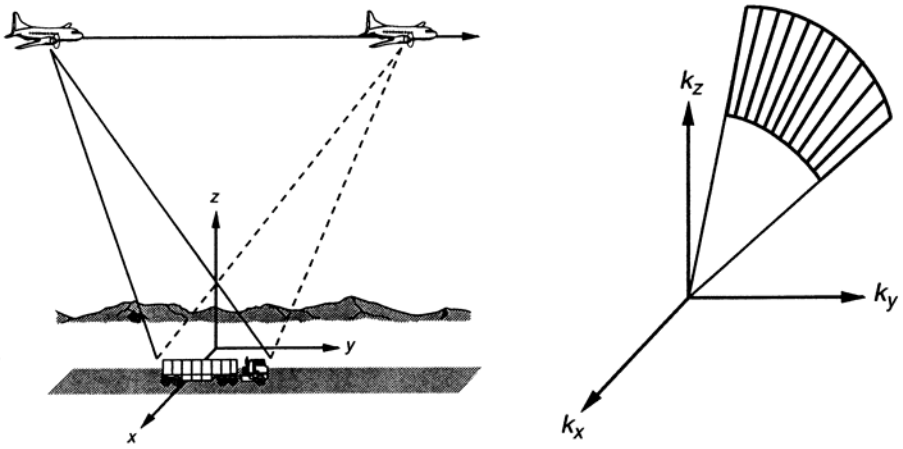
M. A. Epton and B. Dembart, *Low Frequency Multipole Translation Theory for the Helmholtz Equation*, SSGTECH-98-013, The Boeing Company, August, 1998.

M. A. Epton and B. Dembart, *Spherical Harmonic Analysis and Syntheses for the Fast Multipole Method*, SSGTECH-98-014, The Boeing Company, August, 1998.

2.7 Simulated SAR Ray Tracing

The simulation of Synthetic Aperture Radar (SAR) provides a cost-effective alternative to real data collections. In contrast to deployed sensors systems, whose operational parameters are fixed, computer simulations allow continuous variation of system and scene parameters. They have been used to simulate the performance of hypothetical sensors systems and to predict the signature of targets from a large number viewing angles as well as target signature that are inaccessible. These simulated target signatures have been used to design, test, and have been included as part of ATR systems.

Phenomenological models of targets and backgrounds and their interactions are the theoretical foundation of the computer simulations. For example, both image-domain and phase-history-domain approaches have been used to simulate synthetic aperture radar (SAR). The image domain approach uses a generalization of the physical optics approximation to compute target scattering. Such an approach is very amenable to use with a solid geometry target model sampled by ray casting. The phase history domain approach uses a variety of methods to compute target scattering: physical optics (PO), physical theory of diffraction (PTD), method of moments (MoM), and others. Hybrid implementations of these two methods have also been developed. The SAR simulation method analyzed for this benchmark is based on the image domain approach.



A typical airborne SAR collection geometry is illustrated in the figure above. Assume far field conditions and a narrow-band signal. Let α and β denote, respectively, the receive and transmit polarizations of the radar, $u(t)$ the transmitted waveform, and $\gamma(r')$ the so-called SAR reflectivity of the scene, The radar return signal is generally represented as

$$v_{\alpha\beta}(t) = K \int_S \gamma_{\alpha\beta}(r') u\left(t - 2\tfrac{R}{c}\right) ds'$$

where S is the illuminated portion of the scene, $R = |r - r'|$ is the distance from the radar to the point r' on S, c is the speed of light and K is a system constant. In essence, the model is based on the argument that the return from a differential surface element *ds'*, located at r', is a replica of the transmitted signal. This signal is delayed in time by the two-way propagation time from the radar to r' and back, and modified by the reflectivity of the surface element. It can be shown that such a model is consistent with physical optics, and an explicit formula for γ can be obtained.

It is customary to demodulate $v(t)$ by mixing with a reference signal $h(t)$, yielding

$$s(t) = h(t)v(t)$$

Let $\Gamma(f)$ denote the spatial Fourier transform of $\gamma(r)$:

$$\Gamma(f) = \Im\{\gamma(r)\}$$

A single range record *s(t)* can be interpreted as corresponding to the values of $\Gamma(f)$ over a radial line segment in the spatial frequency domain. The complete record of *s(t)* for a sequence of pulses transmitted and received at positions along the platform trajectory constitute a so-called phase history.

The fact that the collected data corresponds to the Fourier transform of the reflectivity density suggests that a reconstruction (image) of the reflectivity can be obtained by inverse Fourier transformation of the data. It will be at best a partial reconstruction because we have only partial data. For a linear trajectory, the phase history represents a planar surface in the spatial frequency domain over which the value of $\Gamma(f)$ has been sampled. The most that can be obtained is a two-dimensional image. Letting $A(f)$ denote a weighted, two-dimensional processing aperture over the data surface, the SAR image formation process is given by

$$g(r) = \Im^{-1}\{ A(f)\,\Gamma(f) \}$$

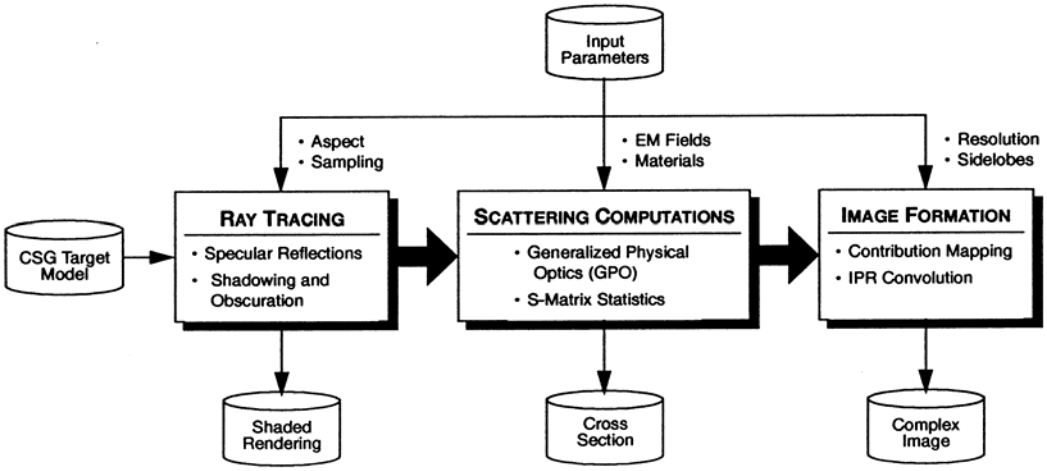
Additional insight is gained by noting that the image formation process is mathematically equivalent to the convolution

$$g(r) = a(r) * \gamma(r)$$

where

$$a(r) = \Im^{-1}\{ A(f) \}$$

is known as the spatial impulse response.



Simulation of the SAR system can be achieved by synthesis of the phase history followed by aperture weighting and inverse transformation, or by direct computation of the reflectivity density followed by convolution with the spatial impulse response. Both methods have been implemented in SAR simulation programs like X-PATCH and RADSIM.

In the process of analyzing this approach, the simulated SAR technique can be broken down into three steps as seen in the figure above.

First, is the process of sampling a scene database made of polygons, splines, and Constructive Solid Geometry. A ray tracing system is used to accomplish this sampling, by simulating how the radar energy bounces through the scene.

Ray tracing is a process where rays are fired from a viewing window into the scene and recursively traced through their specular reflections. Each ray is defined as vector with a starting position at the sensor and a direction defined by the pixel it passes through on the viewing window, in this case our synthetic aperture. Each ray is tested against all objects in the scene to see if an intersection can be found. The process of finding the intersection involves finding the roots of a system based on the combination of the vector and object. If multiple intersections are found the closest intersection is used. Once an intersection is found, the object ID and the intersection coordinates are recorded. In addition, several other properties at the intersection point are determined. These are surface normal, curvature, surface material type, and length of the ray from the intersection point to the origin of the ray. Using the surface intersection normal and the incoming ray, a reflected ray is calculated along the perfect specular reflection direction. This ray is fired from the current intersection point and the next intersection is found. This recursive process continues until either the ray leaves the scene, or a preset number of reflections are found. The intersection results of each original ray and all of its reflections create a ray history that contains all the intersection information, normally stored in a linked list. The output of the ray-tracing section is a ray history for each pixel in the image plane.

In programs like X-PATCH, the ray-tracing portion of the process consumes 50% to 60% of the total computation time. With this being the major time component in the SAR simulation process, it is a prime candidate for parallelization. Parallel ray tracing has been investigated by several researchers and is not a simple problem. This process will be the major thrust of the benchmark effort for simulated SAR imagery.

The second step is the process of converting the ray-traced information, the ray history, into the electromagnetic (EM) response of the sampled scene data. Here each ray path is analyzed to generate a fully polarimetric EM response solution. This is a linear process and does not consume a large amount of time. This step, in the SAR simulation, would be a trivial process to parallelize because each ray history is independent of all the others. Due to the small amount of time and the simplicity of parallelization, this portion of the process is not considered as part of the benchmark.

The final step in the simulated SAR process is converting the 2-D array of EM responses into complex images. This is accomplished by mapping the 2-D array of EM responses into the slant plane. This slant plane image is then convolved with a system Impulse Response (IPR) to form a complex image that can be detected and viewed.

This final step is a unique combination of processes, from the viewpoint of parallelization, and does present the second-highest consumer of CPU time. Creating a parallel version of this section of the process will stress data-passing, as EM responses are mapped onto a rectangular grid called the *slant plane*. This output then runs through a standard convolution. Each of these steps will require different lay-outs of memory and should present some unique problems as a parallel implementation. For this reason, and because this step is a large time consumer, it is part of the simulated SAR benchmark.

2.7.1 SAR Ray Tracing Data

Information was not available for this report.

2.8 Image Understanding

Image processing algorithms represent the third type of algorithm chosen for study. The applications of interest include target detection and classification. A sampling of these algorithms was chosen for this benchmark identifying bottlenecks that are common to image processing applications. The sampling contains algorithms that perform spatial filtering and data reduction. The algorithms selected for the benchmark are a morphological filter component, a region of interest (ROI) selection component, and a feature extraction component. These form the Image Understanding Sequence as shown in the figure below. The morphological filter component provides a spatial filter to remove background clutter in the image. Next, the ROI selection component applies a threshold to determine target pixels, groups these pixels into ROIs, and selects a subset of ROIs depending on specific selection logic. Finally, the feature extraction component computes features for these selected ROIs.

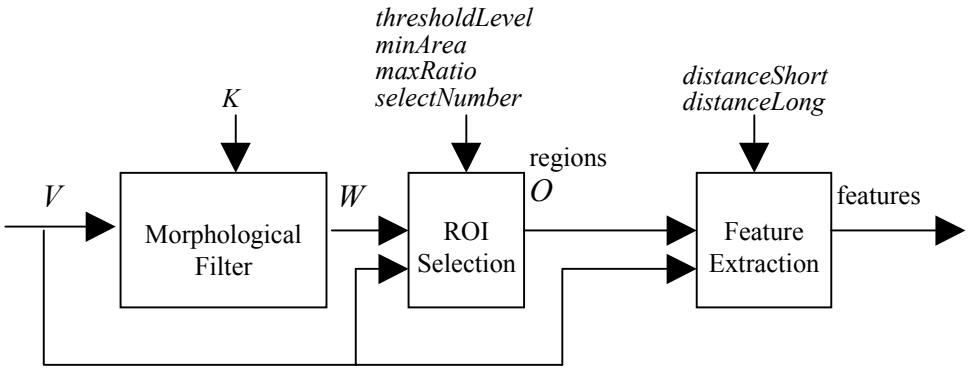


Image Understanding Sequence

Transformations that generate images from symbolic input, as well as Fourier Transforms, were excluded, since these are addressed in other portions of the Benchmark Suite.

The input required by the sequence is a set of parameters and an image, *V*. The first step in the sequence is a spatial morphological filter component generating image *W*. Then, the ROI selection component applies a simple threshold and groups connected pixels into ROIs (or targets) contained in image *W*. This component then computes initial features for each ROI in image *W,* and selects candidate ROIs, depending on the values of these features. These selected ROIs are stored in object image, *O*. The initial features for each selected ROI are stored in the list *regions*. Lastly, the feature extraction component computes additional features for the selected ROIs. The final output is a feature list, *features,* containing all the features calculated for each selected ROI.

Each algorithm has two associated costs: *operational* and *pixel addressing*. The operational cost is a measure of the computational burden placed upon the processors to execute the algorithm, and pixel addressing cost is a measure of the amount of memory usage or access that is required. A brief description and analysis of each component, including its bottlenecks, follows.

The morphological filter component chosen for the benchmark is a relatively straightforward procedure, designed to remove background clutter and retain objects of interest. The total cost of the morphological filter is determined by assuming the kernel is applied over the entire input image, although in practice the kernel is usually only applied over a subset of the image (the input image less a portion at the edges). The address-to-operation ratio is approximately the same for each approach. The filter utilized in this benchmark includes three distinct phases: erosion, dilation, and difference. The number of operations for the filter is

$$\text{size}(V)[4\,\text{size}(K)+1]$$

where V is the input image, K is the kernel, and size(X) is the total number of pixels in X. The operational cost consists of two multiplies, one subtraction, one minimum comparison, and one maximum comparison. The number of pixel addresses is

$$\text{size}(V)[4\,\text{size}(K)+5]$$

where the kernel and input image are accessed multiple times as the kernel is applied over the input image. The address to operation ratio is then

$$(4\,\text{size}(K)+5)/(4\,\text{size}(K)+1)$$

which is bounded in the range {1, 1.8}.

The ROI selection component of the sequence involves a threshold phase, a connected-component phase (where detected pixels are grouped into objects), an initial feature extraction phase, and a selection phase (where ROIs are selected based on the values of the initial features). The initial feature extraction phase measures five characteristics of the object region. Three of these–*centroid, area*, and *perimeter* –are descriptive of the shape and location of the ROI. The other two–*mean* and *variance* –are statistical measures of amplitude over the pixel population of the ROI. The threshold phase has an address-to-operation ratio of two. The operational and pixel addressing costs associated to the connected component phase, the initial feature extraction phase, and the selection phase vary greatly, depending on the implementation and the data involved, so no analysis of these costs is provided here.

After selecting ROIs, additional features are calculated. These give a rough measure of the texture of each ROI. A gray-level co-occurrence matrix (GLCM) contains information about the spatial relationships between pixels, by representing the joint probability that a pixel with a given value will have a neighboring pixel at a particular distance and direction with another chosen value.[18] Since this matrix is square, with dimensions equal to the number of possible pixel values, it provides more information than can easily be analyzed. Statistical descriptors of the co-occurrence matrix have been used as a practical method for utilizing these spatial relationships. Furthermore, Unser designed a method of estimating these descriptors without calculating the GLCM, instead using sum and difference histograms.[19] The descriptors chosen as features for this benchmark are *GLCM entropy* and *GLCM energy,* and are defined in terms of a sum histogram, *sumHist,* and a difference histogram, *diffHist.* These descriptors are calculated for each of two distances and four directions.

It is typical in target detection systems to calculate many features to be used in a target recognition step. The ideal is to choose the fewest and cheapest features possible that provide the best detection result. The cost for the feature extraction component is dependent upon the number of features or targets present in the input image which can range from zero to several thousand in typical applications. This makes the algorithm very difficult to execute efficiently, since many features will have a high computational cost with a small memory access cost, while a few will have a low computational cost with a high memory access cost. Thus, an *a priori*-implementation for feature extraction is generally not possible. Consequently, there is no analysis provided here of the cost involved to calculate these features.

The two main bottlenecks which occur in typical target recognition applications are the result of manipulations of large amounts of data while expending little computational effort, and of smaller amounts of data in computationally intensive functions. The intent of this benchmark is to represent these bottlenecks within the sequence, so that attempts to remove these bottlenecks may be examined.

2.8.1 Image Understanding Data

The following requirements were developed in anticipation of use of synthetic data.

---

[18] Parker, J., *Algorithms For Image Processing And Computer Vision,* Wiley Computer Publishing, 1997.

[19] Unser, M., "Sum and Difference Histograms for Texture Classification," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-8, 1:118-125, 1986.

Requirement	Comment
Input data format	See *DIS Benchmark Suite*, Section 4.2.4.1.
Phase component of complex input should be random.	The magnitude detection is a local calculation; there is no need to bias the statistics. This requirement implies that the input image should be "reverse-calculated" from a magnitude image.
Precision of magnitude and phase values should be to extent of floating-point number specifications.	Acceptance tests require precision to at minimum the specification put forth for numbers. Since inphase and quadrature values–which constitute the input of concern–are "reverse-calculated" when generating these tests, the results (magnitude) should be precise. See *DIS Benchmark Suite*, sections 3.6 and 3.7 for number specifications.
Image sizes (order N pixels)	10^3–10^9; at least 7 sizes, spaced more-or-less regularly.
Image shapes	Always rectangular upon input; one of the two dimensions should never exceed 4096 pixels; short dimension should be more-or-less evenly distributed between horizontal and vertical; warping rotation should be slight on images with very large (>>1) or very small (<<1) aspect ratios, to avoid extreme growth of rectangular storage requirements after warping.
Warping coefficients should always include components other than translation and scale.	Without rotation or distortion, warping can be a trivial matter of memory-stride adjustment (and some interpolation), which would defeat the purpose of this portion of the test.
Kernel and stencil 'shapes' should have 2-dimensional extent.	'Shape' refers to 'on' pixels only. Kernels and stencils should always have nontrivial extent in two dimensions, so that memory access for linearly stored images requires offset calculation.
Kernel and stencil sizes should vary generally in proportion to image size.	In real use, kernel and stencil sizes are determined by expected target sizes, sensor resolutions, and perhaps analyses of background data. Generally, we assume larger imagery implies higher resolution, and therefore larger kernels. There should be some variance of kernel sizes independent of image size, though, so that architectural dependencies on kernel sizes may be observed. Each kernel dimension should never exceed one fifth of its corresponding input image dimension.
Kernel shapes and sizes associated with pixel magnitude (target, background, noise) models	Morph operations should always filter something out of the image.
CFAR stencils convex, off-center, on-surround.	The way CFAR is defined for this benchmark, the 'background' pixels should be defined in a neighborhood around–but not including–the target pixel. There appears to be little value to the test to define the stencil shape in any other way.
CFAR threshold value set according to pixel model	Between 10^0 and $10^{\sqrt{n}}$ objects should result from each test, where 10^n is the number of pixels in the input image.
Varying output features	Centroid X & Y – even distribution (within image boundaries after warping and edge adjustment) Area – $10^0 \leq$ area $\leq$ imageArea/5, Gaussian

	Perimeter – avoid excessively high values (compared to area) indicating highly complex shapes. (Note that when objects are small, perimeter values will always be high relative to area values, due to quantization effects.) Mean – necessarily biased toward bright values Variance – arbitrary distribution
Output data format	See *DIS Benchmark Suite*, Section 4.2.4.3.

Ultimately, suitable imagery from physical sensors was found which satisfied the needs of the benchmark. Synthetic images were not generated.

2.9 Multidimensional Fourier Transform

The Fourier Transform has wide application in a diverse set of technical fields. It is utilized in image processing and synthesis, convolution and deconvolution, and digital signal filtering, to name a few. In fact, the transform is utilized within both the Ray-Tracing and Method of Moments benchmarks described elsewhere in this document. However, special interest in the Fourier transform merits its independent inclusion in this benchmark suite. Specifically, the interest is in the nature of the memory access patterns, which are indicative of a large class of problems.

The multi-dimensional Discrete Fourier Transform (DFT) is defined as

$$F(n_1, n_2, \ldots, n_N) = \sum_{k_N=0}^{N_N} \cdots \sum_{k_1=0}^{N_1} e^{2\pi i k_N n_N / N_N} \cdots e^{2\pi i k_1 n_1 / N_1} f(k_1, k_2, \ldots, k_N) \qquad \text{Eqn. 1}$$

where f is an input complex multi-dimensional array of size $N = N_1 \times N_2 \times \cdots \times N_N$, and F is the output forward transform of f. The Fourier Transform is rarely implemented directly as Equation 1, since the process would require $O(N^2)$ operations. Instead, the transform can be accomplished in $O(N\log_2 N)$ operations, or less, using one of a series of methods generically called Fast Fourier Transforms (FFT). These FFT methods exploit one or more mathematical properties of Equation 1 to reduce the required number of operations.

The bottleneck associated with DFTs that is of interest here is the non-unit-stride memory access associated with the transform. Part of the subscripts of Equation 1 can be "pulled out" of the summations (i.e., the exponential with the subscript k_N can be pulled outside of the sum over k_{N-1} etc.), which shows that the multi-dimensional DFT can be represented by a series of one-dimensional DFTs:

$$F(n_1, n_2, \ldots, n_N) = F_N\left(F_{N-1}\left(F_{N-2}\left(\cdots F_1\left(f(k_1, k_2, \ldots, k_N)\right)\right)\right)\right) \qquad \text{Eqn. 2)}$$

where F_k is a one-dimensional DFT over the specified index. The aspect of Equation 2 to note is that for whatever memory access the inner loops attempt, the outer loop will always be "opposite" or irregular, which prevents a unit-stride access. Rearrangement of the summations or manipulation of the equations can alleviate this memory access bottleneck to some extent, but some non-unit-stride access is present with most DFT implementations.

In order to simplify the implementation and specification of this benchmark, the DFT is limited to three-dimensional transforms only. The implementation of a 3-D transform is complex enough to give an indication of the performance of the architecture on higher dimensional transforms, but simple enough to be relatively easy to implement. The inclusion of one- or two-dimensional transforms would not significantly add any other performance information regarding the candidate architectures. In addition, one- and two-dimensional input can be approximately tested by specifying the length of the remaining dimensions of the array to one.

2.9.1 Discrete Fourier Transform Data

2.9.1.1 Derived Requirements

Requirement	Comment
Input data format	See *DIS Benchmark Suite*, Section 4.2.3.1.
Problem size trend consistent within test	For some sequences, the problem size should grow; for others, it should shrink.
Number of dimensions = 3	All problems are 3-D, but some have one or two dimensions of size=1; those are effectively 1- or 2-D.
Total size of each set large	Each sequence should be enough to keep the system busy for a long time; there is no need to make small sequences. Some variability is useful simply so that users can test sequences that do not require tremendous quantities of resources.
Number of repeats inversely proportional to volume size	Each image should be repeated enough times to keep the processor(s) busy for a significant period. For very large images, the number of repeats necessary to do this diminishes. Each volume of the test should be repeated such that the total number of points done for that volume is as close as possible to 10^7.
Size of each non-unity dimension of volume on order of between 10^2 and 10^4.	There is little point testing very small or oblong transforms. Similarly, volumes very large in one dimension are likely to be very small in another.
No more than 20 volumes in each set.	Rather than add more volumes to make a set larger, make the volumes larger.
Output data format	See *DIS Benchmark Suite*, Section 4.2.3.3.

2.9.1.2 High-Level Data Design

Model Type	Name & Description
Input Size	M $N \approx 10^6$. G $N \approx 10^9$. T $N \approx 10^{12}$. Where N is defined as the sum of the products of the number of repeats for each input volume and the number of points in that input volume.
Dimensions	1 Two of three dimensions are size=1. 2 One of three dimensions is size=1. 3 Zero of three dimensions are size=1.
Trend	None Volume size is arbitrary. Grow Volume size always increases. Shrink Volume size always decreases. (This should be handled by sorting, rather than by restricting sequential randomizations.)

2.9.1.3 Test Sets

Set Name	Input Size	Dimensions	Trend	Comments
FT1	M	3	None	
FT2	G	2	Grow	
FT3	G	3	Shrink	
FT4	T	2	None	
FT5	T	3	Grow	
FT6	T	3	Shrink	
FT7	G	2 (20Kx800x1)	None	Special test for GPS calculation; all volumes have one dimension of exactly 20K, one of exactly 800, and one of exactly 1.
FT8	T	2	None	Identical to test FT4, except no repeats.

2.10 Data Management

The fifth area in which the DIS benchmark suite attempts to measure performance improvement is in data management, specifically in the area of DBMS. Applications for traditional DBMS have been dominated by archival storage and retrieval of large volumes of essentially static data. Some newer applications, such as the Dynamic Database for Battlefield Situation Awareness, demand management of complex, dynamic indices in addition to the data.

The objective of this benchmark is to measure the performance improvement of a given hardware configuration for certain elements of traditional DBMS processing. Performance improvements due to sophisticated database design or special software implementation are avoided and not intended to be part of the benchmark. This benchmark focuses on two weaknesses of conventional DBMS implementations: index algorithms and ad hoc query processing.

Large volumes of data in a DBMS are typically referenced by an index structure. The index can be used instead of brute-force searches over all the data when a query is made. The index defines one or more elements of the data entries as key values. Thus, the key values are specified in advance, and the DBMS maintains a separate index structure based on them. The index is used to accelerate query processing by minimizing the amount of data that must be accessed to satisfy the query.

Two assumptions typical of conventional algorithms are that the data will be predominately static, and that operation can be suspended for index maintenance. Neither assumption holds for the Dynamic Database or other dynamic information systems, and current applications drive standard indexing schemes into frequent wholesale index regeneration, yielding unacceptable performance.

The index structure allows efficient searches over a database when the query can use a pre-defined key value. Queries that do not use a key value are called ad hoc, non-key, or content-based queries. This query type requires a brute-force search over all database entries. Conventional applications usually process an ad hoc query in two stages: an index-based search is used for the index keys in the query formulation, if any, and brute-force search is performed on the results of the index-based search. These brute-force searches are a bottleneck in a typical DBMS. The performance impact of non-key queries can be reduced by parallel searches of the data, which may be applicable to specific hardware architectures, or by partitioning the data.

Partitioning schemes provide an additional performance boost for a general database design where the primary objective is to separate areas of the database into logical sections, each of which is then indexed by its own scheme. The partition allows more efficient searches, when the sections have been chosen well, or when an optimal scheme is known *a priori*. It also supports parallel searches across partitions.

Bottlenecks traditionally associated with DBMS primarily occur in query processing, and the majority of work done to enhance performance has been in this area. Much of this query optimization has increased the query response speed at the expense of maintaining the index over the lifetime of the database. By definition, an index requires an

increase in overhead or up-front processing in favor of quicker, cheaper searches. Typical command operations such as *insert* and *delete* have generally not been optimized. This reiterates the implied assumption of the existence of periods during operation when user interaction can be suspended to deal with index management. The cost associated with index management over the operation life of the database represents a new measure of performance for advanced data management applications, and a corresponding new bottleneck.

The indexing method chosen for use within this benchmark is an R-Tree structure. The R-Tree index allows the key to represent spatio-temporal data, which makes the R-Tree particularly applicable to geographic information; database vendors commonly support it. The R-Tree structure is as close to a de-facto standard for representing such data in a database context as exists today.

The R-Tree index is a height-balanced tree containment structure, that is, nodes of the tree contain lower nodes and leaves. Thus, the tree is hierarchically organized and every level in the tree provides more detail than the previous level. The indexed data object is stored only once, but because of the containment structure, keys at all levels are allowed to overlap. This may cause multiple branches of the R-Tree to be searched for a query whose search index intersects multiple nodes.

A general measure of index maintenance cost for separate command operations is the number of node accesses required for each operation. Other measurements of cost become increasingly software-dependent, and are avoided in this analysis. A generic R-Tree implementation, which is given later in this document, has three command operations to measure: insert, delete, and query. Because the R-Tree is a height-balanced structure, the total number of paths for a full tree is given by:

$$N = \sum_{k=1}^{h} 2^{k-1} F$$

where N is the number of paths, h is the height of the tree, and F is the *fan* or *order* of the tree. Traditional performance measures have focused on the query response: for the generic R-Tree the minimum number of node accesses is h, which is expected from a height-balanced tree, and the maximum number of node accesses is N, or a complete node search over all possible paths. The maximum number is unique to the R-Tree or similar overlapping index trees and represents a significant bottleneck. The problem is exacerbated for improperly managed index structures, and can be alleviated by efficient software implementations and improved hardware architectures that allow more efficient or parallel searches.

Index management over the operation of the database represents a new type of bottleneck for advanced applications. The cost of maintaining the index can be estimated in the same manner as for query commands, by determining the number of node accesses required to complete the command in both the best and worst cases. A descriptive estimate of the average case is also given, with the caveat that the average case is highly implementation-dependent, and will vary for each system.

The insert operation has three separate phases: a search over all paths, insertion (which may cause node splitting), and index key adjustment. The best case occurs when insertion does not require node splitting and no parent keys need to be adjusted; this yields a cost of N node accesses. The worst case does require splitting along each parent, and all parent keys are adjusted; this yields a cost of $N+2h$ node accesses. An average insert would tend to require parent key adjustment and periodic node splitting. Thus, the average insert cost would tend towards the maximum cost.

The delete operation has two phases: a search for the data to be deleted, and a possible key readjustment. The best case has a cost of h node accesses, which represents no key adjustments and an immediate "one" path search for the data. The worst case has a cost of $N+h$, which represents a full search of the data and an all parent key adjustment. The average cost of a delete operation tends to the minimum case, since the operation would include key adjustment but probably not a full search.

The costs of the insert and delete operations are greater than or equal to the query operation in both the best and worst cases. Thus, index management over the operational life of the database represents a significant performance bottleneck when the data is dynamic.

This benchmark has been developed to measure the performance improvements of new hardware architectures for both index maintenance and non-key queries, which represent the two significant performance bottlenecks. One goal is to remove or "level" the algorithmic component over all of the architectures, without preventing any new or unique software implementations that would allow a significant performance improvement due to exploitation of special hardware features. This is done by defining the benchmark as the implementation of a highly simplified database with a specific index structure. The database supports only three simple aggregate data objects whose primary difference is in size. The use of different sizes of data objects is intended to prevent optimization of the implementation for an object of a specific size, and the sizes themselves were chosen to prevent similar multiples. The objects are aggregate in that they contain a set of data attributes or parts that are linked together as a list. An ad hoc query uses an attribute of the object for non-key searches. This type of search with simplified objects is relatively simple to implement, but is representative of more complicated database behavior such as object traversal. This benchmark requires the use of the R-Tree structure, but the participant is encouraged to modify or develop additional implementations tailored for new architectures.

The DIS benchmark metrics provide measurements of the candidate architecture's ability to handle the "highest" load when the number of users is large and the system resources are taxed to their limits. The benchmark simulates this maximum resource utilization by issuing the index commands in a batch rather than a stream mode. A stream mode would more closely mimic a "real" DIS application, allowing for multiple users and possible "down" time for index maintenance. However, this benchmark is primarily interested in the extreme condition, where down-time, in which a database can perform index maintenance with no cost to the users, is assumed not to exist. The performance on successful completion of the entire data set with its multiple commands is the primary metric of this benchmark, and this must include the time required for index maintenance since this will directly affect the users under extreme conditions. Participants are allowed to introduce artificial lags to the command input to simulate a stream mode, but the times reported for individual command completion and overall set completion must include the added lag times.

2.10.1 Data Management Data

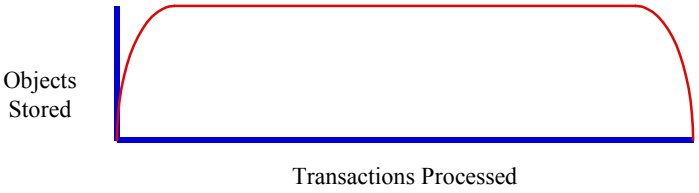
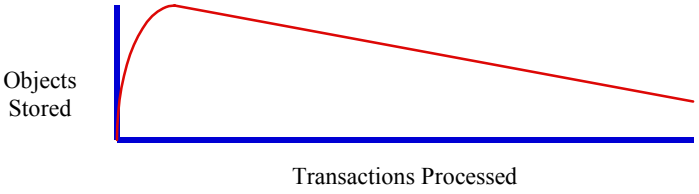
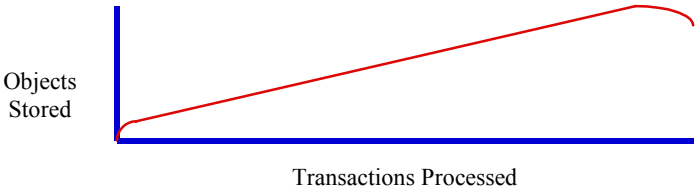
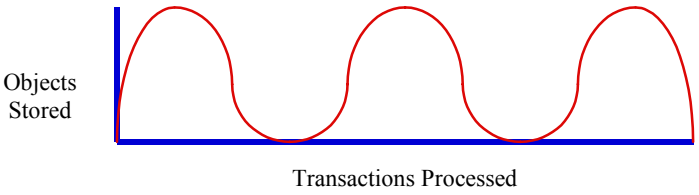
2.10.1.1 Derived Requirements

Requirement	Comment
Input data format	See *DIS Benchmark Suite*, Section 4.2.5.1.
Distribution of small, medium, and large objects is statistically flat.	It was initially felt that small data sets should contain mostly small objects, and large data sets should contain mostly large objects. However, since the object "size" really has to do with number of attributes (and not necessarily the memory occupied by the object), that requirement has no real meaning or need. Instead, small, medium, and large objects are distributed evenly across the population.
Transaction list lengths	10^3–10^9; at least 7 sizes, spaced more-or-less regularly.
Number of data objects	Peak at 1-3 orders of magnitude less than number of transactions.
Hypercubes	Hypercubes should be sized and shaped in proportion to the entire database volume as if they represented military vehicles or (occasionally) buildings in a theater of battle. Object hypercubes are expected to overlap due to reporting and quantization errors, but would not overlap by more than 50%, except in rare cases. Unusually shaped hypercubes should occur, but should be similarly rare.
Vary the "fan" parameter	For some tests, a set of transactions should be held constant while

	the fan parameter varies between tests. Otherwise, generally reasonable fan values should be used (fan value choice is influenced most by size and overlap of hypercubes). However, keep in mind that the purpose of the test is to measure performance on index maintenance, not to emulate a realistic database. Small fan sizes force larger indices for same-size databases.
Non-key (string) attributes need to vary.	There needs to be sufficient variance such that an implementer cannot simply use markers in place of actual storage of attribute contents. Actual character values of attributes may be random, though known substrings must be included at a random locations within the string, so that searches on these attributes can reliably find multiple answers. At least one known substring is required for each attribute; more are preferable.
Lengths of string attribute values should vary.	Variance here allows attribute contents to be stored in variable-length records, if desired. Median lengths would be about 1/8 of maximum, with a relatively small variance. Long strings, though rare, are needed to ensure specifications are met.
No more than one query per insert or delete; actual ratio to vary	In practice, this may not be a realistic proportion. Although the proportion is significant to database design consideration, here we have specified the tree structure and the fact that tree maintenance is the stressmark of interest. Therefore, insert and delete operations are of primary concern; queries are secondary, and are used mostly to provide verification in the form of output. The actual ratio of operations should vary between and within tests.
Bias transaction order for insertion and deletion on a test-by-test basis	Input streams can affect the frequency of redistribution due to the order of transactions. Some orderings will result in more index management for a given implementation than other orderings. Where possible, vary the bias of the orderings on a test-by-test basis, as opposed to within a given test, so that an implementation's reaction to this bias is measurable.
At least one non-key query per test	Most queries will use at least some keys, but all tests should ensure that at least one query that is not based on any key values is issued.
Total volume of DB area must not exceed value that can be contained by float.	Various elements of the R-tree implementation must calculate the volume of a hypercube; the result should fit within a single-precision float. The root node will indicate a hypercube of the entire DB volume.
Output data format	See *DIS Benchmark Suite*, Section 4.2.5.3.

2.10.1.2 Model Design

Model Type	Name & Description
Input size	3: Approximately 10^3 transactions 4: Approximately 10^4 transactions 5: Approximately 10^5 transactions

	6: Approximately 10^6 transactions 7: Approximately 10^7 transactions 8: Approximately 10^8 transactions 9: Approximately 10^9 transactions ("approximately" here means $\pm 10^2$)
Peak tree size	Small: Approximately (Input size / 10^3) objects Large: Approximately (Input size / 10^1) objects ("approximately" here means $\pm$(Input size / 10^5))
Tree size profile	(All profiles general only; small fluctuations are assumed; DB always starts with zero entries, but may end with many.) 1: Full size.  2: Full size early, then taper.  3: Slow growth.  4: Very large fluctuations. (Number and character of fluctuations not necessarily represented by diagram.) 
Tree size fluctuations	Minor: Not more than 5% (of number of objects) deviation from profile. Major: Not more than 20% (of number of objects) deviation from profile.
Tree fan	2: Fan = 2. 3: Fan = 3.

	4: Fan = 4. 8: Fan = 8.
Object hyper-cube overlap	None: Object hypercubes never overlap. Low: Object hypercubes overlap at the time of their insert on average once per 25 object insertions. High: Object hypercubes overlap at the time of their insert on average once per 5 object insertions.
Query rate	Low: Average one query in 30 transactions. Med: Average one query in 10 transactions. High: Average one query in 2 transactions.

2.10.1.3 Test Sets

There are 4032 possible permutations of the seven models. This many tests would be excessive, but finding a subset that adequately represents the whole is important.

Of the possible permutations, some are meaningless. For example, the Tree Size Fluctuations model would be meaningless for any test that has a Tree Size Profile of '4' (indicating very large fluctuations). Similarly, very large transaction sets generally would not utilize small fan parameters, and vice versa. Hence, we eliminate tests that include the following combinations:

- Tree size profile = 4; Tree size fluctuations = Minor
- Input size = 3; Tree fan = 4,8
- Input size = 4; Tree fan = 8
- Input size = 4; Peak tree size = Small; Tree fan = 4
- Input size = 6, 7, 8, 9; Tree fan = 2
- Input size = 8, 9; Tree fan = 3
- Input size = 3, 4, 5; Peak tree size = small
- Input size = 7, 8, 9; Peak tree size = large
- combinations = 1323

Following these eliminations, there are still 1323 possible permutations. The remaining set is therefore selected such that there is representation across the spectrum of input sizes and other parameters, without necessarily providing a datum for every permutation.

Set Name	**Input size**	**Peak tree size**	**Tree size profile**	**Tree size fluctuations**	**Tree fan**	**Object hyper-cube overlap**	**Query rate**
	3, 4, 5, 6, 7, 8, 9	*Small, Large*	*1, 2, 3, 4*	*Minor, Major*	*2, 3, 4, 8*	*None, Low, High*	*Low, Med, High*
	(7 of 7)	*(1 of 2)*	*(3 of 4)*	*(1 of 2)*	*(2 of 4)*	*(1 of 2)*	*(1 of 2)*
DM1	3	Large	4	Major	2	None	Med
DM2	3	Large	4	Major	2	None	High
DM3	3	Large	3	Minor	2	None	Low
DM4	3	Large	1	Minor	2	Low	Med
DM5	3	Large	1	Minor	3	Low	Med
DM6	3	Large	2	Minor	3	None	Med
DM7	4	Large	3	Minor	2	Low	High
DM8	4	Large	1	Major	2	None	Low

DM9	4	Large	1	Minor	2	None	Low
DM10	4	Large	4	Major	2	High	Med
DM11	4	Large	2	Minor	2	High	Low
DM12	4	Large	2	Minor	3	Low	Low
DM13	5	Small	2	Major	2	High	High
DM14	5	Large	2	Minor	2	Low	Low
DM15	5	Small	3	Minor	3	None	High
DM16	5	Large	3	Minor	3	Low	Low
DM17	5	Small	4	Major	8	None	High
DM18	5	Large	3	Major	8	None	High
DM19	6	Small	4	Major	3	Low	High
DM20	6	Large	2	Major	3	High	Low
DM21	6	Small	1	Major	4	High	High
DM22	6	Small	2	Minor	8	None	High
DM23	6	Large	1	Major	8	High	Med
DM24	6	Large	3	Major	8	High	High
DM25	7	Small	4	Major	3	None	High
DM26	7	Large	1	Minor	3	High	Med
DM27	7	Large	2	Minor	3	Low	Med
DM28	7	Large	2	Major	3	Low	Low
DM29	7	Small	2	Major	8	None	High
DM30	7	Small	4	Major	8	High	Med
DM31	8	Small	3	Major	4	Low	High
DM32	8	Small	3	Minor	4	High	Low
DM33	8	Small	3	Major	4	Low	High
DM34	8	Small	3	Minor	4	Low	Med
DM35	8	Small	1	Major	4	Low	Low
DM36	8	Small	4	Major	4	High	Med
DM37	9	Small	4	Major	4	Low	Med
DM38	9	Small	2	Major	4	None	Med
DM39	9	Small	4	Major	4	High	Low
DM40	9	Small	3	Major	8	High	Low
DM41	9	Small	1	Minor	8	None	Med
DM42	9	Small	1	Minor	8	High	Low

2.11 Stressmarks

Following the release of the Benchmark Suite, it became apparent that additional benchmarks would be required to assist in satisfaction of the needs of some program participants. A set of smaller, more specific procedures was desired. These *stressmarks* would more directly illustrate particular elements of the DIS problem, and require less energy to implement, perhaps at the expense of reduced realism in certain areas. This section gives pertinent points extracted from the stressmark specifications.[20]

2.11.1 Purpose

The development of new architectures and approaches for data-intensive computing could be beneficial to many problems of interest to DARPA. Evaluation of the approaches in the context of those problems is essential in order to realize those benefits.

Equally important, the existence of simplified—but meaningful—programs derived from defense applications can provide valuable input to the development process.

On these premises the benchmarks were developed. While the purpose of the Benchmark Suite included the need to realistically represent certain applications, this sometimes led to difficulty in their implementation. Though the benchmarks represented the essence of the data-intensive processing stripped from the source application, they still required a significant amount of code in order to be complete. The target for the benchmarks was about 1000 lines of code, derived from applications with hundreds of thousands or millions of lines. The target for these stressmarks is on the order of dozens of lines.

To reduce the code without eliminating the data-intensive nature of the problem required a focus on elementary segments of the problem. This implies that results should be worse for general-purpose machines, and architectures developed for the "data-intensive" problem should show an even greater improvement. These results, however, must necessarily be less indicative of normal machine operations, and should be interpreted accordingly.

The benchmarks were aimed specifically toward focused code that retained the context of the enveloping application. This approach attempted to avoid the pitfalls associated with 'kernel'-oriented benchmarks, which often indicate performance figures that are unattainable in typical operation.

Stressmarks are necessarily less realistic than benchmarks. Benchmarks are necessarily less realistic than true applications. Participants are cautioned not to allow concentration on stressmarks to interfere with development of machines and software that performs well on real applications.

The critical point for the development of the stressmarks, therefore, is to support the benchmarks, rather than to replace them.

2.11.2 Basic Requirements

One of the major goals of the benchmark suite was to retain a certain degree of application-level orientation, since some elements of data-intensive problems cannot be reliably measured using isolated kernels. The motivation behind the stressmarks, therefore, is to supply problems that are derivative of those offered in the benchmarks, but in the form of isolated kernels. Consequently, the original considerations for stressmarks were:

- Operations should be nominally representative of the data-intensive kernels within the benchmarks.
- Input and output should be limited, and only occur at the very beginning and end of processing tasks.
- Total memory required should be varied, much as the "problem size" was varied for the benchmarks.
- Acceptance and verification tests become difficult to specify, since I/O is limited.

---

[20] Complete specifications can be found in *DIS Stressmark Suite*, http://www.aaec.com/projectweb/dis/DIS_Stressmarks_V1_0.pdf.

- A random number generator and initial seeds should be used to generate input.
- Ideally, stressmarks can be run in a variety of modes corresponding to data types and precisions.
- If possible, parallel (MPI or threaded) versions could be supplied.
- Total size of source code should be extremely small.
- Represent operations on sparse, dense, regular, and irregular data. Attempt to allow parallelism coarse, fine, or both levels.
- Metrics collected should be similar to those collected for the Benchmarks.

2.11.3 Development

Considering the above requirements, a series of stressmarks were proposed, and development began. During this period, it was observed that the bulk of the operations did not modify their fields of data. A new one was created to address this specifically, and others were modified to require memory writes, as well.

It had been observed that the benchmark suite minimized the effects of data starvation by capitalizing on the work of clever programmers. A goal during this development was to avoid the possibility of simple reinterpretation of the problem resulting in reduction of data starvation. To our surprise, this was not an easy task. Repeatedly, our software engineers were able to find ways to change the program steps in such a way that the bulk of the intended problem was reorganized to be more memory-friendly. While this may merely be the consequence of attempting to specify problems that only require a dozen lines of code to solve, it could indicate that the problem space is more limited than initially thought, when a supply of engineering labor is available. In either case, several of the stressmarks required modification due to this issue.

2.11.4 Stressmark Suite

The final suite included seven individual stressmarks, though their results were intended for collective interpretation. This table summarizes them; each is individually described in later sections.

Stressmark	*Problem*	*Memory Access*
Pointer	Pointer following.	Small blocks at unpredictable locations. Can be threaded.
Update	Pointer following with memory updates.	Small blocks at unpredictable locations.
Matrix	Conjugate gradient simultaneous equation solver.	Dependent on matrix representation. Likely to be irregular or mixed, with mixed levels of reuse.
Neighborhood	Calculate image texture measures by finding sum- and difference-histograms.	Regular access to pairs of words at arbitrary distances.
Field	Collect statistics on large field of words.	Regular, with little re-use.
Corner-Turn	Matrix transposition.	Block movement between processing nodes with practically nil computation.
Transitive Closure	Find all-pairs-shortest-path solution for a directed graph.	Dependent on matrix representation, but requires reads and writes to different matrices concurrently.

2.11.5 Which stressmarks?

The seven stressmarks are largely complimentary and should be considered as a set. The true value of the stressmarks is realized when their metrics are evaluated collectively. Evaluation of metrics for any one stressmark independently of the others must be performed with caution. Participants were encouraged to run all of the stressmarks.

2.11.6 Code organization

Most code examples supplied with the specifications demonstrated the function of both the data generation and the data processing segments of the process. Typically, only the latter of these was deemed of interest, and often, the former may be executed on a separate host platform or on an offline basis. Thus, participants were warned to be wary of which algorithmic actions were subject to metric collection, and which were not.

2.11.7 Code optimization

Any code found within the stressmark specifications is for purposes of example only. There is no 'baseline' code. Hence, there is no restriction on code optimizations, nor is there a requirement that example code operate successfully without modification. Participants were expected to generate the best implementations possible, using methods they deem appropriate.

Generation of multiple implementations of stressmarks, each one optimizing a different performance characteristic, was encouraged. For example, one might optimize for maximum throughput in one case, minimum memory storage in another, and minimum power consumption in a third. When doing this, participants were reminded to collect results using as much of the supplied data sets as possible, considering each instance of each stressmark separately. The performance trade-offs of design decisions was to be visible in the results, as this is an important element for evaluation of the candidate DIS approach.

As the stressmarks are minimalist, the likelihood that problem-domain expertise biased results is small. To further minimize that possibility, participants were asked to share their algorithmic methods with other members of the DIS community. The possibility that source code supplied by a participant may be distributed to other teams was advertised.

The intent of each stressmark should be clear from its description. Participants were reminded to resist the temptation to optimize the results specifically for the given data sets, at the cost of performance on arbitrary sets. To reinforce this, we prepared 'surprise' test files, which would be delivered at the direction of the program manager, for the purpose of validation of results.

2.11.8 Data Sets

As with the Benchmark Suite, the primary control over problem size comes from the input data. A wide range of problem sizes was specified by the supplied input data sets. Participants were expected to measure performance of their implementations for each input file in the set, except when the memory requirements of the file exceed the capacity of the test system.

Participants were invited to create additional input files at their discretion; the provided files are to be considered a minimal set. They were also reminded that additional sets may be released from time-to-time during the program.

Requirements of the data sets were effectively identical to those of the Benchmark Suite data sets.

2.11.9 Submission of Results

Participants were expected to supply essentially the same information about their tests as described for the Benchmark Suite.

2.12 Pointer Stressmark

The *Pointer Stressmark* requires a system to repeatedly follow pointers ('hop') to randomized locations in memory. The memory access pattern is therefore defined by the need to collect small blocks of memory from unpredictable locations. This pattern is commonly encountered in applications, however it is normally associated with a small number of consecutive accesses. Still, it tests a capability orthogonal to the other stressmarks of this set.

The *Pointer Stressmark* consists of fetching a small number of words at a given address or offset, finding the median of the values, and using the result and an additional offset to determine the address for the next fetch. The process is repeated until a 'magic number' is found, or until a fixed number of fetches have been done. The purpose of the median operation is to require the access of multiple words at each location, and the additional offset reduces the likelihood of self-referential loops.

The entire process is performed multiple times for each test. This allows the test to be performed in parallel. The discussion here refers to 'threads' in this context, though there is no requirement that they be implemented using that method.

The values to be fetched can be array indices, or they can be memory addresses computed during initialization, at the discretion of the user.

2.12.1 Input

Input consists of a single ASCII file containing all the parameters required for a single run. The table below gives the format and description of each item.

Item Number	Description	Format	Limits
1	Size of field of values, f, in words.	%lu	$2^4 \le val \le 2^{24}$
2	Size of sample window, *w*, in words.	%h	$2^0 \le val < 2^4$, *val* modulo 2 = 1
3	Maximum number of hops to be allowed for each starting value.	%lu	$2^0 \le val \le 2^{32}-1$
4	Seed for random number generator.	%ld	$1-2^{31} \le val \le -1$
5	Number of threads, *n*.	%d	$2^0 \le val \le 2^8$
$3i + 6, 0 \le i < n$	The starting index for the ith thread.	%lu	$0 \le val < f$
$3i + 7, 0 \le i < n$	The minimum ending index for the ith thread.	%lu	$0 \le val < f$
$3i + 8, 0 \le i < n$	The maximum ending index for the ith thread.	%lu	$0 \le val < f$

2.12.2 Algorithm

The general procedure for this stressmark is as follows:

Step	Action
1	Read the input file.
2	Initialize the random number generator and fill the field with random numbers of the range [0…*f*-*w*-1]. These values represent indices into the field. If equivalent addresses are to be used, they may be computed now.
3	Start the timer.

4	Perform these steps once for each thread: (a) Clear the hop count. (b) Set *index* to the starting index for this thread. (c) Fetch values at location given by *index*. This value may be an address or an address offset. Get values at *index*, *index*+1, *index*+2, … *index*+*w*-1. (d) Set *index* to the sum of the median of the values obtained in step 4(c) and the hop count, modulo (*f*-*w*). (e) Increment hop count by one. If hop count equals the maximum number of hops allowed, store the hop count and exit this thread. (f) If *index* is greater than or equal to the minimum ending index for this thread, and *index* is less than the maximum ending index for this thread, store the hop count and exit this thread. Otherwise, go to step 4(c).
5	Stop the timer.
6	Write the output and metrics files.

2.12.3 Data Notes

For Pointer, the size of the field is the primary variation between tests. Some tests determine whether the size of the window affects operation (this relates both to spatial locality and the number of operation per hop). The ending ranges needed to be set by trial-and-error, to ensure the proper number of hops. Ending ranges needed to be situated such that a simple address-bit check would not be satisfactory for completion testing.

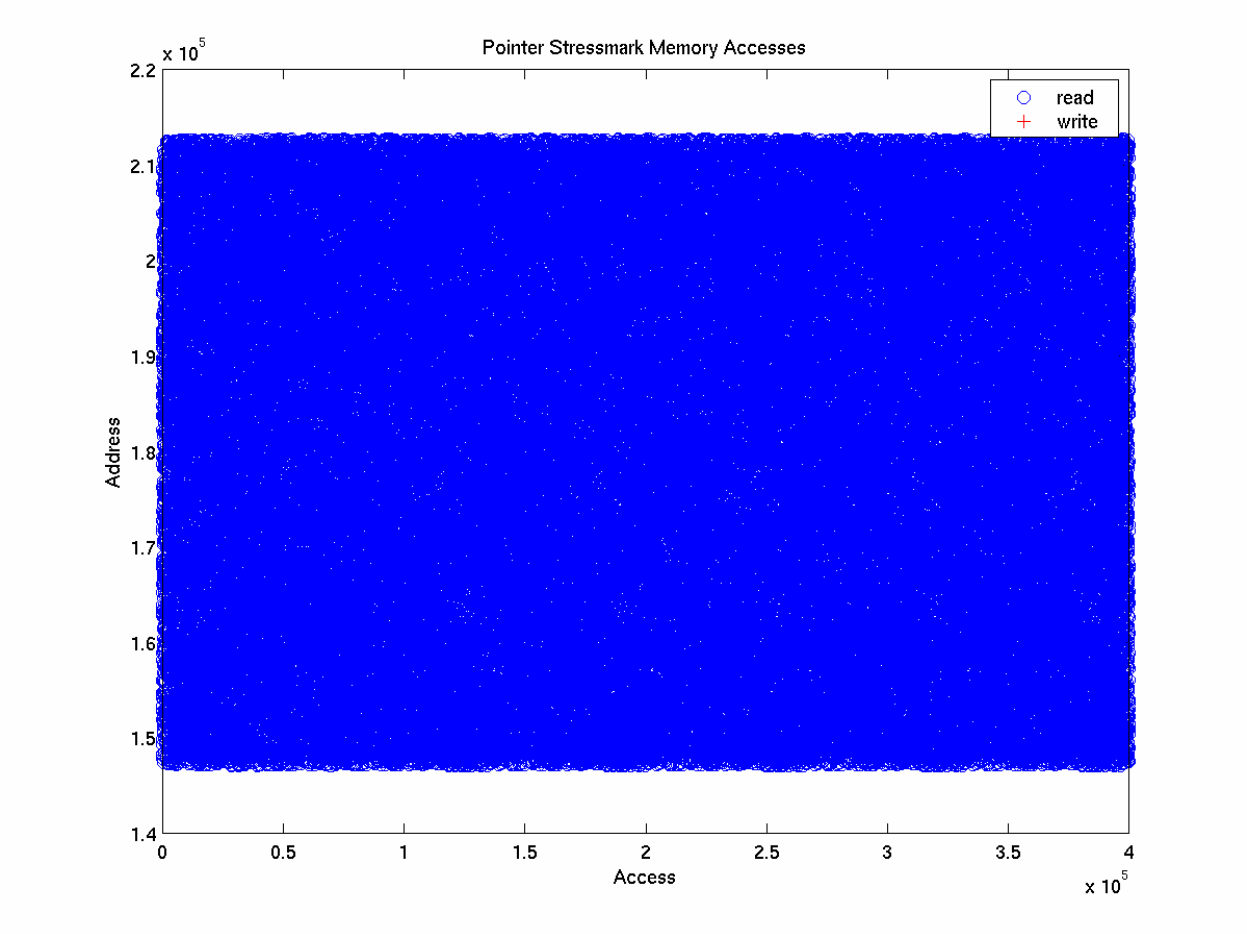
Below is a table of the parameters set for Pointer tests.

Filename	F	W	Maxhops	Seed	N	Thread	Notes
P1	2^10	1	Set such that it is used; not to exceed 1M		16	Wildly various, but adding up to ~1M hops	
P2	2^12	1	Ditto		16	Ditto	
P3	2^14	1	Ditto		16	Ditto	
P4	2^16	1	Ditto		16	Ditto	
P5	2^18	1	Ditto		16	Ditto	
P6	2^20	1	Ditto		16	Ditto	
P7	2^22	1	Ditto		16	Ditto	
P8	2^24	1	Ditto		16	Ditto	
P9	2^20	1	Ditto		16	Add up to 10M	
P10	2^22	1	Ditto		16	Ditto	
P11	2^24	1	Ditto		16	Ditto	
P12	2^20	1	Ditto		16	Add up to 100M	
P13	2^22	1	Ditto		16	Ditto	
P14	2^24	1	Ditto		16	Ditto	
P15	2^20	3	Ditto		16	Same as P6	Window test
P16	2^20	5	Ditto		16	Same as P6	Window test
P17	2^20	7	Ditto		16	Same as P6	Window test

P18	2^22	1	Ditto	Same as P7	8	Still add up to 1M	Thread test
P19	2^24	1	Ditto	Same as P8	4		Thread test
P20	2^18	1	Ditto	Same as P5	2		Thread test
P21	2^16	1	Ditto	Same as P4	1		Thread test

2.12.4 Memory Trace

Here is an example memory trace of this stressmark on a SuperScalar machine. Note the absence of an organized, predictable pattern, and the lack of spatial locality.



2.13 Update Stressmark

The *Update Stressmark* is extremely similar to the *Pointer Stressmark*. It should be considered a companion stressmark, as their memory access patterns are nearly identical to one another. The major difference is that, for this stressmark, elements in the memory field are updated at each access. This consequently makes parallelism at the 'thread' level impossible, because results become nondeterministic. However, it should be noted that parallel implementations could be made practical at a fine level.

The *Update Stressmark* consists of:

- fetching a small number of words at a given address or offset,
- updating the word at the given address,
- finding the median of the values, and
- using the result to determine the address for the next fetch.

The process is repeated until a 'magic number' is found, or until a fixed number of fetches have been done. The purpose of the median operation is to require the access of multiple words at each location. The update requires a memory write operation, and additionally reduces the likelihood of self-referential loops.

The values to be fetched can be array indices, or they can be memory addresses computed during initialization, at the discretion of the user.

2.13.1 Input

Input consists of a single ASCII file containing all the parameters required for a single run. The table below gives the format and description of each item.

Item Number	Description	Format	Limits
1	Size of field of values, *f*, in words.	%ld	$2^4 \le val \le 2^{24}$
2	Size of sample window, *w*, in words.	%h	$2^0 \le val < 2^4$, *val* modulo 2 = 1
3	Maximum number of hops to be allowed.	%ld	$2^0 \le val \le 2^{32}-1$
4	Seed for random number generator.	%ld	$1-2^{31} \le val \le -1$
5	The starting index.	%ld	$0 \le val < f$
6	The minimum ending index.	%ld	$0 \le val < f$
7	The maximum ending index.	%ld	$0 \le val < f$

2.13.2 Algorithm

The general procedure for this stressmark is as follows:

Step	Action
1	Read the input file.
2	Initialize the random number generator and fill the field with random numbers of the range [0…f-w-1]. These values represent indices into the field. If equivalent addresses are to be used, they may be computed now.
3	Start the timer.
4	Perform these steps:

	(a) Clear the hop count. (b) Set *index* to the starting index. (c) Fetch values at location given by *index*. This value may be an address or an address offset. Get values at *index*, *index*+1, *index*+2, … *index*+*w*-1. (d) Set the value referenced by *index* to the sum of its previous value and the hop count, modulo (*f*-*w*). E.g., `x[I]=(x[I]+c)%(f-w);` (e) Set *index* to the median of the values obtained in step 4(c). (f) Increment the hop count by one. If the hop count equals the maximum number of hops allowed, store the hop count and proceed to step 5. (g) If *index* is greater than or equal to the minimum ending index, and *index* is less than the maximum ending index, store the hop count and proceed to step 5. Otherwise, go to step 4(c).
5	Stop the timer.
6	Write the output and metrics files.

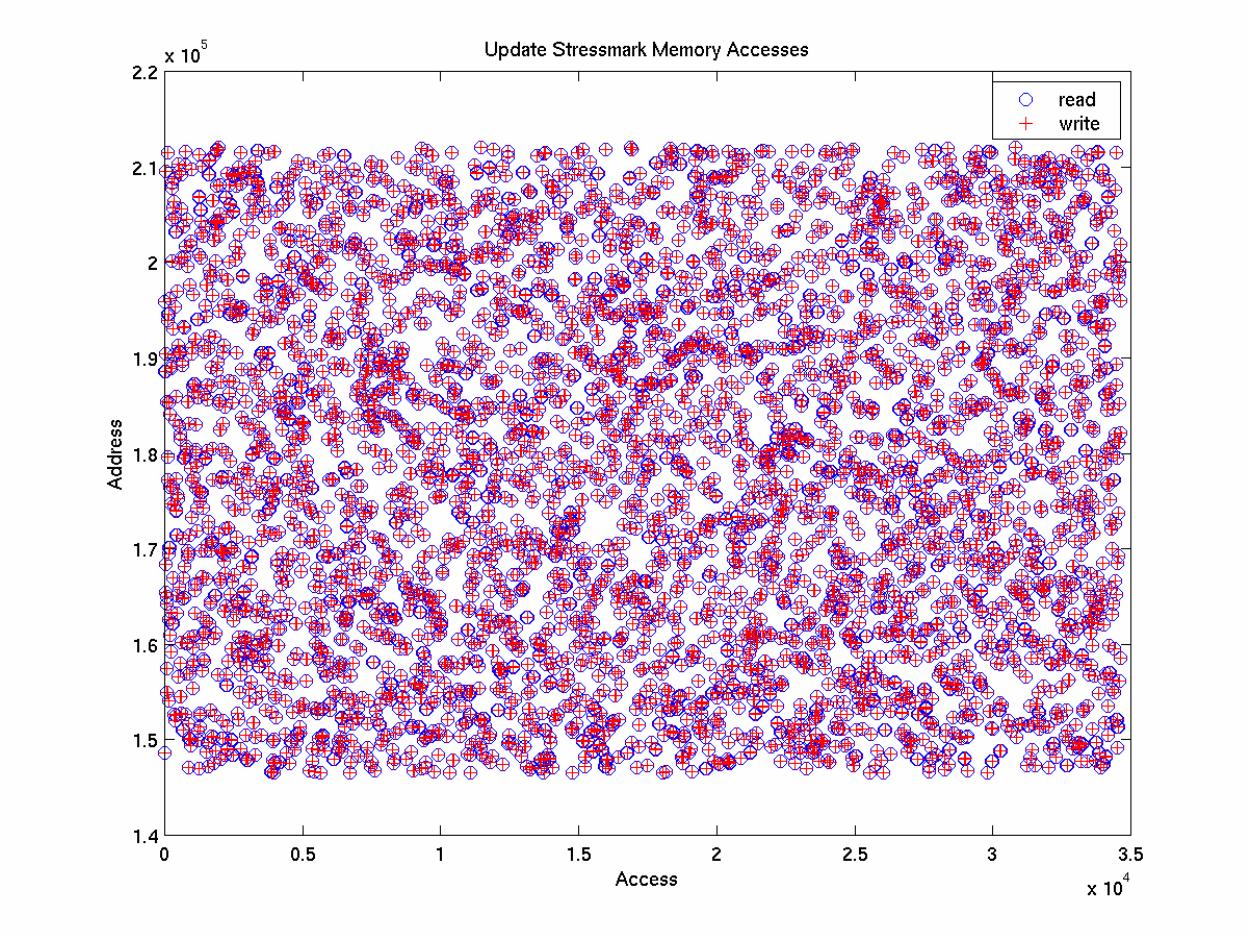
2.13.3 Data Notes

The data requirements here are very much like those for Pointer. Many of the tests were designed to mimic Pointer in key ways, so that the effects of the added write cycle and the lost parallelism could be observed.

Filename	**F**	**W**	**Maxhops**	**Seed**	**Thread (1)**	**Notes**
U1	2^10	1	1M			
U2	2^12	1	1M			
U3	2^14	1	1M			
U4	2^16	1	1M			
U5	2^18	1	1M			
U6	2^20	1	1M			
U7	2^22	1	1M			
U8	2^24	1	1M			
U9	2^26	1	1M			
U10	2^28	1	1M			
U11	2^22	3	1M	Same as U7	Same as U7	Window test
U12	2^22	5	1M	Same as U7	Same as U7	Window test
U13	2^22	7	1M	Same as U7	Same as U7	Window test

2.13.4 Memory Trace

Below is a sample memory trace from Update. It has the same character as Pointer, except that there is a memory write coincident with each hop.



2.14 Matrix Stressmark

The *Matrix Stressmark* characterizes operations dealing with data that is stored in a compact form, and accessed in mixed patterns. In this stressmark, the iterative conjugate gradient method is used to solve a linear system represented by the equation $\boldsymbol{A} \bullet \boldsymbol{x} = \boldsymbol{b}$, where $\boldsymbol{A}$ is a sparse nxn matrix, and $\boldsymbol{x}$ and $\boldsymbol{b}$ are vectors with n elements each. For simplicity of initial data generation, matrix $\boldsymbol{A}$ is positive-definite and symmetric.

The stressmark requires solving the equation $\boldsymbol{A} \bullet \boldsymbol{x} = \boldsymbol{b}$ for vector $\boldsymbol{x}$. As the required method is iterative, the steps are performed until $\boldsymbol{x}$ is found to be within a specified error tolerance, or for a specified maximum number of iterations, whichever occurs first. A random number generator, supplied as part of the stressmark specification, is used to populate the matrix $\boldsymbol{A}$ and vector $\boldsymbol{b}$ initially. In this way, the input required to specify the trials has been reduced to a small set of parameters, which are sufficient to define the linear system. Output consists of a value dependent on the solution vector $\boldsymbol{x}$.

2.14.1 Input

Input for the *Matrix Stressmark* is provided in a single ASCII file, as a list of parameters defining the linear system to be solved. The table below describes each field.

Item Number	Description	Format	Limits
1	A seed value for the random number generator.	%ld	$1\text{-}2^{31} < val < -1$
2	The dimension, n, of matrix A and vectors x and b.	%d	$1 < val \le 2^{15}$
3	The number of nonzero elements to be inserted within matrix A.	%d	$n < val \le n^2$
4	The maximum number of iterations to be performed. The actual number of iterations required may be less if the calculated error is lower than the tolerance specified by the next field.	%d	$0 < val \le 2^{16}$
5	The tolerance of error for the solution vector.	%e	$1.0\text{e-}7 < val < 0.5$

2.14.2 Conjugate Gradient Method

The *Matrix Stressmark* consists of the conjugate gradient iterative method for solving a linear system.[21] Each input file specifies an nxn linear system represented by

$$\boldsymbol{A} \bullet \boldsymbol{x} = \boldsymbol{b}$$

Here, '•' denotes matrix multiply. The conjugate gradient algorithm is used to determine the vector x to within the specified error tolerance, or for the specified maximum number of iterations, if that occurs first.

Define two vectors, r_k and p_k, for $k = 1, 2, \ldots$ (denoting the iteration count). Choose

$$\boldsymbol{r}_1 = \boldsymbol{b} - \boldsymbol{A} \bullet \boldsymbol{x}_1$$

and form the sequence of improved estimates

[21] See Press, W. H., Teukolsky, S. A., Vetterling, W.T., and Flannery, B. P., *Numerical Recipes in C, Second Edition*, Cambridge University Press, 1992.

$$\boldsymbol{x}_{\boldsymbol{k}+1} = \boldsymbol{x}_{\boldsymbol{k}} + \alpha_{\boldsymbol{k}} \boldsymbol{p}_{\boldsymbol{k}}$$

to obtain our solution x. For initial vectors r_1, set $p_1=r_1=b$, and choose $x_1=(0\ldots0)$. For each subsequent iteration, use the following equations to update the vectors:

$$\alpha_k = \frac{\boldsymbol{r}_k^T \bullet \boldsymbol{r}_k}{\boldsymbol{p}_k^T \bullet (\boldsymbol{A} \bullet \boldsymbol{p}_k)}$$

$$\boldsymbol{r}_{k+1} = \boldsymbol{r}_k - \alpha_k \boldsymbol{A} \bullet \boldsymbol{p}_k$$

$$\beta_k = \frac{\boldsymbol{r}_{k+1}^T \bullet \boldsymbol{r}_{k+1}}{\boldsymbol{r}_k^T \bullet \boldsymbol{r}_k}$$

As long as the recurrence does not break down because one of the denominators is zero (which theoretically cannot happen for symmetric positive definite matrix A), the iteration will complete in n steps or less. Perform the iteration steps outlined above until:

- the maximum specified number of iterations have been performed, or

$$\boldsymbol{p}_{k+1} = \boldsymbol{r}_{k+1} + \beta_k \boldsymbol{p}_k$$

- the error is within specified tolerance, which is defined as when the solution x satisfies the equation:

$$\frac{|\boldsymbol{A} \bullet \boldsymbol{x} - \boldsymbol{b}|}{|\boldsymbol{b}|} \leq errorTolerance$$

2.15 Neighborhood Stressmark

The *Neighborhood Stressmark* deals with data that is organized in multiple dimensions, and operated upon by neighborhood operators. Memory access is therefore somewhat localized along one dimension, and spread substantially along all others. Occasionally, processing can be organized such that memory is accessed by multiple synchronous threads with unit-strides.

Image processing applications commonly include this type of operator. Clever programmers will organize code to maximize order of address access, but this is not always practical.

For this stressmark, the relationships of pairs of pixels within a randomly generated image are measured. These features quantify the texture of the image, and require memory access to pairs of pixels with specific spatial relationships.

Imagery to be measured is constructed by populating a blank square image with a multitude of line segments, where the line endpoints, width, and brightness values are randomly generated. The width and bit-depth of the image are input at run-time.

The texture measurements are obtained by estimating a gray-level co-occurrence matrix (GLCM).[22] The matrix contains information about the spatial relationships between pixels within an image. Statistical descriptors of the co-occurrence matrix have been used as a practical method for utilizing these spatial relationships. The descriptors can be estimated by using sum- and difference-histograms.[23]

Two statistical descriptors, *GLCM entropy* and *GLCM energy*, are calculated within the stressmark. Each is calculated for multiple distances and directions, as defined in Section 2.15.2.

2.15.1 Input

Input consists of a single ASCII file containing all the parameters required for a single run. The table below gives the format and description of each item. From these parameters, an image is generated for consumption by the stressmark kernel.

Item Number	Description	Format	Limits
1	A seed for the random number generator.	%ld	$1\text{-}2^{31} < val < -1$
2	The bit-depth of the image. Pixel values range from 0 to 2^{val}-1.	%d	$7 \leq val \leq 15$
3	The dimension, *dim,* of the input image. Images are always square, so *dim* is both the width and the height.	%d	$1 < val \leq 2^{15}$
4	The number of line segments to be inserted into the image.	%d	$0 < val \leq 2^{16}$
5	The minimum thickness, in pixels, of the line segments, *minThickness*.	%d	$0 < val < dim$
6	The maximum thickness, in pixels, of the line segments.	%d	$minThickness \leq val, val < dim$

[22] Parker, J., Algorithms For Image Processing And Computer Vision, Wiley Computer Publishing, 1997.

[23] Unser, M., "Sum and Difference Histograms for Texture Classification," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Vol. PAMI-8, 1:118-125, 1986.

7	The shorter of the distances between pixels to be measured.	%d	$0 < val < dim$
8	The longer of the distances between pixels to be measured.	%d	$0 < val < dim$

2.15.2 Algorithmic Specification

The *Neighborhood Stressmark* entails estimation of two texture measurements obtained through the gray-level co-occurrence matrix (GLCM). These descriptors are *GLCM entropy* and *GLCM energy*. Two input parameters–*distanceShort* and *distanceLong*–define the spacing to use when calculating the descriptors, thus determining the scale of the textures being measured.

GLCM entropy and *GLCM energy* can be found, without calculation of a gray-level co-occurrence matrix, by finding a sum histogram, *sumHist,* and a difference histogram, *diffHist*. Like the GLCM, these histograms are dependent on a specific offset distance and direction. The sum histogram is simply the normalized histogram of the sums of all pairs of pixels a given distance and direction from one another. Likewise, the difference histogram is the normalized histogram of the differences of all pairs of pixels a given distance and direction from one another. For our purposes, each histogram requires one bin for each possible result, or $2^{bit\ depth\ +\ 1}$ bins.

Based on the histograms, the GLCM descriptors are defined as:

$$GLCM\ entropy\ =\ -\sum_i sumHist(i) * log[sumHist(i)] - \sum_j diffHist(j) * log[diffHist(j)]$$

$$GLCM\ energy\ =\ \sum_i sumHist(i)^2 * \sum_j diffHist(j)^2$$

where *sumHist(i)* is the *i*th bin of the normalized sum histogram, and *diffHist(j)* is the *j*th bin of the normalized difference histogram for the distance and direction of interest.

For this stressmark, the *GLCM energy* and *GLCM entropy* are found for each of four directions at two distances. The distances are given as the input parameters *distanceShort* and *distanceLong*. The directions are constant for all tests: 0° (horizontal), 45° (right diagonal), 90° (vertical), and 135° (left diagonal). Note that since images are typically thought of as growing rightward and downward from their origin, these directions represent angles swept clockwise. Note also that with square pixels, these angles never result in quantization issues.

2.15.3 Data Notes

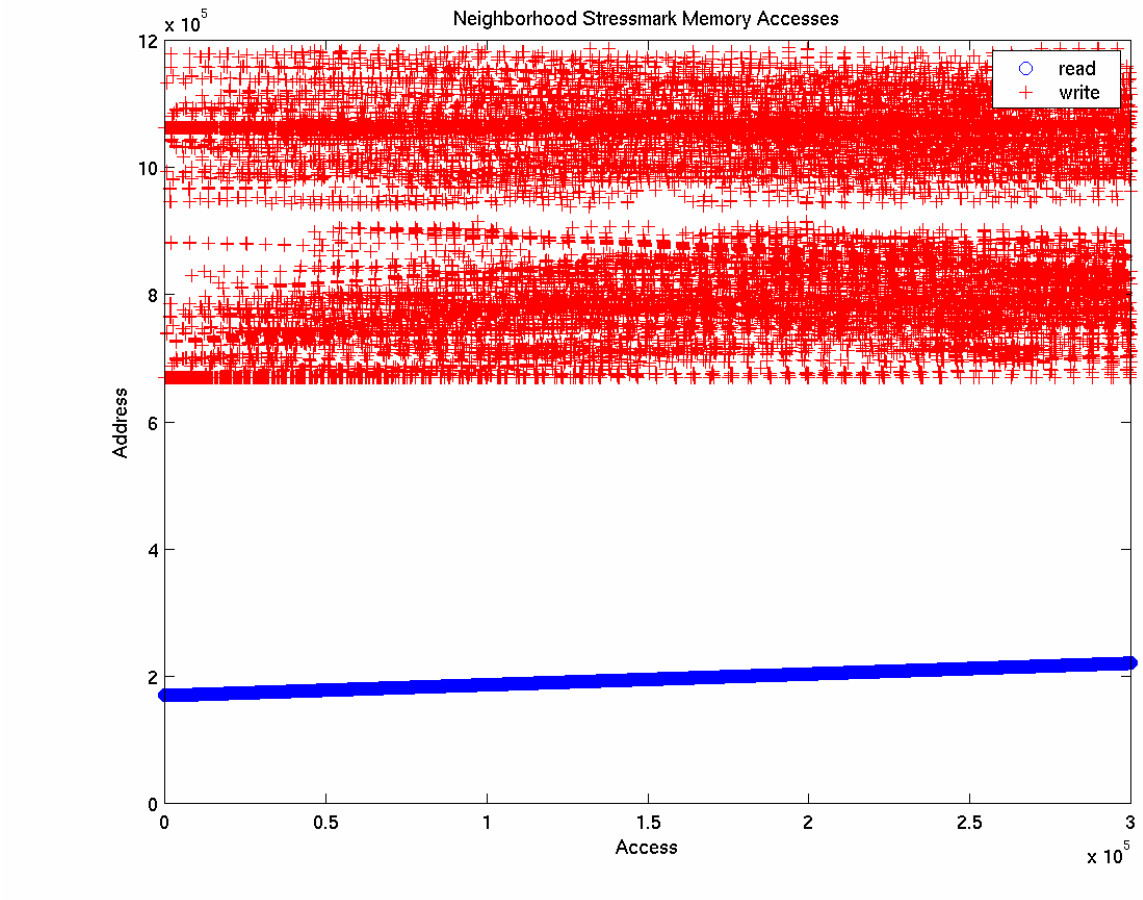
Certain considerations were kept while generating the test files:

- Bit depth affects histogram size, and thus cache-miss likelihood. Different bit depths with a matching random number seed should get same histograms with different resolutions.
- High spatial frequency images yield more random histograms, implying lower spatial locality. Increasing max width raises chances that low-numbered bins will be used a lot.
- Distance measures will not matter much to standard machines, because multiple lines will unit-stride along as partners. However, novel schemes that utilize spatial locality should show better performance on smaller ranges.

- The lines themselves vary slowly, which would hit certain areas of histograms a lot when the lines are wide or the colors change very slowly. When the lines are short, narrow, and vary a lot, there will be less locality of histogram bins.

2.15.4 Sample Memory Trace

The sample memory trace shows the highly localized read for the image kernel, and the histogram writes.



2.16 Field Stressmark

The *Field Stressmark* emphasizes regular access to large quantities of data. It involves scanning for strings, possibly with fine-grain parallelism. In this way, it tests a system's ability to perform on searches when indices are unavailable or inadequate.

The *Field Stressmark* consists of searching an array (field) of random words for token strings, which are used as delimiters. All words between instances of the delimiter form a sample set, from which simple statistics are collected. The delimiters themselves are updated in memory. When all instances of a token are found, the process is repeated with a new one. The statistics for each sample set are reported.

2.16.1 Input

The table below gives the format and description of each input item.

Item Number	Description	Format	Limits
1	Size of field, *f*, in words.	%ld	$2^4 \le val \le 2^{24}$
2	Seed for random number generator.	%ld	$1\text{-}2^{31} \le val \le -1$
3	Offset value for token modifier, *mod_offset*. This value is the number of words between a found token word and the word that should be used to modify it. See Section 2.17.2 for more information.	%ld	$2^0 \le val \le 2^{16}$
4	Number of tokens, *n*.	%ld	$2^0 \le val \le 2^8$
5 + *i*, $0 \le i < n$	The ith token, given as a zero-terminated string of hexadecimal values.	%x […%x] 0	Each element of string: $0 \le val < 2^8$ Length of string: $1 \le val < 2^3$

2.16.2 Algorithm

The general procedure for this stressmark is as follows:

Step	Action
1	Read the input file.
2	Initialize the random number generator and fill the field with random integers of the range $[0 \ldots 2^8\text{-}1]$.
3	Start the timer.
4	Get the next token from the input list.
5	Search through the field, looking for an instance of the token. Note that the first subfield *ends* at the first instance of the token. At the beginning of the field: (a) Set the count of the number of subfields to one. (b) Initialize the *count* and *sum* accumulators to zero. Initialize the *minimum* accumulator to a value greater than or equal to $2^8\text{-}1$. At each token instance: (c) Store the values from the *count*, *sum*, and *minimum* accumulators for subse-

quent output.

(d) Modify the delimiter in the field by adding the value of one new word to each word in the string, discarding the overflow of each word. The words to be added are located by finding the sum of the token modifier offset and the index of the delimiter words, using modulo arithmetic. I.e., F[*x*]+=F[(*x*+*y*)%f]. Thus, each time an instance of the token is found and used as a delimeter, it is modified by summing with another string within the field.

(e) Increment the count of the number of subfields by one.

(f) Initialize the *count*, and *sum* accumulators to zero. Initialize the *minimum* accumulator to a value greater than or equal to 2^8-1.

(g) Proceed with the search, beginning at the next word that is not part of the delimiter.

For each word encountered that is not a part of a token instance:

(h) Increment the *count* accumulator by one.

(i) Increment the *sum* accumulator by the value of the current word, discarding the overflow.

(j) If the value of the current word is less than that of the *minimum* accumulator, set the value of the *minimum* accumulator to that of the current word.

At the end of the field:

(k) Store the values from the *count*, *sum*, and *minimum* accumulators for subsequent output.

(l) Proceed to step 6.

6 Repeat steps 4 and 5 until the input list of tokens is exhausted (total of n times).

7 Stop the timer.

8 Write the output and metrics files.

2.16.3 Tests

Here are some notes about the test parameters:

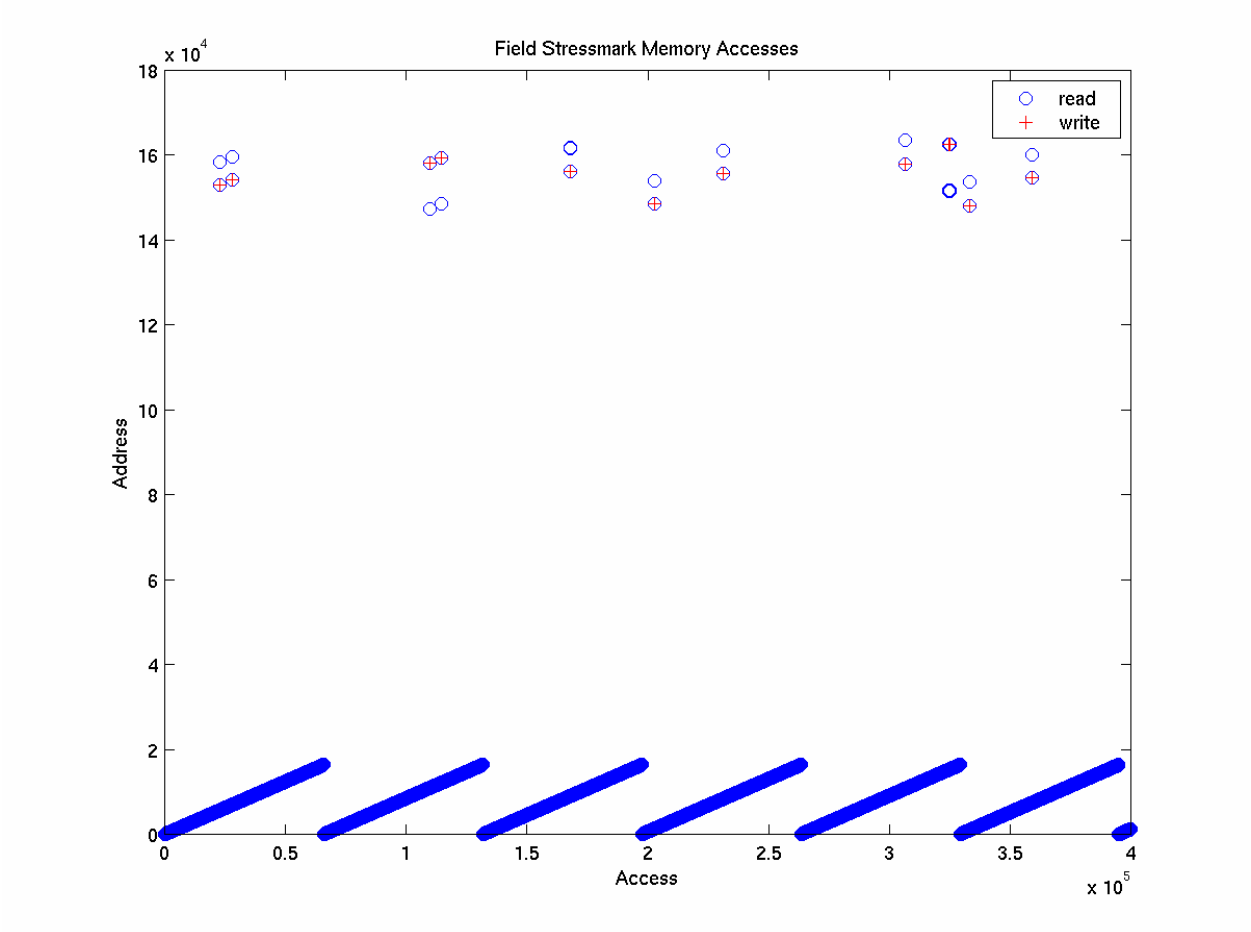
- Some offsets should be large, in order to prevent single-pass schemes.
- Size of field is determinant primarily in keying whether a pass will stay in cache or not. Prefetching may take care of everything.
- Length of token list suggests efficiency. A long list and a short list processed in similar time implies efficiency for multiple runs.
- Ensure no more than 256 instances of delimiters.
- Ensure that modified words always contribute to result.
- The length of string determines how frequently they will be encountered. It also affects how much processing has to be done for a match.

Filename	F	Seed	Mod_offset	N	Tokens	Comments
F1	2^14	rnd	rnd	64	rnd	Adjust tokens so that total number of hits (all n tokens) is

						constant (and high but each <256).
F2	2^16	“	“	32	“	“
F3	2^18	“	“	16	“	“
F4	2^20	“	“	8	“	“
F5	2^22	“	“	8	“	“
F6	2^24	“	“	4	“	“
F7	2^20	Same as F5	Same as F5	32	“	Efficiency test
F8	2^22	Same as F6	Same as F6	Same as F6	Same size as F6, but hit many fewer times	Token hit test

2.16.4 Sample Memory Trace

The Field stressmark has a regular, unit-stride access pattern, as can be seen from the memory trace below.



2.17 Corner-Turn Stressmark

The *Corner-Turn Stressmark* emphasizes effective memory bandwidth without stressing functional units. It involves the matrix transposition ("corner-turn") operation useful in signal processing applications. Although matrix transposition is a required element in other stressmarks and benchmarks within this suite, this stressmark involves practically no computation, so memory bandwidth issues are not readily masked behind processing latency.

The *Corner-Turn Stressmark* consists of transposing a matrix of random words repeatedly. As there is no specific computation, there is no required output.

If the candidate architecture employs multiple computation nodes, the stressmark should be executed using a variety of combinations of these nodes, so the relationships between configuration, problem size, and performance may be studied.

The *Corner-Turn Stressmark* has both *in-place* and *out-of-place* modes, referring to whether the transposed matrix must overwrite the original, or exist as a copy (on different computational nodes if multiple nodes are utilized).

2.17.1 Input

The table below gives the format and description of each item.

Item Number	Description	Format	Limits
1	Row dimension, *x*, of matrix, in words.	%ld	$2^4 \le val \le 2^{15}$
2	Column dimension, *y*, of matrix, in words.	%ld	$2^4 \le val \le 2^{15}$
3	Seed for random number generator.	%ld	$1\text{-}2^{31} \le val \le -1$
4	Number of times to transpose matrix, *n*.	%ld	$2^0 \le val \le 2^{16}$
5	Flag indicating operating mode. 0=*in-place*; 1=*out-of-place*.	%h	$0 \le val \le 1$

2.17.2 Algorithm

The general procedure for this stressmark is as follows:

Step	Action
1	Read the input file.
2	Create a matrix of y rows and x columns, stored in row-major order and evenly distributed across computation nodes. Each element of the matrix should consist of at least one word large enough to contain integer values of the range $[0 \ldots 2^{32}\text{-}1]$.
3	Initialize the random number generator and fill the matrix in row-major order with random integers of the range $[0 \ldots 2^{32}\text{-}1]$.
4	If in *out-of-place* mode, create a second matrix of *x* rows and *y* columns, to be used as a destination matrix.
5	Start the timer.
6	Transpose the input matrix completely.
7	Stop the timer and store results.
8	Repeat steps 5-7 until the matrix has been transposed n times.
9	Write the metrics file. At a minimum, best, worst, and average cases must be reported.

A histogram of all results is recommended.

2.17.3 Test Files

The following parameters were defined in the provided test files.

Filename	X	Y	Seed	N	Flag	Comments
CT1	128	128		1000	0	
CT2	128	128		1000	1	
CT3	1024	1024		50	0	
CT4	1024	1024		50	1	
CT5	1024	8192		10	0	
CT6	8192	1024		10	0	
CT7	800	20000		4	0	GPS
CT8	800	20000		4	1	GPS
CT9	12000	12000		2	1	Should be simple fraction of ct9
CT10	16000	16000		2	0	
CT11	16000	16000		2	1	
CT12	32000	16000		1	0	
CT13	8000	32000		1	0	

2.18 Transitive Closure Stressmark

The *Transitive Closure Stressmark* emphasizes semi-regular access to elements in multiple matrices concurrently. It requires solution of the *all-pairs shortest path* problem, which is fundamental to a variety of computational problems.

The *Transitive Closure Stressmark* utilizes the Floyd-Warshall all-pairs shortest path algorithm[24]. It accepts an adjacency matrix of a directed graph as input, and uses a recurrent relationship to produce the adjacency matrix of the shortest-path transitive closure. It runs in $O(n^3)$ time, which asymptotically is no better than n calls to Dijkstra's single-source shortest-paths algorithm.[25] However, this approach is generally considered to operate better in practice than Dijkstra's, especially when adjacency matrices (as opposed to lists) are employed.

2.18.1 Input

The table below gives the format and description of each item of the input file.

Item Number	Description	Format	Limits
1	Number of vertices, n, in words.	%ld	$2^3 \le val \le 2^{14}$
2	Number of edges, m, in words.	%ld	$0 \le val \le n^2$
3	Seed for random number generator.	%ld	$1\text{-}2^{31} \le val \le -1$

[24] Thomas Cormen, Charles Leiserson, and Ronald Rivest, *Introduction to Algorithms*, McGraw Hill, 1990.

[25] *Ibid*.

2.18.2 Algorithm

The general procedure for this stressmark is as follows:

Step	Action
1	Read the input file.
2	Create an nxn adjacency matrix, D. Each element must be able to represent discrete values of the range $[0\ldots2^{31}\text{-}1]$. Initialize all matrix elements to 2^{31}-1.
3	Initialize the random number generator. Generate m random integer triples $\{x_i, y_i, z_i\}$, where x_i and y_i are in the range $[1\ldots n]$, and z_I is in the range $[0\ldots2^8\text{-}1]$. Use the triples to initialize the adjacency matrix D of the directed graph, by using each x_I as a starting vertex, each y_I as an ending vertex, and each z_i as the length of the edge. (I.e., $D_{x,y}=z$.)
4	Start the timer.
5	For each $k=[1\ldots n]$, let $D^k_{i,j}=\min(D^{k-1}_{i,j}, D^{k-1}_{i,k}+D^{k-1}_{k,j})\ \forall\ i,j\in[1\ldots n]$. Note: This step requires additional temporary storage for D; if desired, storage space may be allocated and initialized to contain the value 2^{31}-1 during step 2.
6	Stop the timer.
7	Calculate the sum of each row and column of D^n, ignoring any matrix elements containing the value 2^{31}-1.
8	Write the output and metrics files.

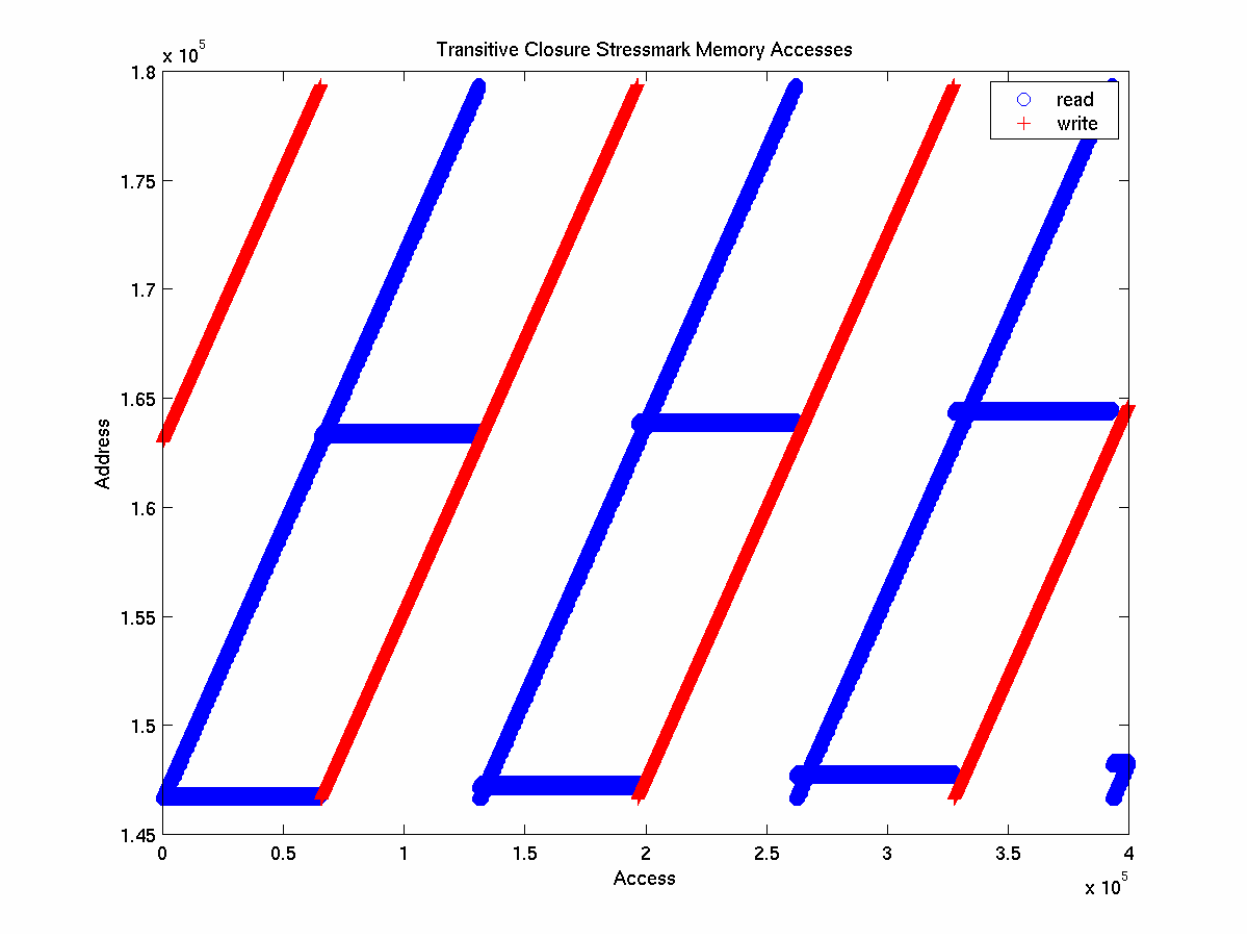
2.18.3 Test Files

The following are the major parameters of the provided test files.

File Name	N	M
TC1	16	50%
TC2	64	25%
TC3	64	50%
TC4	256	25%
TC5	256	75%
TC6	512	25%
TC7	512	50%
TC8	512	75%
TC9	1024	20%
TC10	1024	40%
TC11	1024	60%
TC12	1024	80%
TC13	2048	10%
TC14	2048	25%
TC15	2048	50%
TC16	2048	75%
TC17	2048	90%
TC18	4096	10%
TC19	4096	50%
TC20	4096	90%
TC21	8192	1%
TC22	8192	10%

2.18.4 Sample Memory Trace

The sample memory trace below illustrates the regular, mixed-use nature of the stressmark.



2.19 GUPS

Though not strictly a part of the DIS suite, what came to be known as the GUPS (Global Updates Per Second) benchmark became a useful element in many of the DIS discussions and analyses. In its simplest form, it can be described with this line of C code:

```
while (1) ++rand();
```

This benchmark stresses random access to large banks of memory, and frequently runs at far below peak processing rates due to limits of memory bandwidth, address generation, and bank conflicts.

3 DIVA

The University of Southern California Information Sciences Institute's *Data-IntensiVe Architecture* (DIVA) project[26] designed and developed a new class of multiprocessor processor-in-memory (PIM) integrated circuits specifically devised to serve as smart memory to augment conventional systems. The team submitted results based on simulations for Corner Turn, Pointer, Transitive Closure, and Neighborhood stressmarks. Several configurations were simulated, representing systems of up to 64 PIMs.

3.1 Description

The DIVA team utilized a PIM-based approach to the DIS problem. The team has designed and developed a new class of multiprocessor processor-in-memory (PIM) integrated circuits specifically devised to serve as smart memory to augment conventional systems. Increased bandwidth is achieved both by integrating processing logic into memory chips, and providing "wide-word" instruction-level parallelism. On-chip latencies are very low, and some parallelism is available due to multiplicity of processors within each PIM chip, and multiplicity of PIMs within a system.

The DIVA system as configured for these experiments is comprised of between zero and 64 multiprocessor PIM chips serving as smart-memory coprocessors to an external host processor. The PIM chips are capable of performing normal memory operations without a performance penalty. Additionally, DIVA permits dynamic interaction between host and PIMs by a command protocol over a conventional memory bus allowing multiple host commands to be serviced across the PIM array and within any single PIM at one time. Finally, messages may be passed between PIMs without host processor intervention.

The team generated a suite of software to support the system, enabling execution of methods within the memory subsystem, and managing problem partitioning and communication. Since the PIMs can emulate standard memory, legacy code is executable, though no performance enhancement is available in that mode. To take full advantage of the DIVA system, code and data must be partitioned among the PIMs. A large portion of the DIVA effort was devoted to creation of tools to assist this process. Since most of the partitioning work for the benchmark effort was done prior to the completion of the software suite, it is unknown what level of effort would be required to gain performance with the suite fully in place.

Though not used for benchmarking purposes[27], a 9.8mm-by-9.8mm prototype PIM chip containing 55 million transistors was fabricated in TSMC's 0.18 micron technology through MOSIS.

The DIVA project summaries claim 100x improvement in effective memory bandwidth for a system comprising 128MB of PIM coprocessors, yielding an anticipated 10-100x improvement for certain applications.

---

[26] http://www.isi.edu/asd/project.html

[27] A test was run using the Corner-Turn stressmark for verification purposes. The experiment is discussed later in this section.

3.2 Measurements

The DIVA team provided for results four stressmarks: Corner-Turn, Pointer, Transitive Closure, and Neighborhood. For each stressmark, multiple configurations were tested, with the number of active PIMs in each configuration varying from zero to 64. Generally, only the small test problems were applied.

All results were simulated. (One stressmark was used during hardware validation; results of that experiment are provided in section 5.8, below.) Simulations were performed using a special-purpose, system-level simulator called DSIM. From the DIVA report[28]:

> DSIM uses RSIM[29] as a framework, with significant extensions. RSIM is an event-driven simulator that models shared-memory multiprocessors built with state-of-the-art multiple issue, out-of-order superscalar processors. DSIM extensions include a simpler PIM processor with a Wide-Word unit, the DIVA memory system, the parcel communication mechanism and the PIM-to-PIM interconnect. DSIM supports the full DIVA PIM ISA.
>
> The DSIM host processor is taken directly from RSIM, as well as the host first and second-level caches. The host processor architecture is based on the MIPS R10000, which is configured as a four-issue processor with two integer arithmetic units, two floating-point units and one address unit. Loads are non-blocking. It has a 32Kbyte L1 and a 1Mbyte L2 cache, both two-way associative, with access times of 1 and 10 cycles, respectively. Both L1 and L2 caches are pipelined and support multiple outstanding requests to separate cache lines.
>
> The host is connected to the DIVA memory system via a split-transaction, 64-bit bus. The memory system consists of the aggregation of all PIM memories, where each local memory is visible from both host and local PIM processor. DSIM maintains the current open row of each memory bank to determine the memory access type (page or random mode) and simulates arbitration between host and PIM accesses. The memory latencies seen by the host are 52 cycles for page-mode accesses and 60 cycles for random mode, and include the bus transfer delay, the memory arbitration time and the DRAM access time (4 and 12 cycles for page and random mode, respectively). The memory latencies seen by the local PIM processor, including arbitration and DRAM access times, are 6 and 14 cycles for page and random mode accesses, respectively.
>
> An application library supports a cache-line-flush function to enforce coherence between the host caches and PIM memory, as well as synchronization and communication functions. These functions are linked with the application code, and their execution is simulated by DSIM in the same way as the application code.
>
> DSIM also models the parcel mechanism and the PIM-to-PIM interconnect in detail. Applications executing on DSIM have direct access to the parcel buffers via parcel handling functions that perform the writing/reading to/from the memory-mapped parcel buffers. These parcel handling functions are part of DSIM's application library, and support the full set of parcel buffer status reads, triggering/non-triggering writes to the send parcel buffers and destructive/non-destructive reads from the receive parcel buffers. These functions are linked with the application code, and their execution is simulated by DSIM as part of the application.
>
> Finally, we make the conservative assumption that the PIM processor runs at half the speed of the host processor. Although the inherent speed of the logic is no slower, we make this assumption because the WideWord register accesses could impact the clock speed.

---

[28] *DIVA Report on DIS Stressmarks*, September 2, 2002.

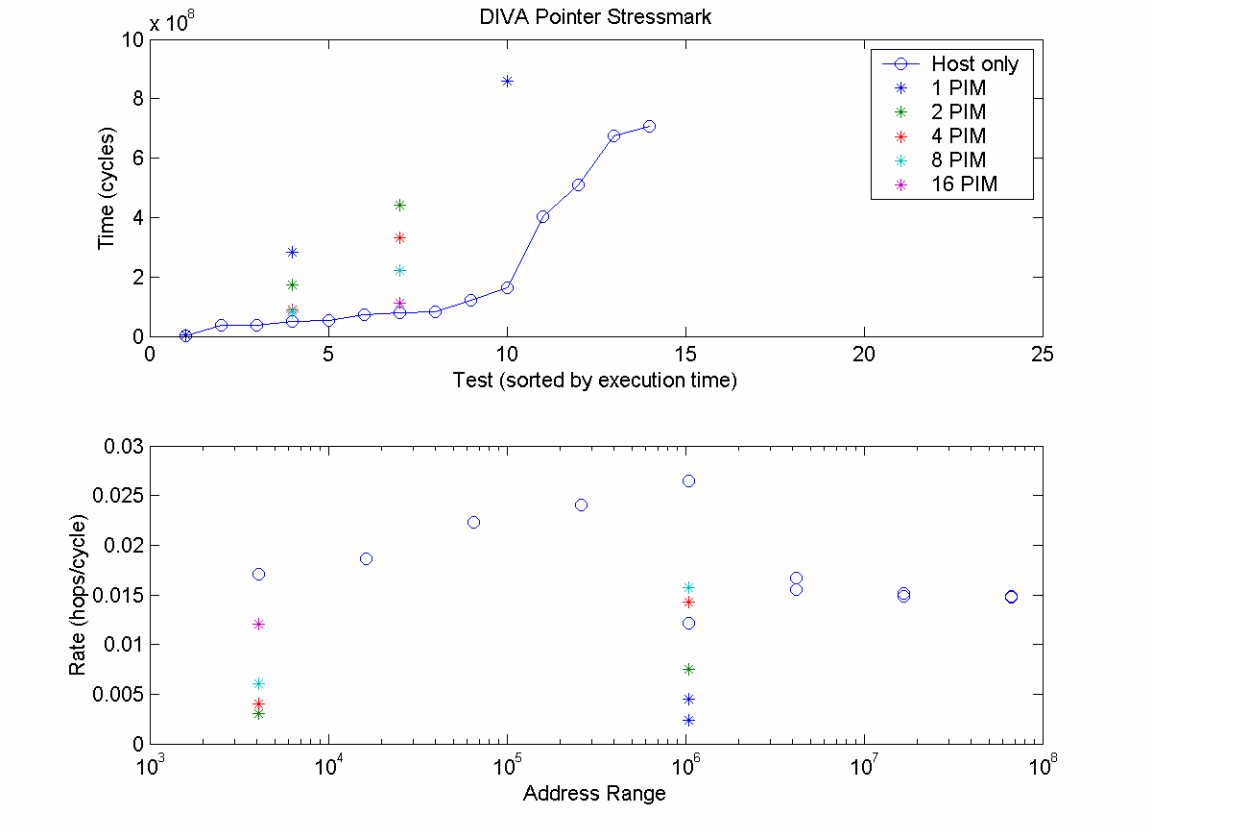
[29] http://rsim.cs.uiuc.edu/rsim

3.3 Pointer

The DIVA implementation of the Pointer Stressmark involved partitioning PIM nodes into groups, each group large enough to keep one copy of the *field* array. Threads are distributed among the PIMS, and executed simultaneously. "Hops" are computed locally when possible; otherwise, the thread is transmitted to the PIM within the group that holds the next address. The host processor monitors the execution of threads, and signals the PIMS when all threads have terminated.

This implementation allows for excellent thread parallelization, and execution time should be determined by the longest thread, as long as the field is distributed somewhat evenly.

The results are shown in the graph below.



Significantly, the address ranges shown are small—less than 10^8 words. The DIVA team ran test files p01 and p05 (shown in the graph above with indices 4 and 7, respectively) when testing multiple-PIM configurations. The p20 test was also run with the 1-PIM configuration, but that test had the same memory field as the p05 test. No tests larger than 10^6 words were done with PIM configurations.

The graph shows that the PIM versions are actually slower than the host-only version. The DIVA team explained the result as follows.

> In our experiments, the HOST version performs better than the 1-PIM version when the input size fits in the host L1 or L2 caches (as in p05.in and p20.in). The PIM version performs better than the host version when the input data set fits in one PIM node and does not fit in the host cache (such data is not reported since none of the DIS input sizes satisfies this condition). Our PIM version of Pointer does not speedup when the array must be partitioned among PIMs. The main rea-

son our Pointer does not scale well is that the rate of communication per hops is very small, and the local computation (an average of a couple of hops) is not enough to amortize the cost of PIM-to-PIM communication.

The DIVA team observes that PIM versions performance exceed the host-only version's when the field size fits in one PIM node but not fit in the host's cache. However, the 1MB size of the PIM's SRAM suggests that this would never be the case with the current floor-plan.

The two tests with the 10^6-word address ranges show markedly different processing rates due to the coarse-level parallelism of the tests. The host-only configuration achieves a rate for the 16-thread test that is 2.2 times that of the 2-thread test, while the 1-PIM version nets only a 1.9 gain over its 2-thread rate.

Due to the inter-PIM communication rate, this approach does not achieve a positive performance gain with respect to this stressmark. Though some performance gain is to be expected due to coarse parallelism, none is in evidence.

3.4 Update

The Update Stressmark was not implemented by the DIVA team. An implementation similar to that of the Pointer Stressmark—even if it had shown positive performance gain—would probably not yield performance gains because:
(a) there are no gains available from coarse parallelization; and
(b) the updates to the memory field would have to be synchronized somehow across PIM groups.

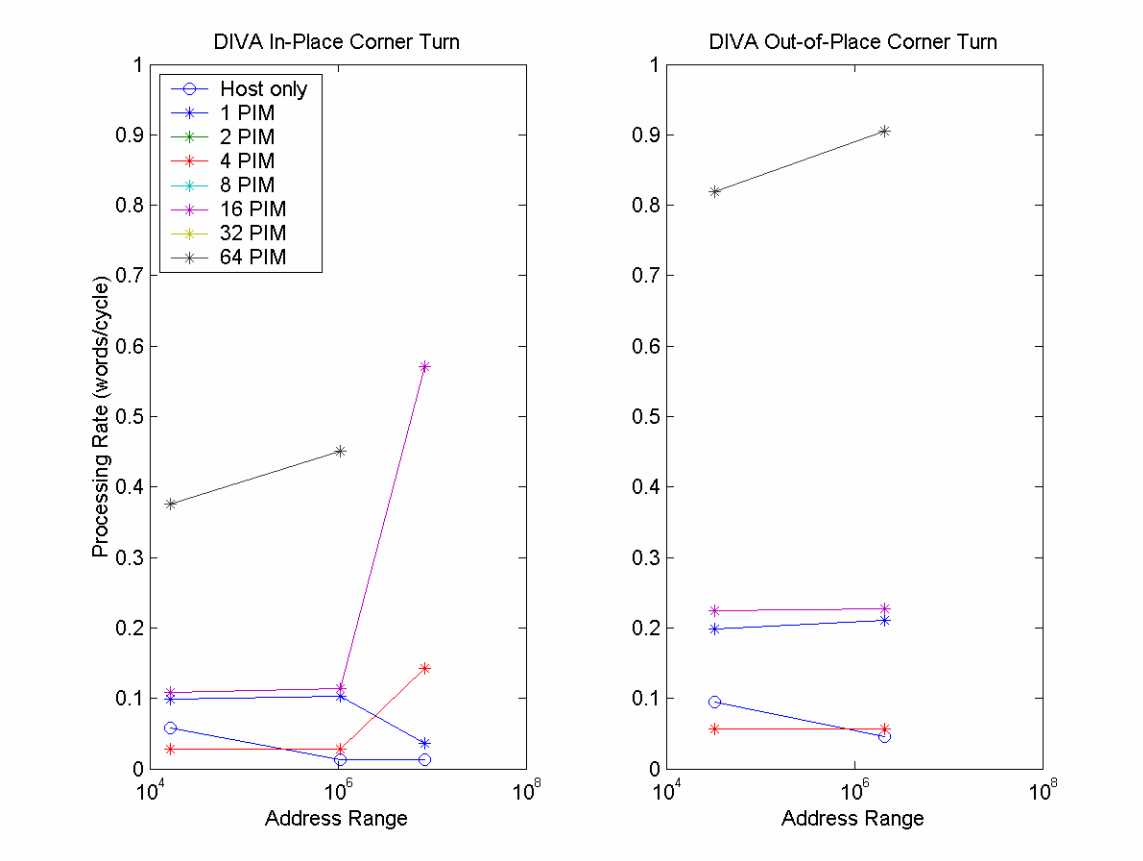
3.5 Corner-Turn

The DIVA team supplied the following information regarding its implementation for the Corner-Turn Stressmark:

> Our Cornerturn implementation performs a hierarchical matrix transpose, where the matrix is partitioned into blocks and each block is assigned to a PIM node. The transpose of each block is computed by partitioning the block into subblocks, which are transposed in WideWord registers using permutation operations.
>
> The host performs the initial block partitioning (see file partition.c), keeping a table with the assignment of blocks to PIMs, and coordinates synchronization between host and PIMs. In the first phase of the computation, each PIM computes the transpose of its local block. After that each pair of PIMs owning blocks that need to be swapped to form the transposed matrix communicate using the PIM-to-PIM network.
>
> The local block transpose is performed as a set of transposes of 8x8 sub-blocks (except for block sizes that are not multiple of the number of matrix elements that fit in a WideWord register). For the out-of-place transpose, each 8x8 subblock is loaded into the WideWord register file (an 8x8 matrix with 32-bit elements requiring 8 WideWord registers), and transposed via a sequence of permutation operations. The transposed subblock is then stored back in memory at the target location. In the in-place transpose (of square blocks) two subblocks of size 8x8 are loaded in WideWord registers, each subblock is transposed in registers, and then the transposed subblocks are stored back in memory, swapping locations to form the transposed block. This implementation takes advantage of the large capacity of the WideWord register file, avoiding loads and stores to memory during the transpose of each 8x8 subblock.
>
> After computing its local transposed block, each PIM exchanges its transposed block with the PIM that owns the location of the block in the transposed matrix. For example, for a square matrix divided into 4 blocks where block-00 is assigned to PIM-0, block-01 to PIM-1, block-10 to PIM-2 and block-11 to PIM-3, PIM-1 exchanges its transposed block with PIM-2. PIMs-0 and PIM-3 keep their transposed blocks since they should remain in the same location in the transposed matrix.

> The communication phase is performed in 2 steps: in the first step PIMs owning blocks in the upper triangular submatrix send their blocks to PIMs owning blocks in the lower triangular submatrix; the second step completes the exchange of blocks with PIMs in the lower triangular submatrix sending blocks to PIMs in the upper triangular submatrix.
>
> Finally, this version of Cornerturn avoids contention on the PIM-to-PIM network by assigning each pair of blocks that will exchange locations in the transposed matrix to neighbor PIMs. This assignment is based on the fact that communication occurs between fixed pairs of PIMs, and that when assigning a block to a PIM it is possible to determine the location of its transposed block in the transposed matrix, and then assign the block corresponding to this location to the nearest PIM available.

This graph shows the DIVA results for the Corner-Turn Stressmark. Results have been segregated by test type; the left side shows in-place operation, and the right side shows out-of-place. Both sides give processing rate versus address range.



The graphs show a substantial increase in processing rate corresponding to increasing numbers of PIMs.

Regarding the host-only versus the 1-PIM implementation, the DIVA team had this to report:

> Our HOST version of Corner Turn shows high memory stall times for input sizes that don't fit in the host L2 cache. This application has very little temporal reuse, since each matrix element is accessed a few times only during each matrix transpose. Thus primarily spatial reuse is exploited in cache, and each new cache line is only reused a few times. In the PIM version, the WideWord datapaths also exploit the available spatial reuse.

> Furthermore, the WideWord loads/stores and operations on eight matrix elements at a time also reduce the number of accesses to memory. Finally, the latency seen by the PIM processor is lower than that suffered by the host for large input sizes. For example, for input ct03.in (1024x1024 matrix transpose, out of place), the matrix size is four times larger than the host L2 cache, resulting in memory stalls of 98% of the host execution time. The 1-PIM version spends 40% of the execution time stalled for memory, due to the lower on-chip latencies and a reduction on the number of memory accesses (the average latency seen by the PIM is 11.6 cycles, since most of the accesses are in random mode).

The data suggest that in-place corner-turning benefits from the local bandwidth more than out-of-place, as would be expected.

Increasing the number of PIMs continues to net performance gain, but at reduced efficiency. The mean rate gains per processor over the host-only version are as follows:

1 PIM	3.9296
4 PIM	0.8106
16 PIM	0.8094
64 PIM	0.7859 (extrapolated)

Significantly, the address ranges tested are less than 10^7 words. The DIVA team did not run test files ct07 through ct13. It is expected that host-only performance would continue to degrade as problem size increased. Because the PIM versions show no such degradation,[30] the improvement over the host-only configuration for larger problem sizes cannot be extrapolated.

3.6 Transitive Closure

The DIVA team's implementation of the Transitive Closure Stressmark was described as follows:

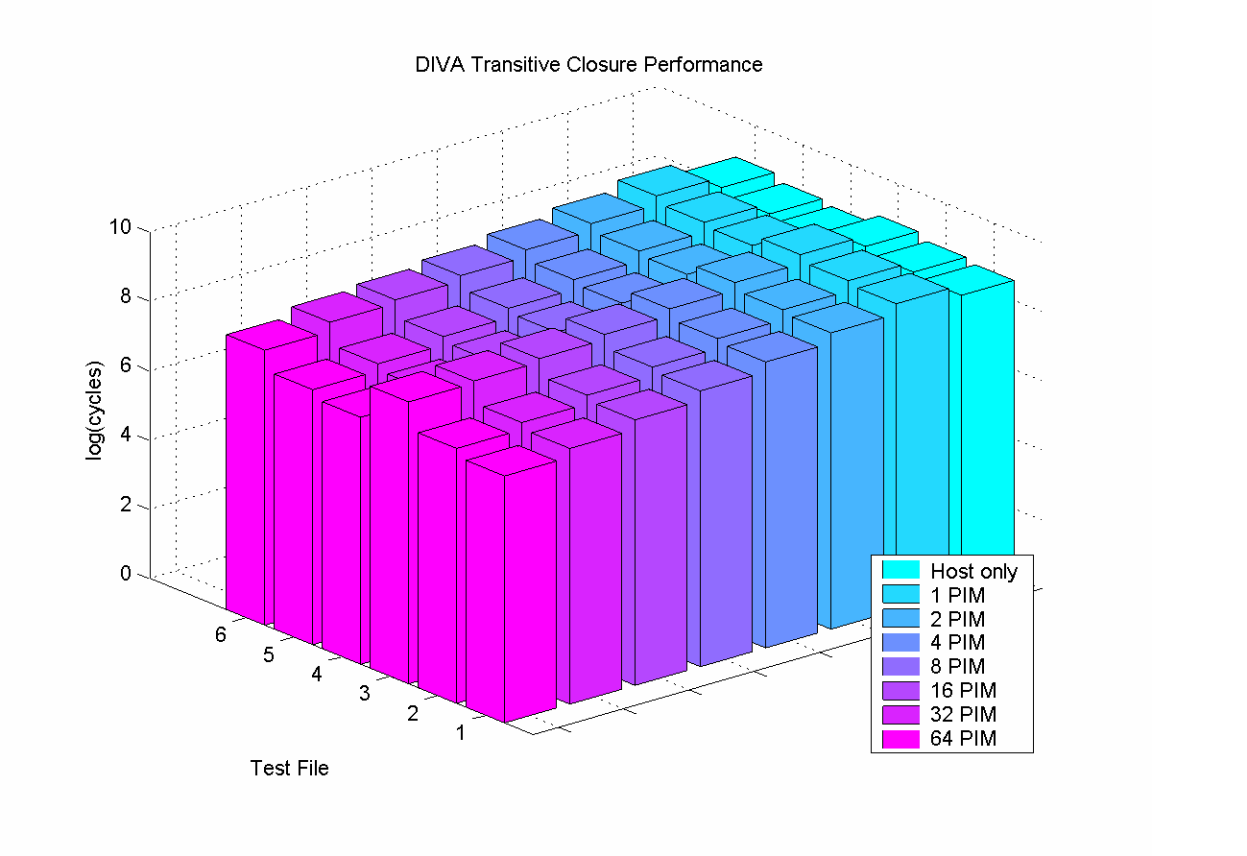
> The DIVA version of Transitive Closure is based on the DIS sample code, and uses a dense matrix to represent the distance graph. It exploits both fine-grain parallelism, by performing WideWord arithmetic operations on eight 32-bit elements of the matrix in parallel, and coarse-grain parallelism, by partitioning the data and computation among PIM nodes.
>
> The host processor computes the matrix partition and coordinates synchronization. Matrices din and dout are partitioned by rows and a set of consecutive rows is assigned to each PIM node. For the main loop nest of Transitive Closure, for each iteration of the outer loop k, each PIM node performs the inner-loop computation (loops i and j) on its local set of rows, using a copy of row k previously sent by the PIM that owns row k. Therefore, for each iteration of loop k, the PIM node that owns row k sends a copy of this row to all other PIMs. All PIM nodes synchronize on each iteration of loop k, after the communication phase.
>
> The multicast of a matrix row from one PIM to all other PIMs is performed using the multicast mode supported by the DIVA parcel buffer mechanism. The sender processor writes a parcel payload to the parcel buffer, and then writes a parcel header for each destination PIM. The write to the parcel header triggers the sending of the parcel to the specified destination. This multicast mode allows the sender processor to write the parcel payload only once, reducing the cost of assembling parcels in the parcel buffer.
>
> The local computation on each PIM node takes advantage of the Wide-Word unit in the computation of the minimum value of each pair of elements from two matrix rows. Selective execution us-

[30] One exception is the 1-PIM, in-place series.

ing a WideWord operation (wmrgcc) merges the contents of two WideWord registers according to condition-code bits, allowing an eÆcient computation of the minimum value of each pair of elements of two WideWord operands.

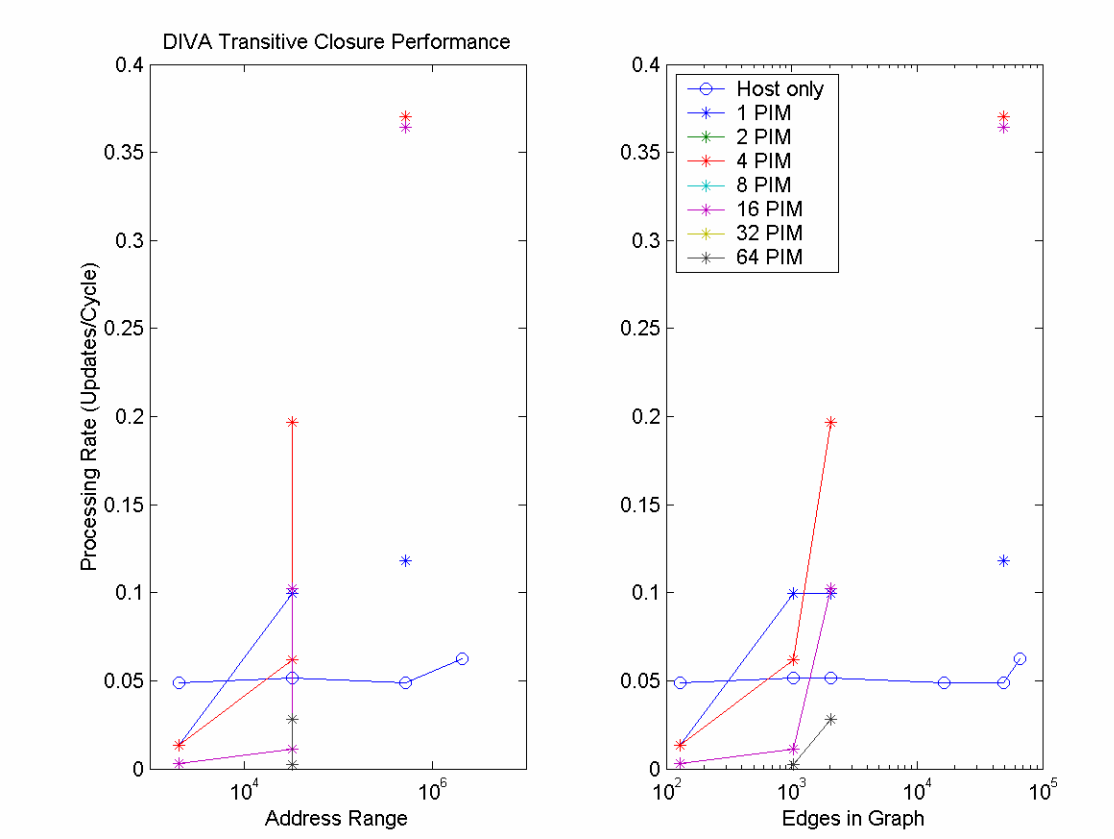
Finally, for both the HOST and PIM versions, the inner loops (loops i and j) of the main loop nest were interchanged, so that the HOST can benefit from spatial locality at the caches, and PIMs can exploit spatial reuse in WideWord registers.

The times found by simulation of this stressmark are shown in the graph below.



Note that the tests executed on DIVA were small; the largest required 0.5 Mword. Increased performance gains for larger tests may be expected, but were not demonstrated.

Conversion of the times to processing rates, and graphing against address range and edge density results in the following graphs.



The host-only version displays a relatively constant processing rate, independent of address range or edge density. The latter is to be expected; the former, however, illustrates that the tests used by the DIVA team were small, and therefore may not be indicative of typical performance.

The processing rates of the PIM configurations all show some dependence on both address range (dependent on the number of nodes) and the number of edges. Since the rates are shown to be rising with increased values of each of those parameters, the true potential of the system is likely not indicated.

The team provided additional information:

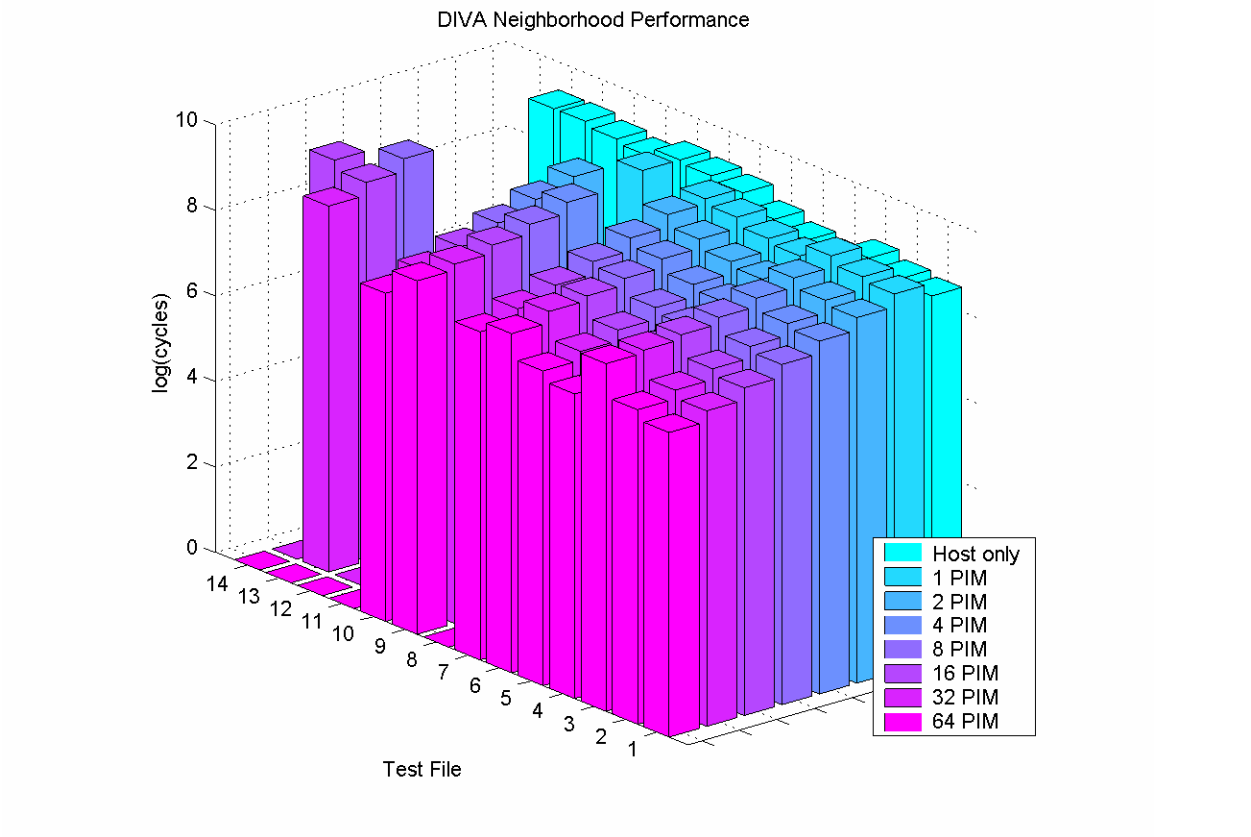
> Our PIM version benefits from both fine- and coarse-grain parallelism, and from the higher bandwidths available at the PIMs. For example, the HOST version for input tc05.in spends 65.2% of its execution time stalled due to cache misses, with 11.3% of the misses satisfied at the L1 and 58.4% satisfied at the L2, resulting in an average memory latency of 6.7 cycles. The 1-PIM version shows a higher average memory latency (9.5 cycles), but it issues less memory accesses, since the WideWord unit is used to transfer data to/from memory and perform the computation. Therefore the 1-PIM memory stall time is smaller than that of the HOST version. The use of the WideWord unit also results in the added benefit of exploiting spatial reuse, since the matrix is accessed with stride one in the row dimension.

3.7 Neighborhood

The DIVA team also experimented with the Neighborhood Stressmark. Regarding the implementation, the team wrote:

> The Neighborhood implementation on DIVA exploits coarse-grain parallelism by partitioning the computation among PIM nodes. Each PIM computes a partial histogram locally, and at the end of the computation phase, the PIM nodes perform a parallel reduction to compute the final histogram. The parallel reduction takes n-1 steps, where n is the number of PIM nodes. The communication is scheduled to take advantage of the PIM-to-PIM interconnection topology (bidirectional ring), avoiding congestion in the network.

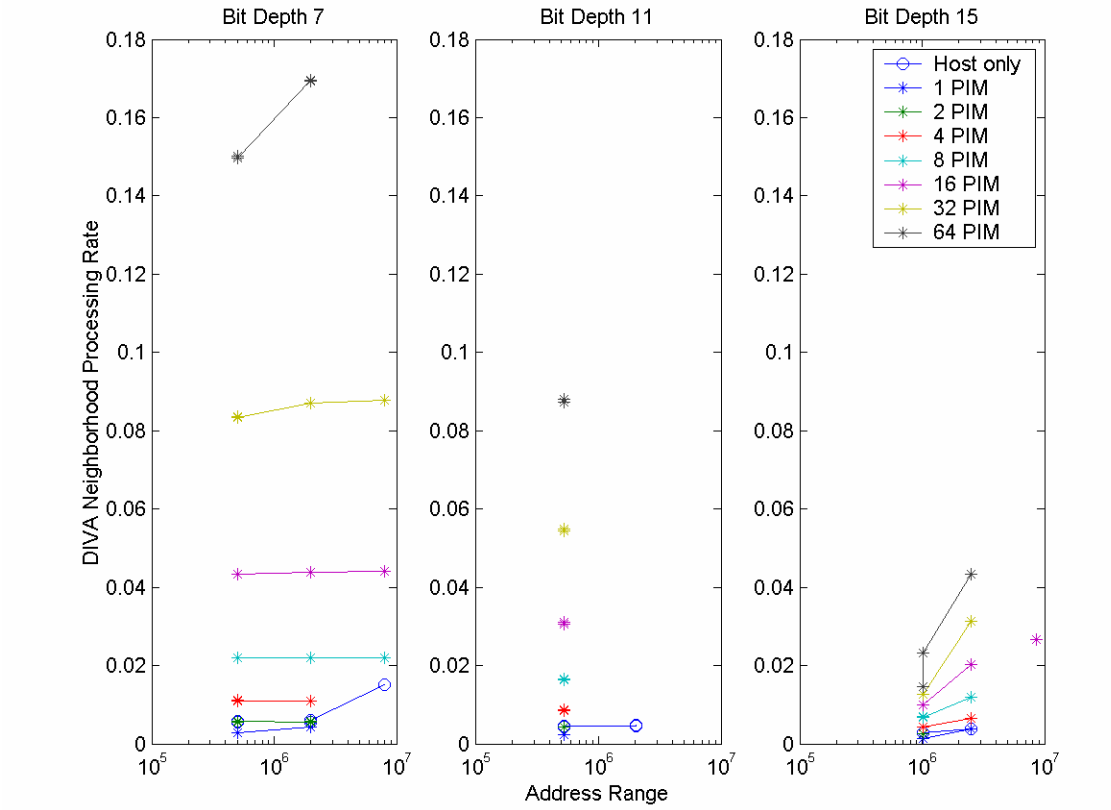
The following graph shows the processing times (in cycles) extracted from the supplied results.



The data show performance gains for PIM versions, except that performance is lost for the 1-PIM version against the host-only version for most tests. The DIVA team suggests this occurs whenever the image fits in the host's L2 cache, because the memory latencies seen by the PIM are larger than the L2 access time. However, the coarse-grain parallelism available by partitioning the problem across PIM nodes gives considerable performance gains.

The tests executed on DIVA were small; most of the images contained 1M pixels or less. The largest test performed was with a 4M-pixel image.

The following graph shows the processing rates for these experiments, organized by bit depth (which determines the size of the histogram) and graphed against address range (dominated by image size).



In all cases, the processing rate appears to grow with problem size. This suggests that performance efficiency had not yet peaked for either the host or the PIM configurations for the evaluated problem sizes. In other words, the overhead (for example, compulsory cache misses and communication delays during sub-histogram combination) is not amortized well due to the small number of pixels processed.

The graph shows an inverse correlation between processing rate and bit depth. In fact, the performance difference between bit-depths is greater than that between shown address ranges. While it is possible that this could be an indication that the DIVA system is not especially suited for histogram updates, a more likely explanation is that the smaller bit depth of the pixels allowed the DIVA team to exploit fine-grain parallelism. More pixels can be processed simultaneously due to the smaller dynamic range and resulting smaller required storage word.

3.8 Hardware Experiment

One benchmark-related experiment was performed with the first prototype chip, primarily to verify certain operations. The parameters of the experiment differed significantly from those that were simulated, but some information is valuable. The DIVA team provided the following information:

> Here is a description of the hardware experiment with cornerturn. This experiment was conduct to: (1) test some of the functionality of the PIM WideWord unit (permutations, conditional execution), and (2) demonstrate the potential for performance improvements for a benchmark with little data reuse (temporal reuse) and large bandwidth requirements. Therefore so far we have only one point (matrix size, type of transpose (out of place)), and have not conducted an extensive evaluation by varying the matrix shape and size and other parameters.

Hardware	We have a host system that consists of a PowerPC 603 + memory controller, a system bus, and 2 PIM chips plugged in as DRAMs. Our experiments so far do not include PIM-to-PIM communication, since we are currently in the process of testing the parcel buffers and PIM route components (PiRCs).
Hardware Parameters	• PPC603 (HOST) running at 166MHz • Host cache size: 16 KBytes (4K 32-bit matrix elements) • Host cache line size: 32 bytes (8 matrix elements) • Host cache associativity: 4 way • System bus running at 66MHz • PIM processor running at 133MHz
Experiment	Out-of-place transpose of a 64-by-128 matrix (total size = 32Kbytes). This size was chosen so that the working set (input and output matrices) does not fit in the host cache but is smaller than the PIM node memory.
Host version	The host version was written in C, but optimized by hand so that the code generated by the compiler is efficient (we checked the assembly code to make sure that is true). Of course the host version would be more efficient if written in assembly, but we did not do it because of time/resources constraints.
PIM version	The PIM version was written in assembly, using the same algorithm for the 8-by-8 transpose in WideWord registers as described in the Stressmarks report. The loops enclosing the 8-by-8 transposes were also written in assembly.
Measurements	We perform the transposes simultaneously on both host and PIM, and for each 1000 transposes on the host, we measure the number of transposes executed on the PIM (we chose an interval of 1000 host transposes to amortize the overhead introduced by the measurements on the hardware).
Results	For every 1000 host iterations we observe 35151 (with a 0.5% variation) iterations of the transpose on the PIM (35X speedup!!).
Comparison with simulation results:	The system we model in the simulator is different than the host system used in the hardware experiments. In the simulator the host runs twice as fast as the PIM processors and it has a 2KByte L1 and a 1MByte L2 caches. Also, the simulator models a MIPS R1000 host and our hardware platform is based on the PPC603. The cornerturn program executed on the simulator is more complex than the 1-PIM, assembly-only, fixed-matrix-size version running on the hardware. The matrix size used in the hardware experiment was chosen such that the data does not fit in the host cache but fits in PIM memory. Since our hardware platform has a very small cache, this problem size is not adequate for a similar experiment in the simulator. One possibility would be to configure the cache parameters of the simulator and run the same problem size on the simulator, but I don't believe this would be a meaningful comparison due to the different platforms (R10000 x PPC603). The two items above make a direct comparison between the hardware and simulation results very difficult. Therefore, to "validate" our hardware results we estimated (analytically) the running times on the hardware, to check whether the observed speedups are near what we should expect. We parameterized our analytical model with values measured on the hardware, as for example, memory latencies, and our estimates are in the "ballpark" of the measured speedups.

3.9 Programming

As can be seen from the descriptions of the stressmark implementations, some software engineering was applied to achieve the results presented. The Diva project, though, included a comprehensive compiler development effort, which could eventually automate the problem-mapping process. It is unclear from the tests how much of the performance can currently be achieved through automated means, though no loss would be expected in any case, due to the PIM's ability to emulate RAM.

3.10 Remarks

It must be kept in mind that, with the exception of one unrelated test, the supplied stressmark results were all from simulations. The programs were run in the absence of any competing processes or unrelated OS overhead, but the same general conditions existed for both the HOST and the PIM simulations.

The results suggest that the PIM approach is appropriate for cellular processing. The stressmark requiring the most global access, Pointer, suffered a performance loss against the baseline. Speed gains on the other stressmarks were good, though more inter-PIM communication generally led to lower performance.

4 IRAM

The *IRAM (Intelligent Random Access Memory)* project[31] proposed to develop single- and multiple-chip systems for data-intensive applications. The single-chip combines a processor and high capacity DRAM to deliver vector super-computer-style sustained floating point and memory performance at reduced power. An IRAM is intended to be smaller, use less power, and be less expensive than conventional machines with separate chips for the processor, external cache, main memory, and networking. The IRAM design is reported to be scalable within a chip, allowing processing power to vary with memory size or power budget, without changes to the architectural specification. IRAM was developed to be programmable using traditional high-level languages, though additionally new software and compiler technologies were developed to utilize IRAM's high bandwidth.

A multi-chip system called ISTORE was developed to extend processing in memory to other levels of the storage hierarchy. The focus of that work was to provide a scalable, self-maintaining, data-intensive computing system, exchanging centralized processing and interconnect for less expensive processors which gain performance through their high bandwidth access to data. The processing power in the storage hierarchy is also used to increase reliability through monitoring and automatic reconfiguration.

4.1 Description

The VIRAM *(Vector IRAM)* architecture utilizes a delayed vector pipeline[32] to hide memory latency; consequently, there is no need for caches. Instead, VIRAM is built around a banked, pipelined, on-chip DRAM memory that is well matched to the memory access patterns of a vector processor. In addition to the vector processor and embedded DRAM, VIRAM has a superscalar MIPS core, a memory crossbar, and an I/O interface for off-chip communication. The prototype implementation of VIRAM is designed to run both the vector and scalar processors at 200MHz. It has 16 MB of DRAM organized into eight banks with no subbanks, four 100MB/s parallel I/O lines, a 1.2V power supply, and a power target of 2 watts.

The team calculated VIRAM peak performance (using multiply-adds) at 3.2GFLOP/s for single-precision floating-point, 6.4GOP/s for 32-bit integer, and 12.8GOP/s for 16-bit integer operations.[33]

4.2 Measurements

The team utilized codes and some data from the DIS Suite, but generally, the testing was limited. In some cases, selected portions of the stressmark were executed, while in others, a subset of the tests was performed. Only single-chip performance was measured.

---

[31] http://iram.cs.berkeley.edu

[32] In such a pipeline the execution of all arithmetic operations is delayed for a fixed number of clock cycles after issue to match the latency of a worst-case memory access, thereby freeing the pipeline's issue stage. In this way, the next instruction can be issued, and thus the pipeline does not stall for RAW hazards. See Krste Asanovic, *Vector Microprocessors*. PhD thesis, University of California, Berkeley, 1998. UCB//CSD-98-1014. Also see Christoforos Kozyrakis. *A media-enhanced vector architecture for embedded memory systems*. Technical Report UCB//CSD-99-1059, University of California, Berkeley, July 1999.

[33] Randi Thomas, *An Architectural Performance Study of the Fast Fourier Transform on Vector IRAM*, Master of Science Report, UC Berkeley, June, 2000.

All measurements were taken from a near-cycle-accurate simulator. Some of the larger data sets were not feasible because of the limitations of simulation. The team observed that processing rates reach a plateau quickly (i.e., at small problem sizes) on IRAM, for most of the problems.

Of *primary* interest to this study was the issue of what happens when the problem size is larger than the IRAM can hold (13 MB total). Unfortunately, the simulator was not set-up for such tests, and no results in that area are reported.

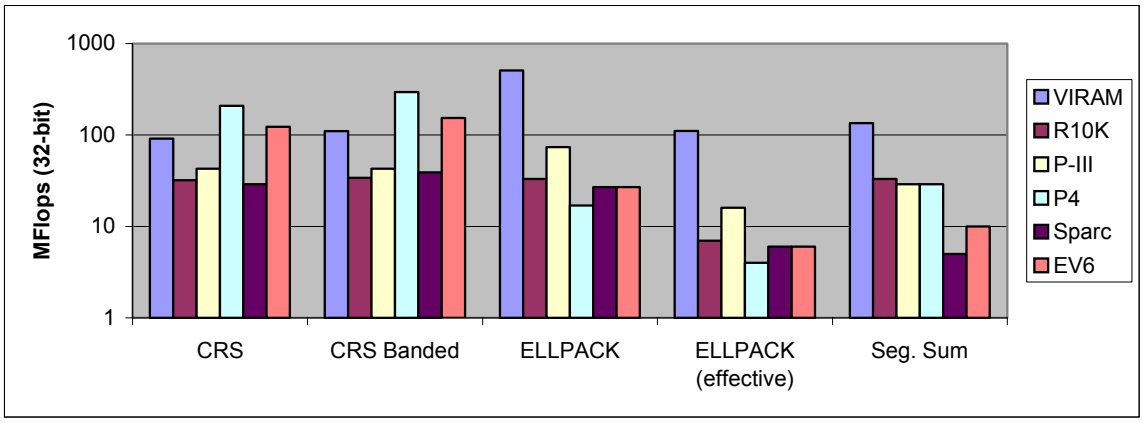
4.3 Benchmarking Environment

For comparison with the VIRAM design, the team selected a set of commercial microprocessor systems. Most were high-end workstation or PC processors, but a low-power Pentium III was also included. Details of the systems are shown in the table below.

	SPARC IIi	MIPS R10K	Pentium III	Pentium 4	Alpha EV6
Make	Sun Ultra 10	Origin 2000	Intel Mobile	Dell	Compaq DS10
Clock	333MHz	180MHz	600MHz	1.5GHz	466MHz
L1	16+16KB	32+32KB	32KB	12+8KB	64+64KB
L2	2MB	1MB	256KB	256KB	2MB
Memory	256MB	1GB	128MB	1GB	512MB

4.4 Matrix Stressmark

The IRAM team did not strictly utilize the DIS Matrix Stressmark. However, the team did utilize the SPMV portion for analysis and comparison with other architectures. The following chart shows the performance of VIRAM and several other architectures, for several sparse-matrix storage formats and algorithms. The matrix parameters were set at N = 10K and M = 177 782.



While the VIRAM is vector-oriented, all of the machines used for comparison are superscalar and cache-based. The different performance characteristics are clear in the chart. The Segmented-Sum and ELLPACK formats vectorize well, and show a greater performance disparity between VIRAM and the other architectures.

VIRAM performance is relatively constant across formats, resulting in consistently better performance than other machines for the ELLPACK and Segmented-Sum formats. The CRS and CRS-banded formats exhibit better performance on the two recent machines with much higher clock rates, Pentium 4 and Alpha EV6; but they are much worse on the UltraSPARC II and MIPS R10000.

Note that the problem as tested fit within a single VIRAM chip.

4.5 Neighborhood Stressmark

For Neighborhood, the team used inputs n01, n02, and n03. All three of these involve a 500^2-pixel image, with 7, 11, or 15 bits of dynamic range, respectively. Emphasis was placed upon vectorizing the histogram and logarithm routines in the provided reference implementation, and the results given here are only for those portions of the stressmark code.[34] Because there is no math library for VIRAM, a simple polynomial approximation that produces 3-4 correct decimal digits was used. When comparing with other machines, the team used the same approximate logarithm function, except when the built−in logarithm function was faster.

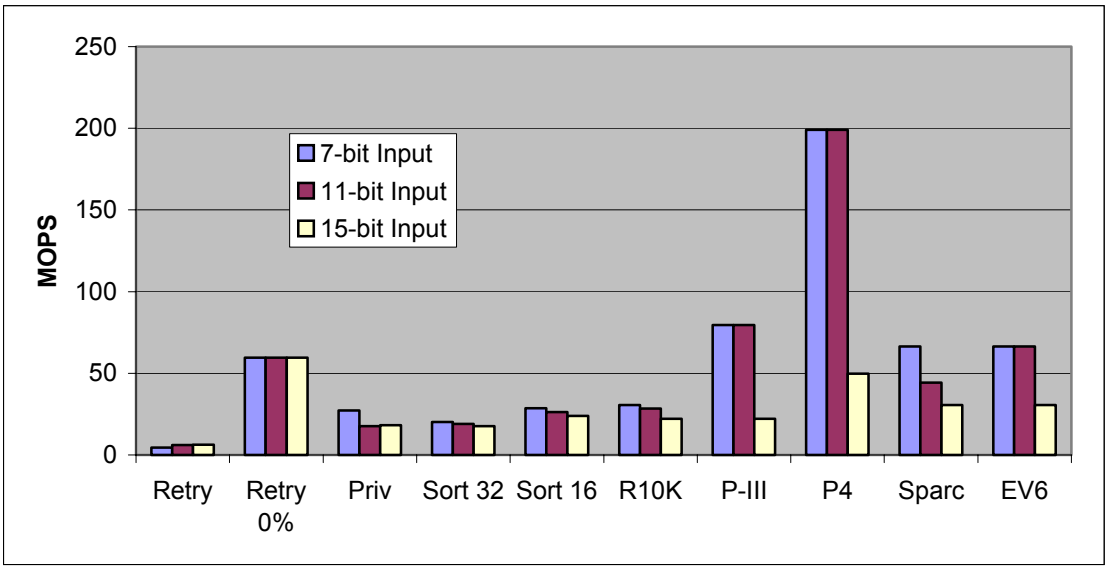
Three histogram optimizations were tested:

- Retry is the default optimization provided by Cray's vcc compiler. It attempts to vectorize the histogram computation by detecting and compensating for duplicates. This entailed essentially no source code changes for the compiler performed the optimization automatically.

- Privatization ensure that the vectors of updates to the histogram are independent: each strip of 16 data elements updates 16 different histogram copies that are summed at the end.

- A sort−diff−find−diff (sort) algorithm exploits the fact that it is relatively simple to update a histogram on a vector processor when the input data are sorted. Groups of 64 elements (the size of a VIRAM register of integers) are sorted, and the histogram is updated on a whole group in one step.

The following graph shows the results. Four experiments (*Retry*, *Priv*, *Sort 32*, and *Sort 16*) represent the VIRAM architecture. (The *Retry 0%* configuration indicates the default compiler algorithm, run on data of the same size but without any duplicates. It is shown for reference only.) The remaining experiments give the results of the basic code running on each of five other architectures.



The results show that on VIRAM, the sort method gives the best performance over the range of bit depths. They also show the improvements that can be obtained when the algorithm is tailored to shorter bit depths. The Cray method performs poorly primarily because of the presence of many duplicates. The code to compensate for dupli-

[34] It is presumed that performance of the VIRAM architecture would be good on the kernel-oriented portion of the stressmark relative to typical superscalar, cache-based systems.

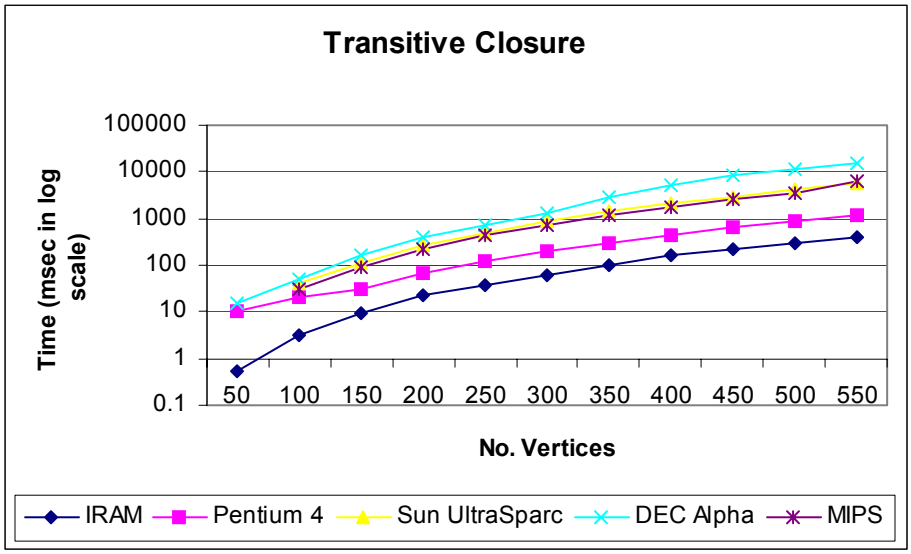
cates essentially executes in scalar mode and so degrades performance. Note that as the bit depth increased the Cray method improved due to fewer duplicates in the sum and difference data arrays. The privatized code suffers because it needs to sum up the histogram copies at the end and so expends more operations as the size of the histogram increases.

The superscalar/cache-based machines benefit when there are duplicates, since they may hit in the cache when updated a second or third time. We see sharp decreases in performance when the histograms are too large to fit entirely within cache (e.g., 15-bit input). In that case, VIRAM's performance compares well with the faster microprocessors.

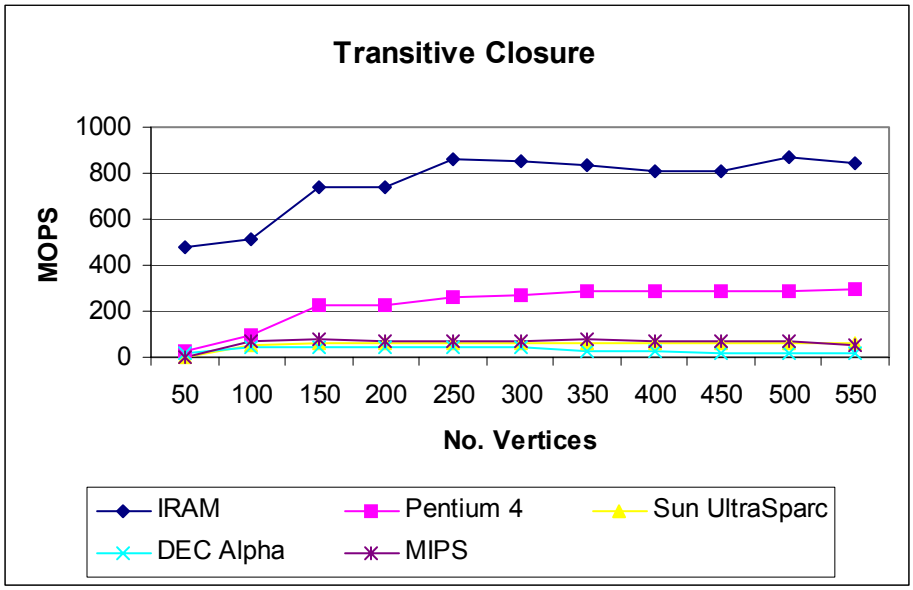
4.6 Transitive Closure Stressmark

The IRAM team did not run any of the provided test files from the Transitive Closure Stressmark, but it did supply results for a small range of parameters. Since the architecture/algorithm combination was not sensitive to the number of edges in the graph, only the number of vertices was varied.

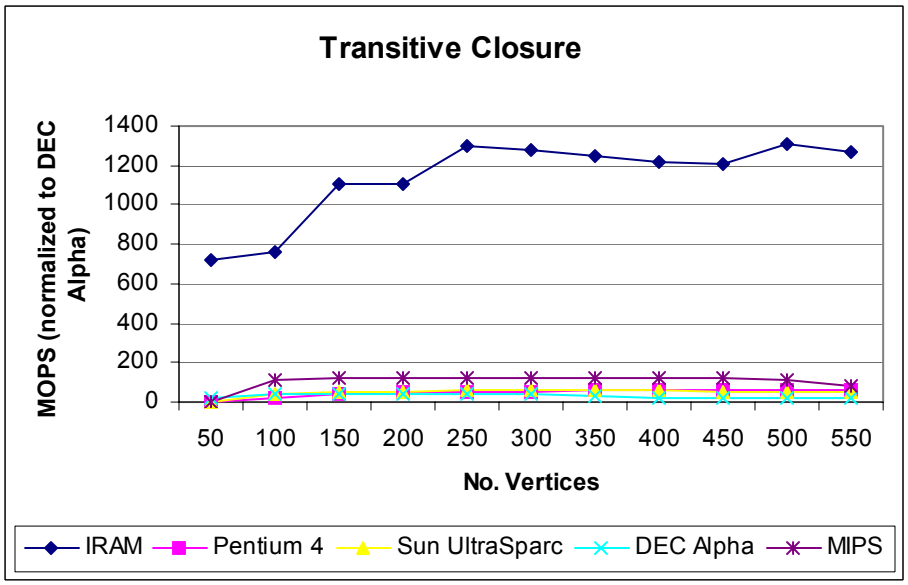
Below is a graph provided by the team showing the time for Transitive Closure runs for the four architectures described above.



To find processing rate, the team calculated MOPS = 2 (op) × n^2 (number of vertices) / time (sec) / 100K. The resultant rates are shown in the graph below.



Since each machine has a different clock cycle, it is hard to compare the performances. The graph below gives a rough estimate of how Transitive Closure would run on the different machines if the clock rate were normalized to 300MHz. These numbers are computed by multiplying the measured MOPS rate by 300 MHz divided by the listed actual cycle time.



Though real implementations of these architectures might perform quite differently than shown, it is probably fair to assume that the high relative performance shown by the IRAM would persist. As tested, though, the IRAM performance would drop to zero for problems with greater than ~600 vertices. The performance of an IRAM system configured to handle such cases cannot be inferred from the data.

4.7 Corner-Turn Stressmark

The IRAM team provided results[35] for a corner-turn operation. This test was not strictly the stressmark as specified for DIS, so results cannot be used for direct comparison.[36]

An out-of-place corner-turn of 1Kx1K 32-bit words was performed. At a 200MHz VIRAM clock rate, the operation required 5.56×10^5 cycles, or 2.78ms.

4.8 FFT Benchmark

The IRAM team did not implement the DIS DFT Benchmark. However, some FFT analysis was reported.[37] Below is an excerpt from that study:

> We compare the performance of our most-optimized FFT algorithm on a simulated version of VIRAM to that of eleven high-end fixed- and floating-point Digital Signal Processors (DSPs) and DSP-like architectures, and find that VIRAM outperforms all of the fixed-point DSPs and all but two of the special-purpose floatingpoint FFT DSPs. On 1024-point FFTs, VIRAM achieves 1.3 GFLOP/s in floating-point mode, and 1.9 GOP/s in fixed-point mode.
>
> Despite its high performance relative to the DSPs, however, we find that the VIRAM architecture is being underutilized by as much as two thirds while running the FFT algorithm. We thus embark on an architectural analysis to determine the underlying cause of this underutilization, and discover that it results from bottlenecks in VIRAM's memory functional units and memory access conflicts in VIRAM's memory system. For larger FFTs, the memory system impact becomes more severe, and we find that the number of memory banks and subbanks plays a crucial role in the scalability of our algorithm's performance to large FFT sizes.

The VIRAM design was reportedly modified after the above tests; some improvements should have resulted. New measurements were not supplied.

---

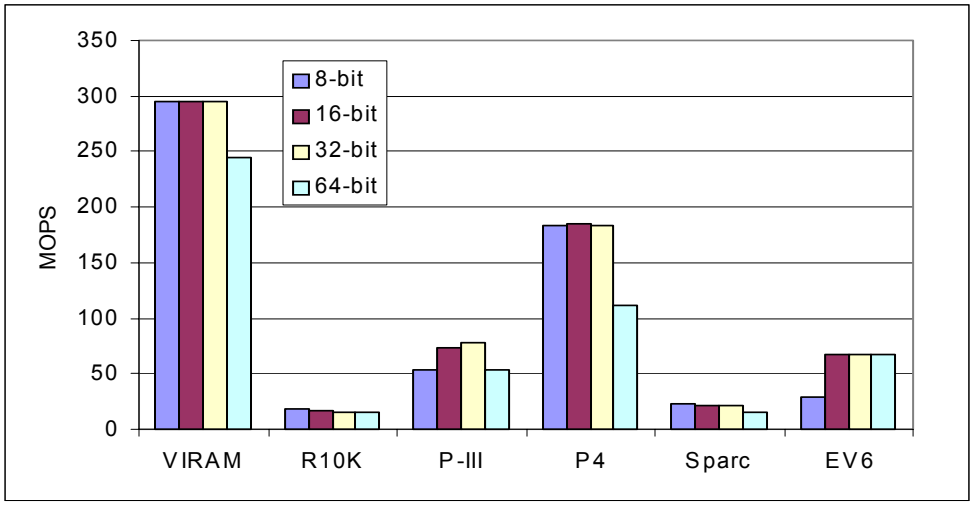
[35] Results generated by the SLIIC team from USC-ISI (East).

[36] However, the parameters of the test were very similar to those of test ct03.

[37] Randi Thomas, *An Architectural Performance Study of the Fast Fourier Transform on Vector IRAM*, Master of Science Report, UC Berkeley.

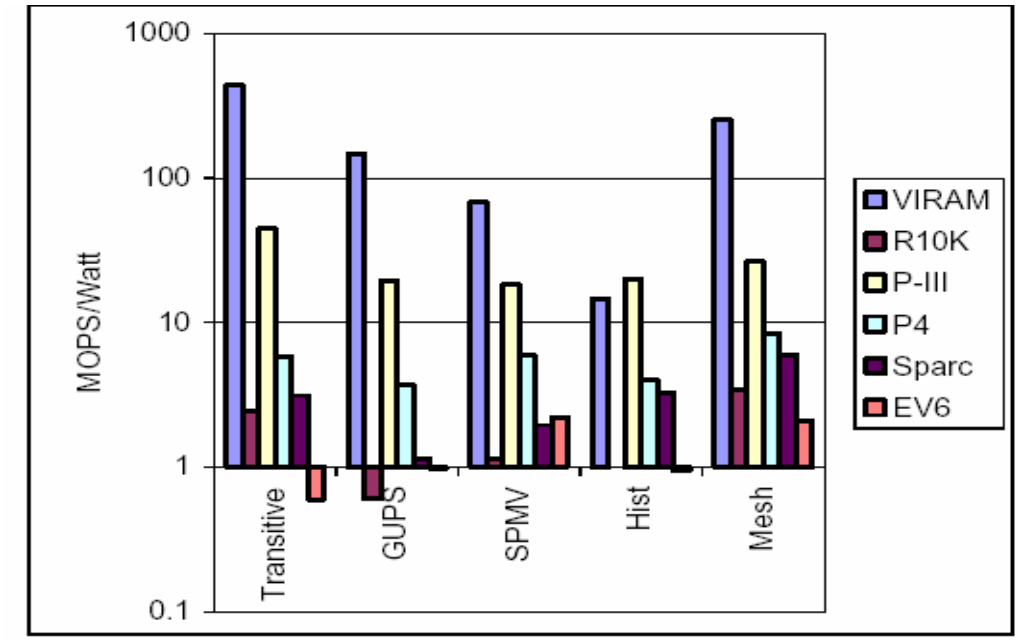
4.9 GUPS

The IRAM team gave the following results for its GUPS tests:



4.10 Power Consumption

The IRAM team supplied the following comparison of power (in terms of operations per watt) for several benchmarks, including one (Mesh) not reported here.



4.11 Programming

For benchmarking, the IRAM team utilized its compiler, which is based on Cray's vectorizing C compiler. The compiler allows programmers to assert that a loop is free of dependencies. Presumably, in the absence of this assertion, the compiler will do its best to identify dependencies and do loop transformations. It is unknown to what degree the performance reported here is dependent on manual assertions. Certainly, legacy code is fully supported, and at least partial gains are available for it.

4.12 Remarks

In its benchmarking report, the team made a valuable observation:

> While memory is important in all of our benchmarks, simple bandwidth was not sufficient. Most of the benchmarks involved irregular memory access patterns, so address generation bandwidth was important, and collisions within the memory system were sometimes a limitation.

Like most other data provided for this report, the results given here are all based on simulations. The team utilized codes and some data from the DIS Suite, but not in accordance with the supplied benchmark specifications. Still, their results showed a marked performance advantage of IRAM, especially after normalizing the clock rates. Excellent power efficiency was also found, relative to multi-chip systems.

The tests were arranged to fit within a single IRAM chip. At that size, the problems may also fit in today's cache hierarchies. The high performance of the PIII system indicates that the tests did not fully exercise the memory requirements of DIS problems. The tests were biased in the sense that only things that could be kept on-chip within the IRAM were tested. The cache-based computers were configured and ready for larger problems.

That said, the IRAM team was focused primarily on developing the chip. The team believes a multi-chip system based on the IRAM would be a candidate for DIS problems. Development work and experimentation would be required to properly determine the value and limitations of IRAM in that domain.

5 Smart Memories

The *Smart Memories* project[38] is creating the computing infrastructure for the next generation of embedded applications. The goal is to create a more universal computing component than today's microprocessor to provide the power, performance and manufacturing cost within a factor of three to a custom solution. This effort covers from programming model down to VLSI design and includes software tools to help programmers tune their application. This project leverages two important changes in computers systems: the basic wire limits of the underlying VLSI technology and the changing nature of the application space toward more streaming data. By modifying active repeaters and making them into switches, the wires will be used to connect memory banks to each other and to the processor interconnect with some reconfigurable logic to provide added functionality to create a 'smart' memory block. The same programmable wires are then used to connect these smart memories to the processors to create a multiprocessor where both the memory and the processor are programmable. Smart memories architecture is configurable to support parallel, stream-based, and legacy applications.

5.1 Description

The Smart Memory program is leveraging two important changes in computer systems—the basic wire limits of the underlying VLSI technology, and the changing nature of the application space toward more streaming data. This smart memory architecture is optimized for the wire-limited technologies of the future, and is enabled by recent advances in architecture, operating systems and compilers. In future advanced chips, wires will need active repeaters to help reduce the long wiring delays. By modifying these repeaters and making them into switches, we create reconfigurable wires with the same performance as dedicated wires. These wires will be used to connect memory banks to each other and to the processor interconnect with some reconfigurable logic to provide added functionality to create a `smart' memory block. The same programmable wires are then used to connect these smart memories to the processors to create a multiprocessor where both the memory and the processor are programmable.

A smart-memory system will function as a universal computation server because it will efficiently support many different modes of operation:

- For parallel applications, smart memories will be used to enhance the functionality of individual nodes. For example, smart memories will prefetch non-unit stride data for sparse applications, or they will be used to improve instruction bandwidth for database applications with poor instruction cache behavior.
- For stream-based applications, the interconnect will be reconfigured to deliver the performance of hardwired designs.
- For legacy applications, the smart memory will be used to implement mechanisms for exploiting fine-grain, dynamic parallelism automatically using speculative execution techniques.

5.2 Measurements

The Smart Memories team supplied no benchmarking data or report. An experiment was done by the Scalable Graphics Systems team, which utilized a Smart Memories block for a streaming ray-tracer. The experiment is discussed in Section 9.

[38] http://www-vlsi.stanford.edu/smart_memories

5.3 Programming

No benchmarks were programmed, so no examples of source code were available.

Since one of the goals of the effort was to develop an architecture that is configurable to support legacy applications, it is presumed that legacy code could operate on the system with no source modification. It is likewise assumed that no performance penalty would result in this configuration. It is unknown whether performance benefits would be available without source code modification.

5.4 Remarks

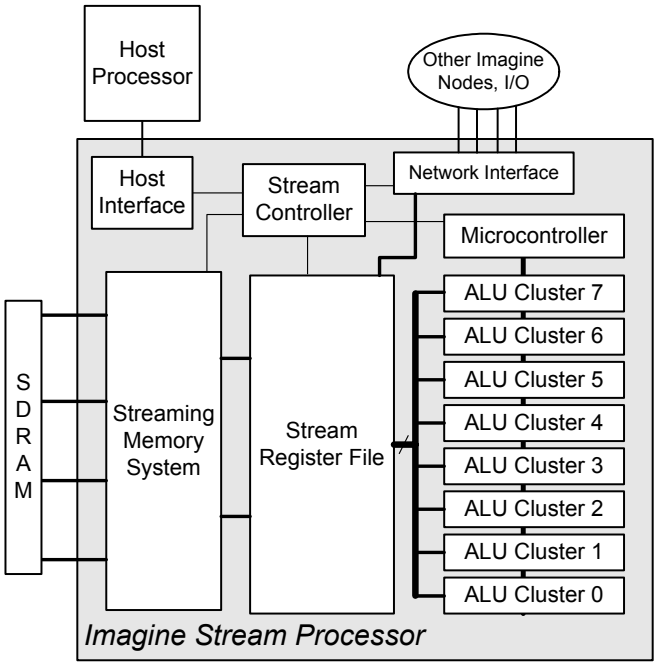
As no programming or DIS benchmarking details were provided, no conclusions are drawn. The design documents presented during project reviews and technical meetings suggest potential for value to DIS problems.

6 Imagine

Imagine[39] is designed to perform as a special-purpose image-processing, signal-processing, or graphics engine with the flexibility of a general purpose computer and a programmable signal processor. Fast stream operations are supported through a combination of vector processing, VLIW arithmetic clusters, a streaming memory system, and conditional vector operations. Imagine uses a three-level bandwidth hierarchy organized to support stream operations to overcome bandwidth bottlenecks of global register files and memory systems. Streams are transferred between memory and a 32GB/s stream register file (SRF) by a four-bank streaming memory system (2GB/s) that reorders references to improve bandwidth. Once a stream is loaded from memory, it is typically circulated between the SRF and the arithmetic clusters several times before returning the result to memory. Finally, during a computation kernel, intermediate results are forwarded directly between local register files associated with the arithmetic units without need to return to the global register file, using the 544GB/s local register bandwidth. Imagine overcomes the performance-limiting effects of conditional operations by sorting streams according to a conditional variable rather than through conditional control flow. Imagine achieves flexible performance by using a streaming architecture that exposes the parallelism and locality inherent in many signal- and image-processing applications.

6.1 Description

The *Imagine stream processor*[40] is an experimental prototype designed in a Texas Instruments 0.15 μm L_{drawn} CMOS standard cell technology. It provides full programmability and high performance by effectively exploiting the data-level and instruction-level parallelism available in computationally demanding media applications. At 1.7V and 232 MHz, Imagine provides a peak performance of 9.3 GFLOPS and 18.6 GOPS on floating-point and 16-bit integer media applications while dissipating 6.4 W.



The figure to the left contains a block diagram for Imagine, which functions as a co-processor to a scalar host CPU. The streaming memory system executes stream loads and stores by reading or writing a sequence of related data records into a contiguous block of storage in the stream register file (SRF), a 128KB on-chip SRAM. Eight SIMD arithmetic clusters execute kernels that read and write streams from the SRF. The micro-controller sequences and issues VLIW instructions during kernel execution. The network interface provides 2.3 GB/s of input and output bandwidth to support multi-Imagine systems and I/O transfers. Finally, the stream controller manages the operation of

[39] http://cva.stanford.edu/imagine

[40] Processor description from Khailany, et al, *A 9.3 GFLOPS 6.4 W Programmable Stream Processor for Media Applications*, unpublished draft provided to the author.

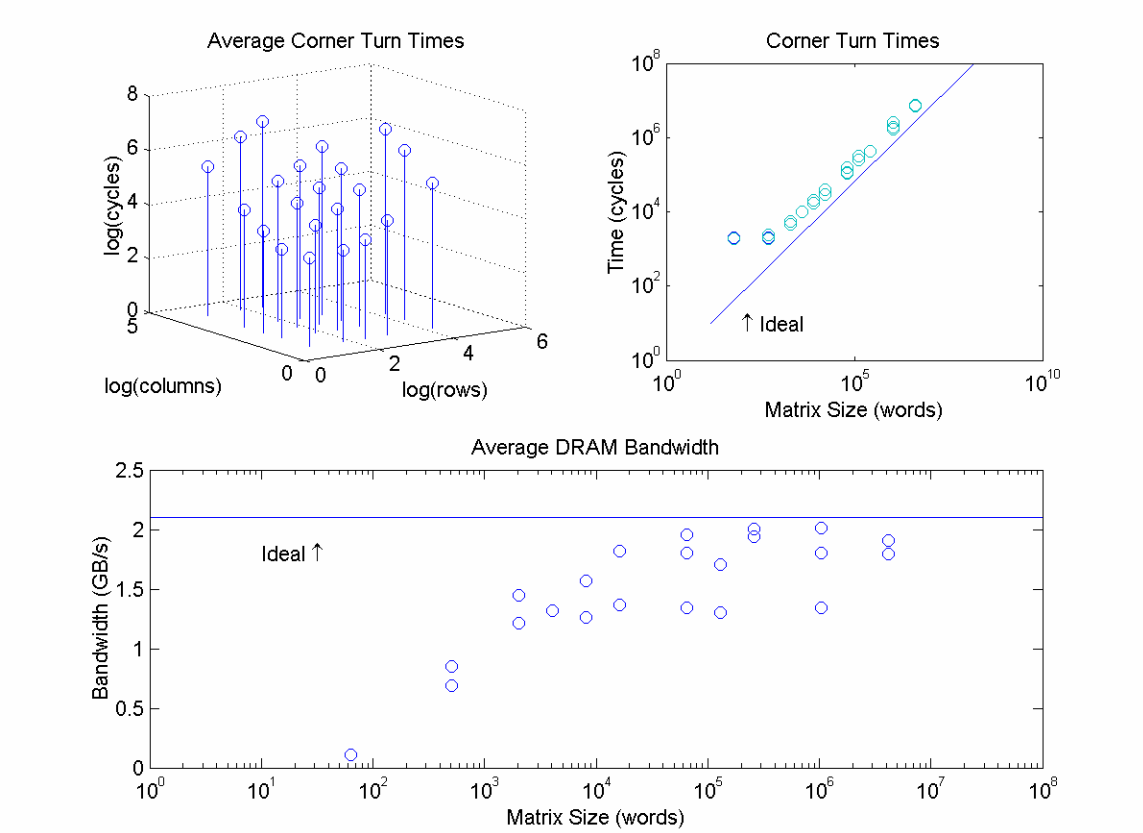
all of these units.

6.2 Measurements

The Imagine team utilized code from the Neighborhood, Matrix, and Corner-Turn stressmarks, but did not run any of the tests supplied with the suite. All provided results were simulated, typically assuming a core speed of 400MHz, and a memory system speed of 133MHz. At this rate, the peak memory system bandwidth is calculated to be 2.1GB/s.

6.2.1 Corner Turn

The Imagine team did not use the supplied test inputs. Instead, a series of matrices were turned out-of-place. The following graph shows the average results over four runs of each matrix.



The raw times (upper graphs in the figure) show the expected growth with problem size. At first look, they do not indicate any extraordinary dependence on the shape of the matrix; only on the total number of elements. The best average DRAM bandwidth (lower graph) approaches the theoretical ideal as matrices grow, peaking at about 2GB/s when the matrix includes 1M words. However, two other 1Mword cases show lower average bandwidths. Inspection of the data shows that matrices with one dimension less than 256 elements run with reduced efficiency. There does not appear to be any significant gain in efficiency for especially large matrices, but the lack of tests for very large corner-turns makes this speculative. No data are given for matrices with greater than 10M elements.

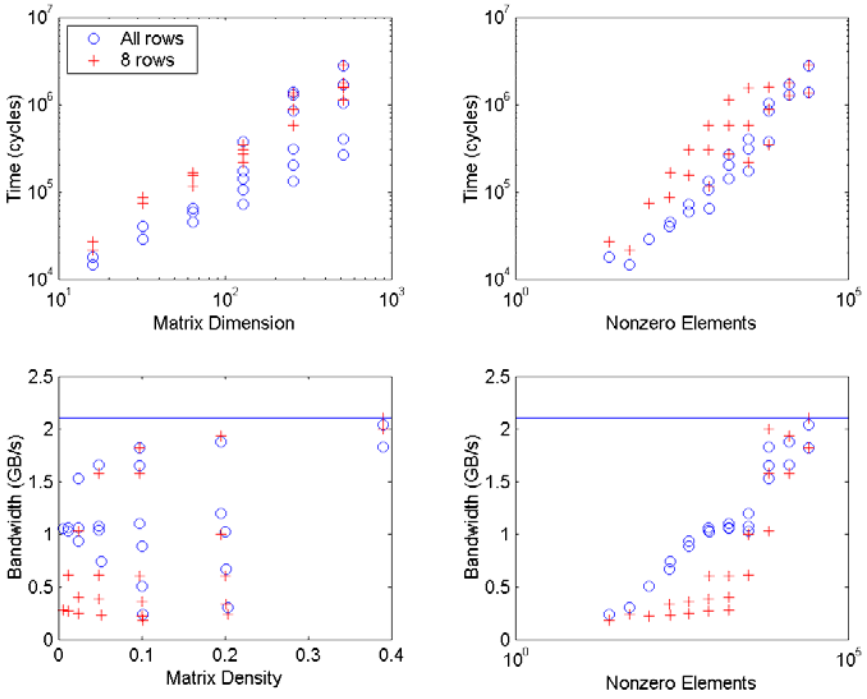
6.2.2 Matrix

The Imagine team did not use the supplied test inputs for the Matrix Stressmark. Twenty-two tests were run, with matrices ranging from 2^4 to 2^9 rows and columns, and density of nonzero elements ranging from 0.01 to 0.4.

The team provided the following information with respect to implementation of the stressmark:

> Imagine has 8 clusters, which are operated in SIMD fashion. In order to take advantage of this, 8 rows of the matrix are read and calculated in parallel.
>
> In the equation Ax = b, A is a sparse nxn matrix. Sparse matrix information is expressed by the pair of data and index. One row of the matrix A has two streams—a data stream and a index stream which stores the locations of the data in a row. In the conjugate gradient method, matrix A is only used during the time A*P (nxn * nx1), A*X (nxn * nx1) are computed. Each row can be calculated independently in these matrix computations, so this row-major order matrix representation is justified.
>
> Because A is a sparse matrix, row computation can suffer from the short stream effects—which happen if the size of the stream is short, especially when the matrix is extremely sparse.
>
> In the implementation, kernel fusions—merges of more than two kernels into the one—were done manually in order to reduce the kernel call overhead.
>
> There are two options to compute matrix-vector multiplications (A*P and A*X) in this implementation. The first option is to run the kernel per every eight rows of matrix A, and the second option is to run the kernel once per all rows of matrix A. The second option has the longer stream size, so it is better for the small size matrices and extremely sparse matrices. The first option has the shorter stream size, but it is easier to overlap the cluster computation and memory operations so it is better for larger matrix sizes.

The results of the experiments are shown in the graph below.



The results show that the system achieves memory transfer rates approaching the theoretical maximum when the matrices are large and relatively dense. Smaller, sparser matrices result in short-stream effects, and lower efficiencies. These are mixed results with respect to DIS problems. While Imagine is able to sustain operations at near-peak memory bandwidths for very large matrices, the system suffers due to the sparse nature of the problem.

The data do not support extrapolation for problems of larger matrices with lower densities, because the tests presume that the streams can fit within the Stream Register File. When this is no longer possible, strip-mining can be employed, but the cost of doing this for Imagine is unknown. For still larger matrices, even the vectors will not fit within the SRF; performance in that configuration is unknown.

6.2.3 Neighborhood

The Imagine team did some tests using the example code from the Neighborhood Stressmark, but did not use the supplied test data. The results were supplied too late to be incorporated here.

From inspection of the raw data, we find:
(a) Only small images were utilized. Some of the images had less than 100 pixels; all had less than 20K.
(b) Only bit-depths of 8 and 10 were tested. This means the histograms would always fit in registers or nearby memory, even for today's off-the-shelf systems. And,
(c) The larger images utilized larger neighborhood settings, meaning that even larger images would see a small number of histogram updates.

We conclude, therefore, that the supplied Neighborhood Stressmark data would not provide as much information for this analysis as the implementation commentary provided by the team, paraphrased here.

> The neighborhood stressmark has been written to ensure more computation can be done with the same set of streams in the SRF. This worked well for the generation of sum and difference values of pixel pairs, separated by a given distance.
>
> The histogram generation kernels need many invocations because the clusters have limited temporary storage, and hence the entire histogram cannot be stored at a time. In order to achieve this, the scratchpad has been used to generate a part of the histogram during each kernel call. The kernels themselves are long because they are limited by the "divider" unit used for scratchpad look-up. Moreover, there is a sequential dependency between memory operations and kernel invocations since the working set is big. As a combination, this has proved to be the limitation for the neighborhood stressmark.

The Imagine team's analysis suggests that—as seen for the Matrix Stressmark—irregular memory accesses can undo the gains offered by stream-oriented processing.

6.3 Programming

Imagine is a stream-oriented processor. Though the team developed tools to facilitate its use, it is unlikely that legacy code will enjoy support until significant advances are made in the state of the development art. For the foreseeable future, stream-based processing is likely to require substantial manual intervention to fully exploit the value of the devices.

6.4 Remarks

The data presented here are the result of small tests. It seems likely that larger problems would perform better—since they amortize overhead costs more—as long as local storage capacities are not exceeded. This could not be verified for this report.

High throughput seems to be shown for the tested problems. However, the streams-oriented processing is not likely to deliver performance gains for large, irregular problems. We predict that performance on Pointer, Update, and

GUPS would be poor. The architecture appears to scale well,[41] but the programming model suggests that many applications will bear a relatively high initial implementation cost.

[41] Brucek Khailany, et al., *Imagine: Media Processing for Streams*, IEEE Micro, Mar/April 2001.

7 Scalable Graphics Systems

The *Scalable Graphics Systems* project[42] focused on building scalable graphics technology to produce and manipulate imagery with several orders of magnitude more performance than currently available. The project implemented a prototype graphics system with data-parallel rendering algorithms and appropriate architectural support, and built appropriate hardware to efficiently support image display and low-level imaging operations. The project served as a demonstration mechanism for both the *Imagine* and *Smart Memories* projects.

7.1 Description

The project included a comprehensive system and architecture, most of which was not tested in the context of DIS benchmarks, and are not described here. One portion, the SHARP (Stanford Hardware-Accelerated Ray-tracing Project) architecture, was mapped to a hypothetical Smart Memories-based machine[43], modeled and simulated at a high level.

7.2 Measurements

The Scalable Graphics Systems project did not execute any DIS Benchmarks. However, as part of its demonstration, SAR ray tracing was shown, utilizing some of the scene data extracted from the DIS test data sets.

The simulation was very coarse, but based on the Stanford Smart Memories chip, the architecture showed the capacity for 90M ray-patch intersection calculations per second,[44] which would approximately meet the DIS goal.[45]

The results were based upon the one tesselated model included with the DIS benchmark set, plus approximations of some of the other scenes. However, there was a significant difference between the data used for the tests and that provided with the DIS Benchmark Suite: the former employed a regular grid-based acceleration structure, while the latter utilized a hierarchical bounding-box model. Though the tests ultimately generated correct scenes, the models used were effectively streamlined by the users *a priori*.

It could be argued that this streamlining could be implemented as a preprocessing step and therefore give identical results automatically. However, it is believed that the inefficiencies of the hierarchical representation were placed there intentionally to mimic the data-intensive nature of the more difficult real problem.[46] With simplified input files, it is unclear whether all elements of the intended DIS problem had been addressed. Additionally, the advantages and limitations inherent in the hierarchical model may not be assumed for the purposes of extrapolation to larger processing problems as derived from X-Patch.

---

[42] http://www-graphics.stanford.edu/projects/flashg

[43] Timothy J. Purcell, et al, *Ray Tracing on Programmable Graphics Hardware*, ACM Transactions on Graphics, Vol. 21, No. 3, July 2002.

[44] Presented by Pat Hanrahan at the DARPA DIS Principal Investigator's meeting, March 2002.

[45] There were actually two DIS goals: 1M ray-patch intersections per second per chip at mid-term, and 100M ray-patch intersections demonstrated at program conclusion.

[46] Engineers at ERIM International who were familiar with the problem confirmed this belief. The engineers who developed the DIS Ray Tracing benchmark were no longer with the firm and could not be located for comment.

Nevertheless, even for a grid-based structure, the bandwidth required to support such a throughput of intersection calculations is significant, and the experiment shows that stream-based calculations on a stream-oriented configuration of Smart Memories may benefit DIS problems.

7.3 Programming

The SHARP ray-tracing architecture was developed specifically for its one purpose. It cannot be considered as a general-purpose processor in the context of this evaluation. The Smart Memories hardware upon which it could be based is discussed in Section 7 of this document.

7.4 Remarks

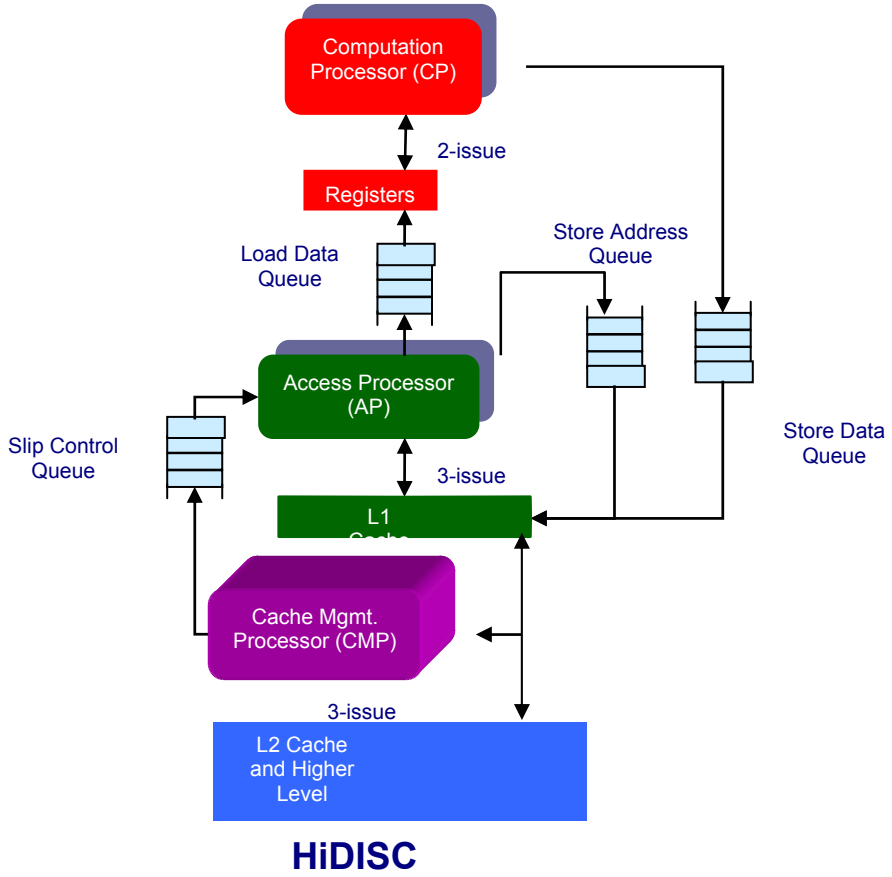
Though there were no experiments that directly utilized DIS Benchmarks, the experiment performed with the SHARP ray-tracing architecture suggests that stream-based approaches may benefit certain DIS applications. Although the SHARP architecture is directed toward programmable graphics hardware, and the Smart Memories-based machine met those requirements, the experiment did not explicitly demonstrate the value of that particular implementation above other programmable graphics implementations.

8 HiDisc

The *Hierarchical Decoupled Instruction Stream Computer* (HiDISC) system[47] adds a processor between each of the three levels of the memory hierarchy. Each processor executes an independent stream of instructions, allowing the compiler to generate code to improve the performance seen by each level of the memory hierarchy.

8.1 Description

HiDISC provides low memory access latency by introducing enhanced data prefetching techniques at both hardware and software levels. Dedicated processors for each level of the memory hierarchy act in concert to mask the memory latency. Concurrency is achieved by separating the original, single instruction stream into two streams based on the functionality—access or execute—of the instructions. Asynchronous operation of the streams provides for a temporal distance[48] between the streams, making prefetch possible.



The HiDISC architecture is a variation on traditional decoupled architectures. In addition to the two processors of the original design, HiDISC utilizes an additional one for data prefetch. A diagram of the system follows.

Along with the model, a compiler was developed that forms three streams from the original program: the computing stream, the memory access stream, and the cache management stream. These streams are stored separately in the program memory of each of the processors. Some synchronization overhead is introduced to the streams.

[47] http://www.isi.edu/acal/hidisc

[48] Sometimes referred to as *slip distance*.

8.2 Measurements

All experiments for this project were run on a special-purpose simulator based on the SimpleScalar 3.0 tool set.[49] The simulator was execution-based, with an architecture description detailed enough to include pipeline states. The system used as a baseline was *sim-outorder*, with 16 register update units and 8 load/store queues. Other reported simulation parameters are shown in the table below.

Branch predict mode	Bimodal
Branch table size	2048
Issue width	4
Window size for superscalar	RUU: 16 LSQ: 8
Slip distance for AP/CP	50 cycles
Data L1 cache configuration	128 sets, 32 block, 4 -way set associative, LRU
Data L1 cache latency	1 cycle
Unified L2 cache configuration	1024 sets, 64 block, 4 - way set associative, LRU
Unified L2 cache latency	12 cycles
Integer functional unit	ALU(x 4), MUL/DIV
Floating point functional unit	ALU(x 4), MUL/DIV
Number of memory ports	2

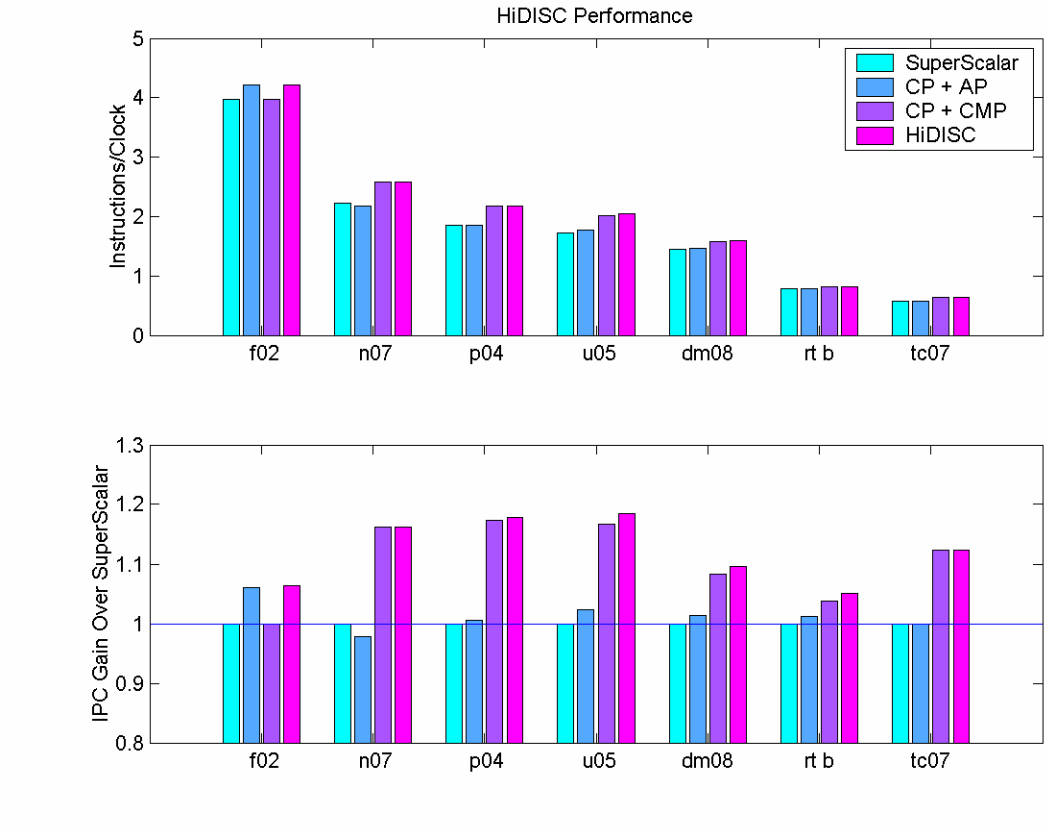
In its final report[50], the HiDISC team presented a series of experiments showing that the SuperScalar architecture exhibited lower instructions-per-clock (IPC) with longer miss latencies for most DIS benchmark codes. The results reported here are all based on the longest latency tested: 120 cycles.

[49] Doug Burger and Todd M. Austin, *The Simplecalar Tool Set, Version 2.0*. Technical report, University of Wisconsin-Madison, 1997.

[50] http://pascal.eng.uci.edu/projects/HiDISC/Final_report.pdf

The upper bar chart in the figure below gives the IPC for one test file in each of seven benchmarks. The bars represent:

- SuperScalar. This is the baseline configuration.
- CP + AP. Computation Processor and Access Processor.
- CP + CMP. Computation Processor and Cache Management Processor.
- HiDISC. Computation, Access, and Cache Management Processors.



The lower graph shows the IPC gain compared to the SuperScalar baseline. Field is the most regular stressmark kernel. It is not surprising that little gain is available from the already-high IPC rate. According to the HiDISC team, the Pointer and Update stressmarks show especially good results because pointer chasing can be executed far ahead with the decoupled access stream.

8.3 Programming

The HiDISC compiler automatically separates the access and execute streams. No special source-level operations are required, and legacy code can receive the full benefit of decoupled processing.

8.4 Remarks

The HiDISC experiments included a good variety within the scope of DIS benchmark applications, but caution should be exercised during interpretation, since only one test problem was simulated for each.

By our calculations, the system achieved the best results for the Pointer stressmark, since the access stream can execute well ahead of the computation stream. Unfortunately, this would only be true for window sizes of one. Performance is expected to decline for Pointer tests with other window sizes. We expect that problems requiring high GUPS performance would not benefit from the HiDISC approach.

In its report, the team observed, "It should be noted that the working set for the Data Management Benchmark fits quite well in the cache. As should be expected, a program with a small working set is not a good candidate for a prefetching architecture such as HiDISC." The team did not supply results for Data Management Benchmark working sets that did not fit within cache.[51]

According to the HiDISC team, the DIS benchmarks perform well with the decoupled processing in part because they include many long latency floating-point operations which can hide memory latency. The stressmark suite is more "access-heavy". The resultant disproportion between the two processing streams limits the performance of HiDISC. The team concludes that the proper application domain for decoupled architectures includes a balance between computation and memory access.

This is a mixed result for DIS. The decoupling allows hiding of memory latency in a more comprehensive fashion than simple prefetching, but DIS problems are frequently, by their nature, "access-heavy". As memory access latencies increase, the decoupling must likewise increase if processor starvation is to be avoided. Unfortunately, a given algorithm ultimately has a fixed decoupling potential. Attempts to adjust this potential by adjusting the algorithm were explored in other DIS projects.

[51] The dm08 test working set peaks at about 2MB. Test files requiring up to 1GB were provided.

9 Aries

The *Aries* project[52] is developing and prototyping a new, secure, parallel language and supporting architecture for a wide class of physical simulation and symbolic information processing tasks. The target architectures under design for these applications are SIMD-based computers, which also respond as primary memory and participate fully in the Aries multiprocessor array.

9.1 Measurements

The Aries team supplied no benchmarking data or report.

9.2 Programming

No benchmarks were programmed, so no examples of source code were available. Since one of the goals of the effort was to develop a new language, targeted primarily toward SIMD processing, it is highly unlikely that legacy code would enjoy any direct support.

9.3 Remarks

No benchmark results were available for evaluation.

[52] http://www.ai.mit.edu/projects/aries/aries.html

10 Impulse

The *Impulse* project[53] developed a memory controller that adds an optional extra level of address remapping, reducing the wastes of bandwidth and cache capacity afflicting current systems. A compiler was developed to support automatic detection and exploitation of opportunities for Impulse optimizations. The compiler can analyze data locality, improve it statically with loop optimizations, and introduce appropriate Impulse data remappings, prefetching, and superpage formation in off-the-shelf programs. The Impulse system enables scatter-gather access of sparse or arbitrarily organized data.

The Impulse approach is to modify the main memory controller, not the CPU or memory itself. It can therefore be retrofitted to existing systems. The team's target was a 10-fold increase in effective memory bandwidth of memory-bound programs that exhibit irregular memory access patterns.

10.1 Description

Impulse's extra level of address indirection is implemented as follows. Consider a workstation that employs a microprocessor that exports a 32-bit physical address and thus can address four gigabytes of physical memory. If only one gigabyte of DRAM is installed in the machine, an access to the other three gigabytes of physical addresses will generate a bus error. Impulse allows software to specify remapping functions that translate these so-called *shadow physical addresses* to physical addresses (ones directly backed by DRAM). Rather than moving entire cache lines to the cache/CPU, where only a fraction of their contents will be used, Impulse can compress sequences of useful values into cache-line-sized blocks.

The optimizations that this remapping enables include scatter-gather access of sparse data, creation of superpages from disjoint and unaligned physical memory, no-copy tiling of dense matrices, and pointer-based prefetching of C and C++ objects.

A novel feature of Impulse is the ability for a processor to issue multiple (up to 32) load requests to the memory controller in a single bus operation. The processor does so by filling a cache line with offsets or addresses, flushing the line back to the controller, and then fetching a corresponding data block where the controller has been configured to place the requested data. This optimization speeds up random pointer fetching by more than a factor of three[54], and is applicable in many situations where static remappings are not.

The team automated the use of Impulse by developing a compiler, which analyzes data locality and applies appropriate Impulse optimizations. Language extensions were also provided, so developers can manually introduce remappings to their code.

10.2 Measurements

The team delivered results for four DIS stressmarks: Pointer, Matrix, Transitive Closure, and Corner Turn. Tests were performed on a simulator, and later verified with hardware.[55] The simulation used URSIM, an execution-driven simulator derived from RSIM[56]. All tests were done twice: once with remapping support, and once without.

---

[53] http://www.cs.utah.edu/projects/impulse

[54] According to the project summary. The specific claim was not tested for this report.

[55] Data provided for this report are all the result of simulations.

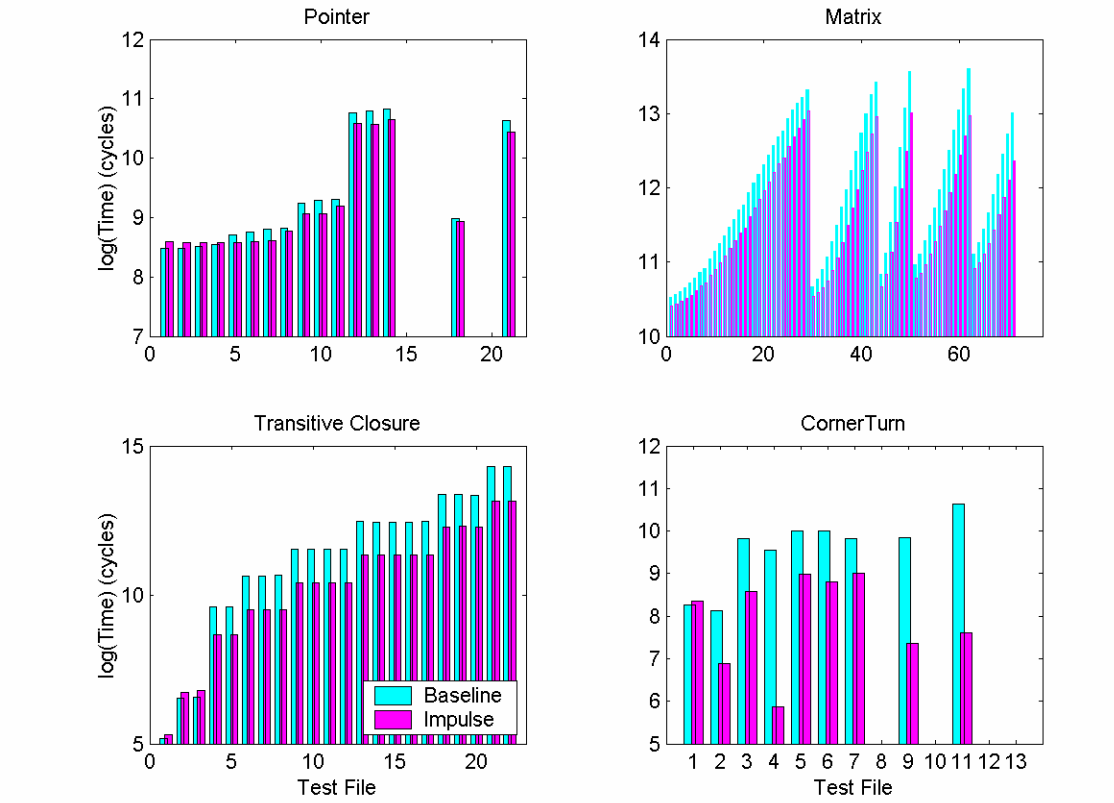
The following simulation parameters—modeled after a R10000-based SGI O200 system—were reported:

Clock rates	Processor: 450 MHz System bus, memory controller, and SDRAM: 150 MHz
Processor	Four-way issue superscalar, with a 32-entry instruction window and a 16-entry memory queue
Branch prediction	512 entries, two-bit prediction scheme, can predict four branches deep
Page size	The base page size is 16KB. Superpages are built from powers-of-two multiples of base pages. The largest superpage is 64MB. The TLB is fully associative and hardware-managed. It has single-cycle latency and 64 entries. Each TLB entry is able to map a pair of properly-aligned, consecutive pages.
L1 Instruction cache	Non-blocking, virtually indexed, physically tagged, two-way associative. It has 32KB of storage, 64B lines, and a one-cycle latency.
L1 Data cache	Non-blocking, write-back, virtually indexed, physically tagged, two-way associative. It has 32KB of storage, 32B lines, and a one-cycle latency.
L2 cache	Non-blocking, write-back, physically indexed, physically tagged, two-way associative. It has 256KB of storage, 128B lines, and a six-cycle latency with a two-cycle repeat rate.
System bus	The system bus multiplexes addresses and data. It is 64 bits wide, has a three-cycle arbitration delay and a one-cycle turn-around time, and supports eight outstanding cache misses. It supports critical-word-first. The memory latency, defined as the time between when a request is presented on the bus and when the first word returns to the processor, is 14 bus cycles for a normal access, and 7 bus cycles for an access that hits in the Mcache.
DRAM banks	The DRAM backend contains two 128-bit data buses, each of which has four banks. Banks are interleaved at the L2-cache-line level.
Memory controller	The Impulse memory controller contains eight shadow descriptors. The shadow engine contains two data-path pipelines. Each pipeline has a dedicated MTLB, which is four-way set associative, has (32) 128-byte entries, and a one-memory-cycle latency. For a shadow address, the shadow engine generates the first physical address in four cycles and then one address per cycle thereafter if no MTLB miss occurs.
Compiler	All benchmarks compiled using the SPARC SC5.0 compiler, with an optimization level of "-xO4".

Note that the automatic Impulse compiler was not utilized for these benchmarks. Small changes to source code were made to manually invoke Impulse optimizations. Examples of the code changes are provided later in this section.

[56] http://rsim.cs.uiuc.edu/rsim/

The following graph shows the raw reported times for each of the four stressmarks.



From a cursory inspection of the raw data, it can be seen that Impulse offers a consistent performance gain. Very few tests show a loss of performance, and we will see that these cases are for tests requiring very small amounts of memory. There is no ambiguity about the gains, since the comparison is between configurations including the same processor, bus, clock rates, and memory chips.

10.3 Pointer

The Impulse team described its implementation of the Pointer stressmark as follows:

> To optimize Pointer using Impulse, the basic idea is first to swap the "l" for loop with the "hops" while loop, then to gather the first windows for all threads into dense cache lines, then to let each thread process its respective window and generate address for the second window, then to gather the second windows for all threads, and so on.

Below is the source for both versions (remapped and non-remapped) of the stressmark. The code illustrates the required level of programmer involvement when manually mapping shadow addresses.

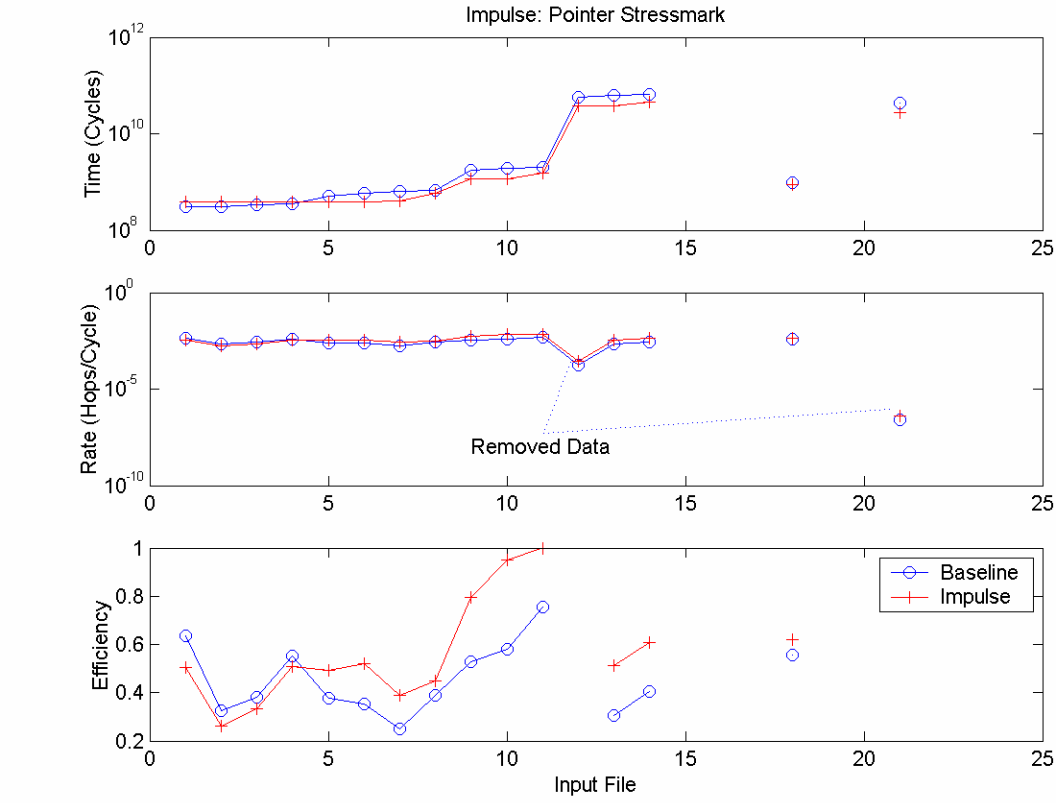
```
/*
To improve the performance of this stressmark, we swap the inner loop
with the outer loop and then use "dynamic cache-line assembly".  The
basic idea is to gather the first windows of all threads, then let all
threads work on their respective windows to generate the addresses for
the second windows; then gather the second windows for all threads,
generate addresses for the third windows; and so on.
*/
```

```
#ifndef IMPULSE
   for (l = 0; l < n; l++) {
      hops = 0;
      minStop = thread[l].minStop;
      maxStop = thread[l].maxStop;
      index  = threads[l].initial;
      while ((hops < maxhops) &&
             (!((index >= minStop) &&
                (index < maxStop)))) {
         .....;
         index = (partition + hops) % (f - w);
         hops++;
      }
      threads[l].hops = hops;
   }
#else /* Remapping version */
   /* setup Impulse MMC to map ap[i] => fields[idx[i]] */
   Impulse_setup(fields, w, (unsigned *) &ap, (unsigned *) &idx);

   /* Gather the first windows of all threads */
   for (i = 0; i < n; i += OBJS_IN_L2CLINE) {
      for (j = i; j < i + OBJS_IN_L2CLINE; j++) {
         idx[j] = threads[j].initial;
         threads[j].hops = maxhops;
      }
      prefetchline((unsigned) &(ap[i]));
   }
   for (hops = 0; hops < maxhops; hops++) {
      completed_threads = 0;
      for (j = 0; j < n; j += OBJS_IN_L2CLINE) {
         k = MIN(j + OBJS_IN_L2CLINE, n);
        for (i = j; i < k; i++) {
           ......;
           idx[i] = (partition + hops) % (f - w);
            if (this thread completes) {
               some bookkeeping;
               completed_threads++;
              idx[i] = -1;
            }
        }
         flush_then_prefetchline((unsigned) &(ap[j]));
      }
      if (completed_threads) {
         /* Compress "ap" by taking out completed threads. Since this happens
            at most "n" times, we don't care the efficienty. */
         for (j = i = 0; j < n; j++) {
            if (idx[j] != -1) {
               ap[i++] = ap[j];
               some bookkeeping;
            }
         }
         for (; i < n; i++)
            idx[i] = 0;
         n -= magic_number_found;
      }
   }
#endif
```

It was observed that the team did not execute any of the tests that included a *window* parameter greater than one. When queried about this, the team responded:

> Impulse MMC currently can handle only power-of-two object sizes, so we cannot use it for w > 1, because w must be an odd number. We normally pad odd-sized objects, however, padding cannot work for this benchmark. Impulse MMC also requires each scattered/gathered object to be aligned with its size. For instance, if the size of an object is 16, the last four bits of its address must be 0000. In Pointer, a window can start at any position of the array, thus may not be aligned with the window size. For example, assuming A[0] = 0x100000, a window of three elements starting at A[2] (i.e., 0x100008) will not be properly aligned if we expand it to four elements. One possible way to gather data when w > 1 is to generate addresses for each element, instead of each window, but we haven't tried to experiment with this approach yet.
>
> Impulse MMC always gathers a full cache line for a shadow request. When n < 8, n windows can fill only part of a cache line; gathering of the other part of the cache line would be wasted, so Impulse is unsuitable for such cases. As a matter of fact, remapping's speedup for the standard set of inputs are not as good as those we published in our papers. We used an input with 64 threads, while all standard inputs have no more than 16 threads.

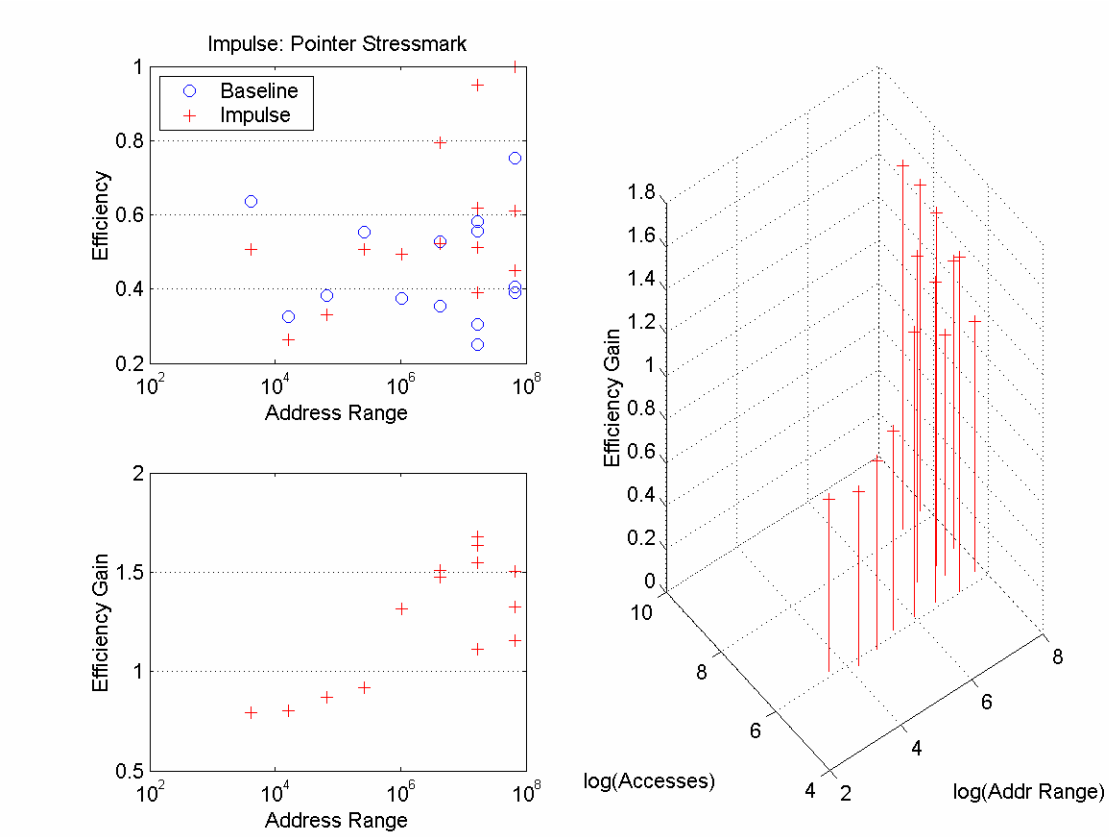
The following graph shows the data in three forms: raw, by processing rate (i.e., normalized by work), and processing efficiency.



After calculating the processing rate, the data points associated with tests p12 and p21 showed unusually slow processing rates, both for the baseline (non-remapping) and Impulse (remapping) configurations. Those data were therefore discarded. It is likely that an error in random number generation resulted in more pointer 'hops' for the Impulse

team than the benchmark required. The Impulse team did not provide output files, so correct execution of the remaining tests is assumed. The analyses here do not suggest any problems other than the two aforementioned tests.

The efficiency is found by normalizing *both* series to the best rate of *either* series. This is valid since the architectures are identical, outside of the memory controller.



Viewing the efficiency against address range (top left of above graph), some efficiency loss for very small problems can be seen. This loss is likely to be the manifestation of initialization overhead for the Impulse system. Above 1MB or so, Impulse offers increased efficiency (bottom left). Note that the efficiency gain is not always high for the larger problems. With the gain shown versus both address range and number of accesses (right), it can be seen that tests having large fields but a small number of hops do not experience the full benefit of Impulse.

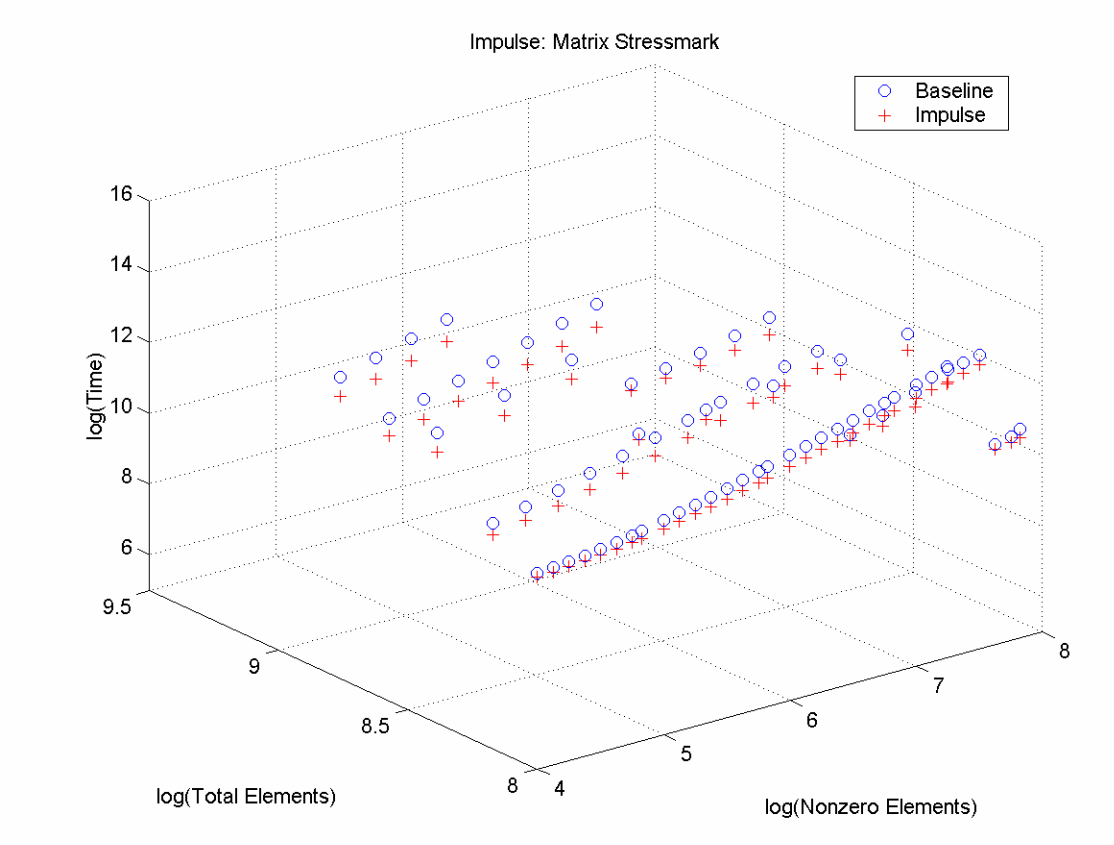
10.4 Matrix

The Impulse team supplied the following code fragment to illustrate the optimization of the Matrix Stressmark.

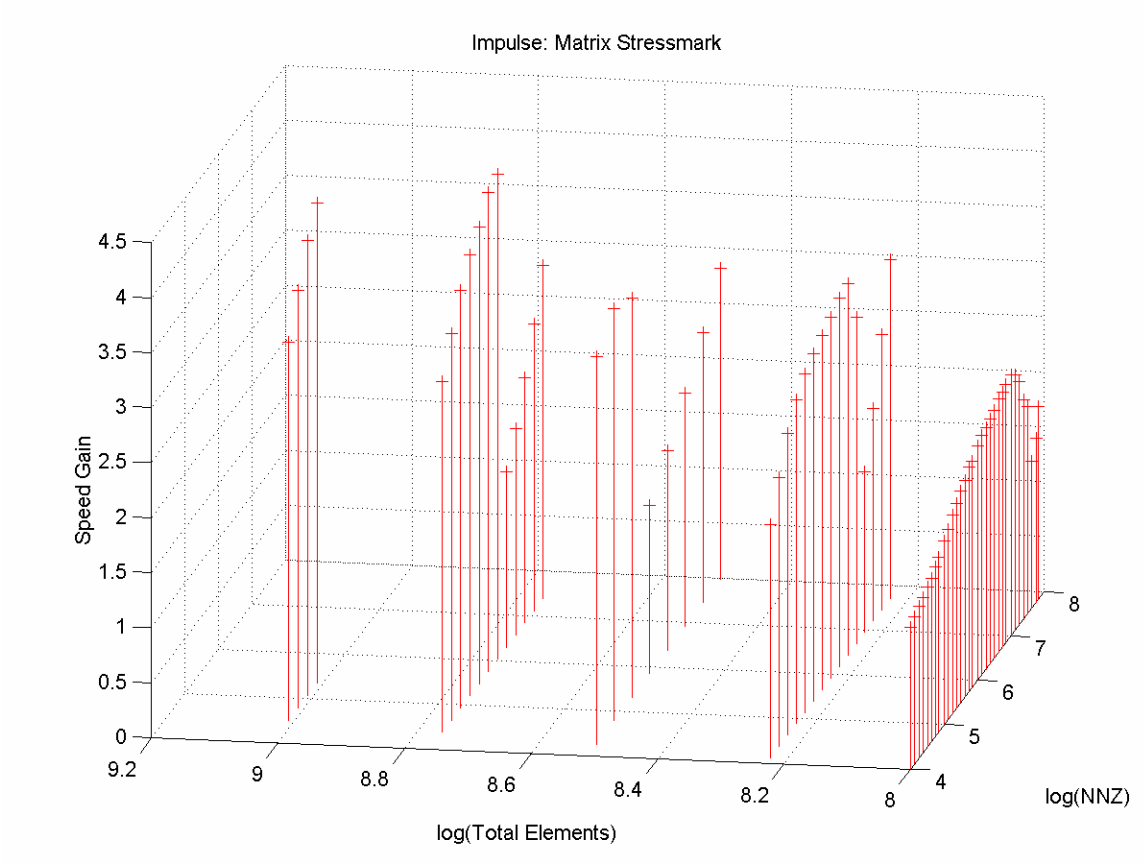
```
/*
The core computation of Matrix is the sparse matrix-vector product
alogirhtm implemented in function matrixMulvector(). Impulse uses
"scatter/gather via an indirection vector" to change indirect accesses
inside this function to direct accesses. The source code of this
function looks like the following:
*/
void
matrixMulvector(double *value,
```

```
                int *col_ind,
                int *row_start,
#ifndef IMPULSE
                double *vector,
#else
                double *alias_vector,
#endif
                double *out)
{
   int l, ll;
   double sum;
   int tmp_rs, tmp_re;

   ll = row_start[0];
   for (l = 0; l < dim; l++) {
      sum = 0;
      tmp_rs = row_start[l];

      if (tmp_rs != -1){
         tmp_re = row_start[l+1];

         for (ll=tmp_rs; ll < tmp_re; ll++) {
#ifndef IMPULSE
            sum += value[ll] * vector[col_ind[ll]];
#else
            sum += value[ll] * alias_vector[ll];
#endif
         }
      }
      out[l] = sum;
   }
}
```

Significantly, only one iteration of the main loop was executed for each problem. Below are the results graphed against the two primary problem specifications.

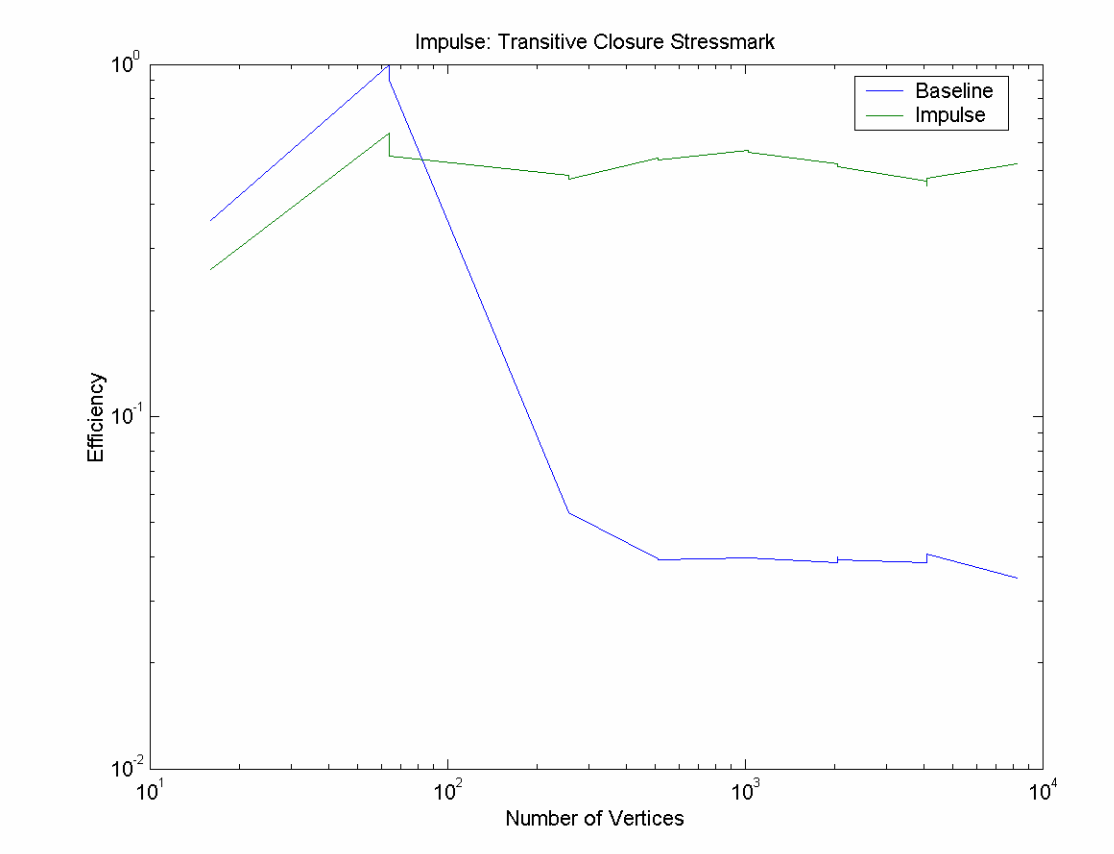


For this stressmark, it can be seen that performance gains are offered with excellent consistency. Impulse performance is never worse than the baseline, and it improves as a function of the matrix size. When the matrix is small, speed increases with the matrix density. The graph below shows the speed gain relative to the baseline (non-remapped) configuration.


Impulse: Matrix Stressmark
Speed Gain
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
9.2
9
8.8
8.6
8.4
8.2
8
log(Total Elements)
4
5
6
7
8
log(NNZ)

10.5 Transitive Closure

For Transitive Closure, the Impulse team remapped memory such that both row-major and column-major array accesses appeared as row-major to the processor. The efficiencies of the tests are shown below.

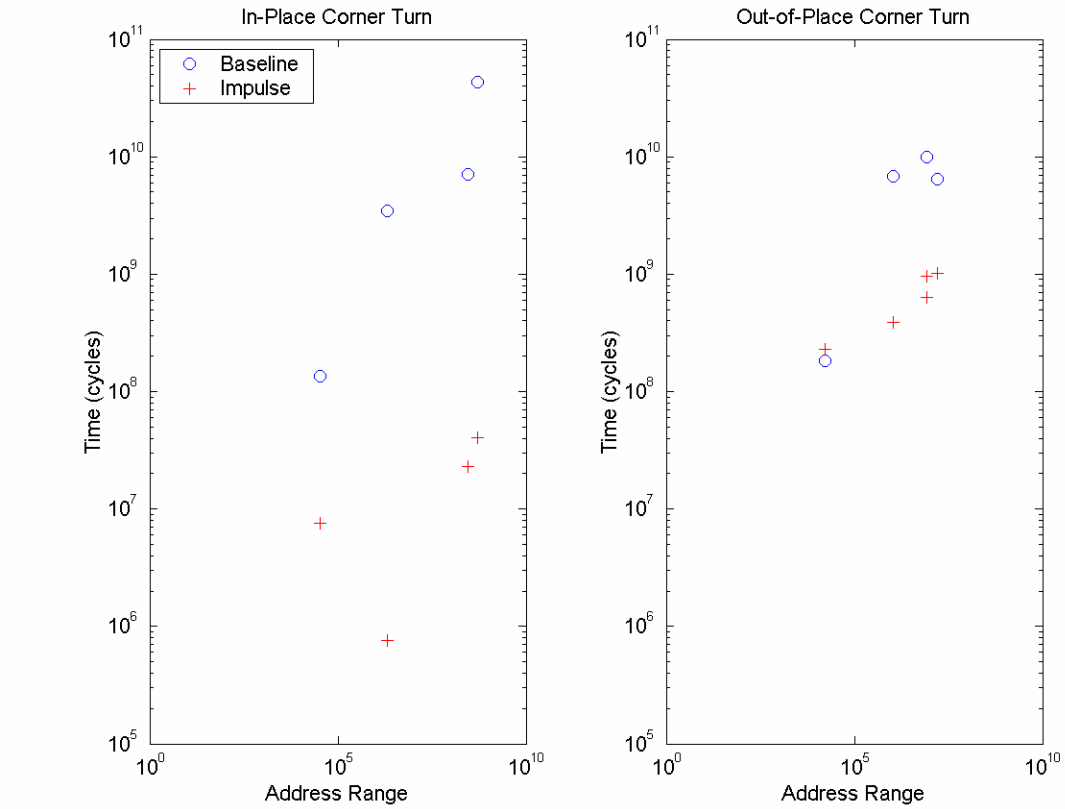


The baseline configuration shows the expected performance curve: efficiency peaks for a small problem size (one that is well-suited to maximize the use of the cache hierarchy), then drops substantially for larger problems. The Impulse configuration maintains high efficiency throughout the range of tested problem sizes. In other words, the utility of the processor is extended to a wider range of problems.

10.6 Corner-Turn

For Impulse, remapping allows memory to be accessed in column-major (or other) order as quickly as row-major order. In that sense, an in-place corner-turn is just the time required to map the array into the shadow address space. An out-of-place turn simply requires the addition of a memcpy() to duplicate the array. In practice, an out-of-place turn would never be done in an Impulse system, since a single array can be accessed in either order without duplication.

The graph below gives the reported time data. As expected, the Impulse system is dramatically faster for in-place operations.



10.7 Programming

Examples and descriptions of the programming of Impulse appear along with the stressmark implementation discussions above. The primary consideration here is that the user has the opportunity to declare alternate mappings of memory. To exploit this for new applications and code would require a negligible amount of additional programming effort.

For legacy code, the Impulse memory controller operates as a normal controller. In this circumstance, no processing gains are offered, but there are no losses, either.

Though it was not utilized for the benchmarking effort, the team's compiler appears to allow automatic optimization of code, offering performance advantages even without additional programming effort.

10.8 Remarks

Since the Impulse team tested configurations that were identical except for the memory controller, the results give the most direct comparison possible. While measurements provided here were from simulations, we can safely conclude that the reckoned value of Impulse is not the consequence of clock rates, CMOS fabrication process, or any other of a myriad of design variables.

After submitting its benchmark measurements for this report, the Impulse team verified its simulations using a prototype system. It is unknown what disparities, if any, were found. We assume the verification confirmed the findings presented here.

The project demonstrated excellent efficiency improvement, resulting in a performance gain of 2.24 on average across tested applications. For in-place corner turns, 2-3 orders of magnitude of gain were shown.[57]

The remapping mechanisms enabled by Impulse effectively answer one of the complaints of software engineers: that of the loss of throughput when accessing data by means other than row-major order. The cost of using the language extensions is relatively low—or nil if the compiler can completely obviate them. The Impulse team claims that the controller itself is no slower with Impulse, meaning that code which does not or cannot exploit the remapping will suffer no performance loss compared to that of standard controllers.

One limitation of Impulse is the finite availability of shadow address space. Especially in the case of retrofit to existing systems, the Impulse approach may be of limited value in machines possessing a large fraction of addressable memory as physical RAM.

[57] Recall that corner-turning requires a change in memory access configuration. In an Impulse system, the need to accomplish this by moving or copying data is obviated, giving extremely good performance for this step.

11 Malleable Caches

The *Malleable Caches* project[58] is providing the ability to control caches to dramatically improve their utility and power consumption. Through column caching and curious caching, malleable caches permit many architectural optimizations, such as power management, adaptive prefetching, and support for vector operations. A flexible indexing technique enables parts of the cache to be used as general-purpose associative memory. Finally, a "cache pressure gauge" allows better overall use of cache resources in a dynamic, on-line fashion. These mechanisms enable adaptive cache management by providing the user, compiler, or operating system finer control over cache resources, which in turn can result in dramatic speedups for stream-based and real-time applications.

11.1 Description

The Malleable Caches project investigated several cache-related strategies, including:

- Better cache management for multi-processing
- Better OS scheduling
- Cache Compression
- Curious Caching
- Aggressive, adaptive prefetching

However, only Column Caching, used for improved cache management of multi-processing systems, was explored in the context of DIS benchmarks. In this scheme, each process gets exclusive use of a portion of the cache, and all processes additionally share the use of common portions. In the experiments described below, the allocation of the dedicated portions of cache is static over the life of each process.

11.2 Measurements

The Malleable Caches team supplied no benchmarking data or report, however, during its program review in March 2001, the use of DIS benchmarks was discussed.

The team found that, run in isolation, the techniques developed for the project did not significantly help the processing of the DIS benchmarks. The focus, then, shifted toward a more realistic problem domain: the execution of multiple stressmarks concurrently, with cache shared among processes.

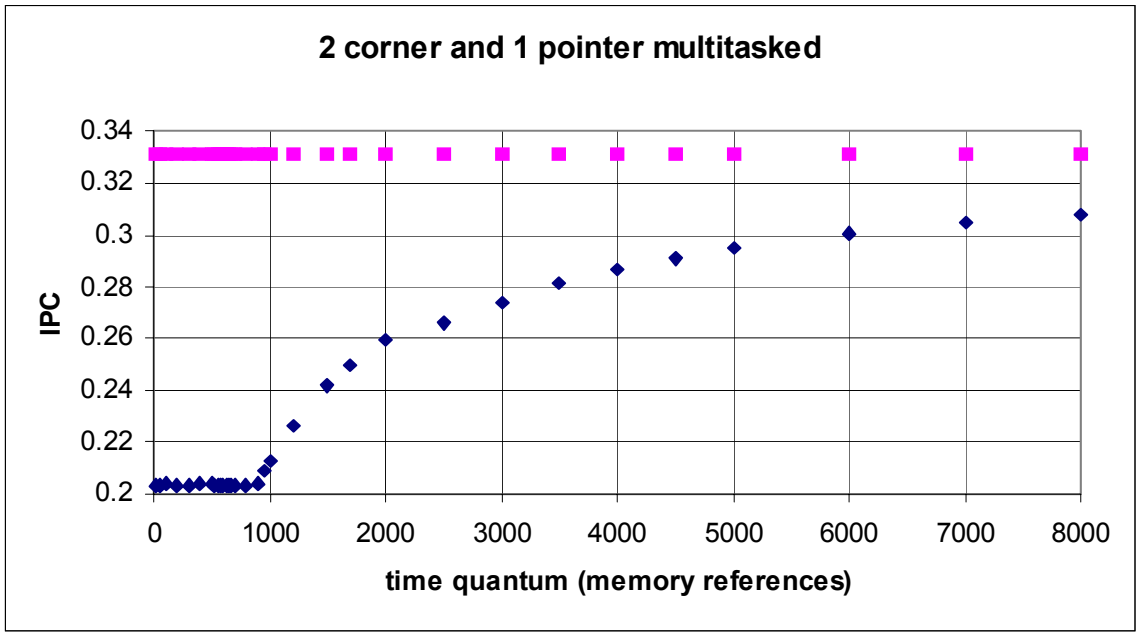
The team's experimental results are reproduced here without further analysis. All of these tests assumed a single-issue, in-order RISC machine, with one cycle of latency to a single, 4KB, fully-associative cache, and 100-cycle latency to DRAM. The experiments also utilized stressmark code that employed programmer and compiler optimizations, though the details of those optimizations were not given.

The tests involved simultaneous execution of multiple stressmarks, and sought to compare the merits of a Least-Recently Used (LRU) policy against static cache partitioning. The stressmark input parameters were not given.

[58] http://csg.lcs.mit.edu/projects/?action=viewProject&projectID=8&projectGroup=Architecture

11.2.1 First Experiment

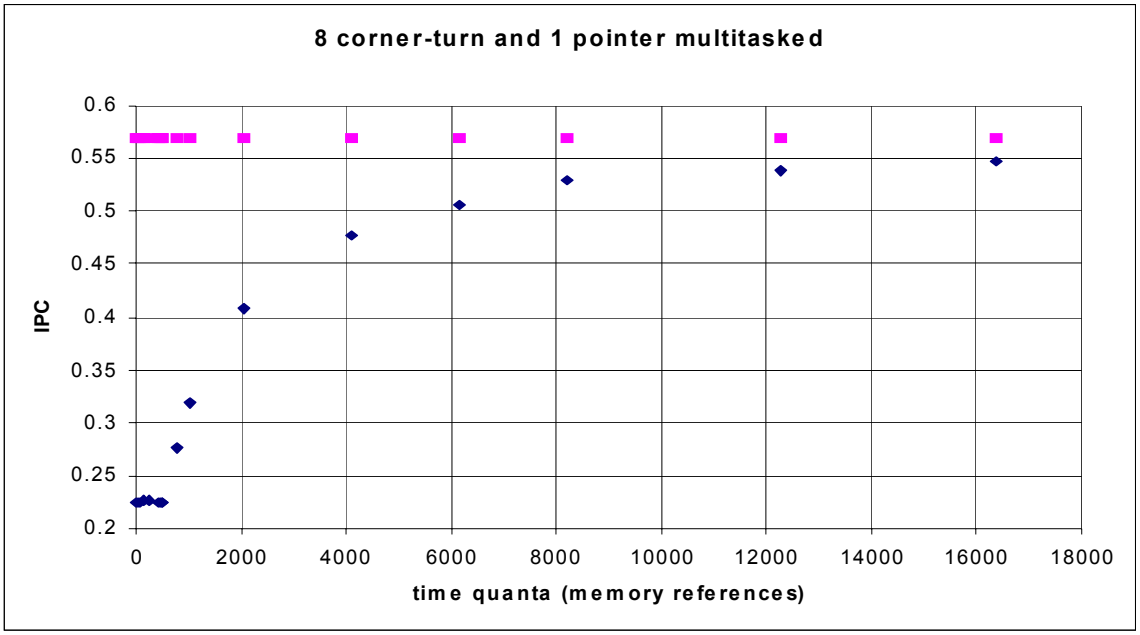
The first experiment involved two Corner-Turn tasks and one Pointer task. The graph below gives the Instructions Per Cycle (IPC) found by simulation of the LRU (blue diamond) and Column Caching (pink square) policies. The *time quantum* measurement refers to how many memory references are performed between context-switches.



The results show that as the time quantum becomes large, the LRU policy becomes increasingly appropriate. This can be understood simply as a function of the process enjoying more time to clean out the cache, which was polluted by the competing processes. If the quantum is sufficiently large, the LRU will outperform Column Caching. For small time-quanta, though, Column Caching offers a 65% improvement in performance.

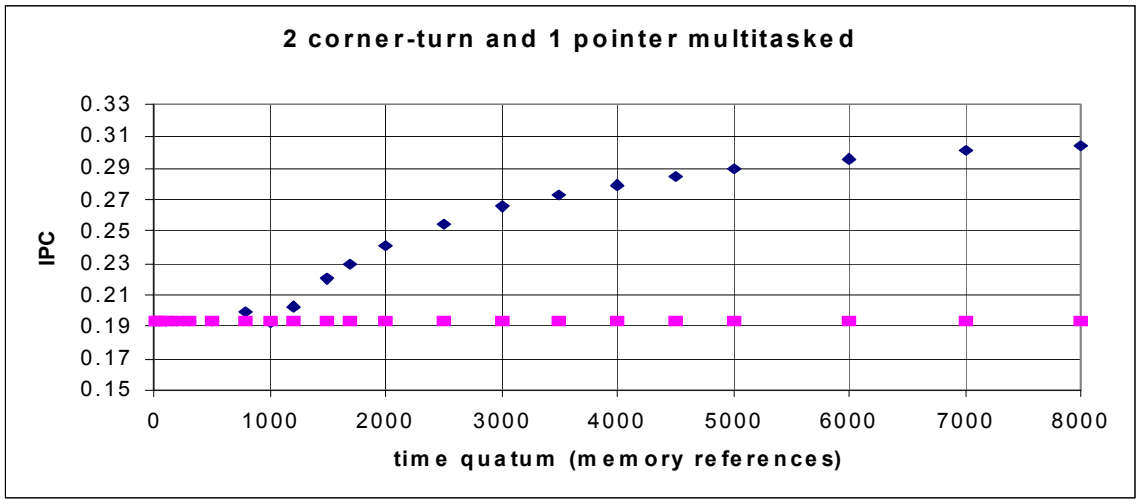
11.2.2 Second Experiment

The second experiment showed that for a somewhat more complex situation, Column Caching gave a more significant performance improvement. The graph below shows IPC (again, blue diamond for LRU, pink square for Column Caching) for a run of eight Corner-Turn tasks with one Pointer task. For small time quanta, Column Caching shows a performance gain of 2.5.



11.2.3 Third Experiment

To illustrate the fact that poor partitioning can lead to bad performance of Column Caching, the Malleable Caches team showed the following graph, which shows the same circumstances as the first experiment, except with a different static partition. As can be seen, in this case Column Caching gives no performance gain for small time quanta, and a performance loss for large quanta.



11.3 Programming

No examples of source code were available, but the team worked with both sample versions of the stressmark code as well as hand-optimized versions. The team's approach to cache management did not presume source-level language extensions for all explorations, but it did in the case of Column Caching. It is assumed that the approach would fully support legacy code—perhaps with some performance loss—and deduced that performance improvements are only available to code that utilizes special library functions.

11.4 Remarks

The maximum performance gain demonstrated by the experiments was 2.5 for the concurrent execution of nine stressmark tasks with a small time-quantum between context switches. The Malleable Caches team was the only one to experiment with caches shared between concurrent processes. The research suggests that column caching can improve performance for data-intensive computing under certain conditions, even after code optimizations. However, it appears that a method to find the proper balance of columns within the cache does not exist for arbitrary (combinations of) problems. Even when a problem may benefit, the techniques must be applied with particular care; performance may suffer otherwise.

The remaining techniques explored by the team are not within the scope of this document. However, in its final PI meeting presentation, the team summarized its experiments as follows:

What Did Not Work	What Did Work
Experiments in the domain of general-purpose microprocessors (improving cache is a second-order effect)	Experiments in the domains of tiled architectures and network processors
Column caching by itself	Column and curios caching combined
Curious caching by itself (cache pollution troublesome)	Adaptive L2 compression
Non-adaptive L2 compression (decompression overhead too costly)	Cache-aware scheduling

12 AMRM

The Adaptive Memory Reconfiguration and Management (AMRM) project[59] developed a cache architecture, smart compiler algorithms, and operating system strategies to deliver increased memory system efficiency by enabling applications to manage placement and movement of data through the memory hierarchy. This effort emphasized application-adaptive architectural mechanisms, hardware-assisted blocking, prefetching, and dynamic cache structures.

12.1 Description

The AMRM approach to the DIS problem emphasized that performance gains could be realized without reconfigurable circuits. Instead, key portions of the memory hierarchy are made adaptive and controlled by software at compile- and run-time. Specifically, the cache system and supporting elements are designed to enable multiple data movement structures and policies, including:

- multilevel caches
- intelligent prefetching schemes
- dynamic, "cache-like" structures (such as prediction tables, stream caches, and victim caches); and
- multiple cache configurations.

The AMRM goals centered on reduction of all three forms of cache-misses (conflict, capacity, and compulsory).[60] To this end, several methods were exploited:

- Adaptive Line Size Cache (ALS), which uses different line sizes concurrently;
- Adaptive Fetch Line Cache (AFL), which uses a more simple adaptation mechanism on the presumption that line sizes need only change relatively slowly over the life of a program.
- Stream Buffers (SB), or prediction-based prefetching; and
- Victim Cache (VC), which involves a mid-level associate cache placed in the cache hierarchy to hold data evicted from the upper-level cache.

As part of its effort, the AMRM team examined analyzed load stream behavior, and developed a hardware-based technique to perform behavior classification on the fly, which will enable machines to be access-pattern-aware. The team classified load patterns into four types: next-line, stride, same-object (additional misses that occur to a recently accessed object), and pointer-based transitions. The team reported that these four classes represent more than 90% of all cache-misses in the examined programs, and that the automatic technique can accurately classify 85% of all misses, on average across programs.

Interestingly, the team reported that complete removal of the access latency for any one of these classes does not provide noticeable benefits for most of the tested applications. A system would have to hide the latency for multiple classes in order to yield a substantial benefit.

[59] http://www.ics.uci.edu/~amrm

[60] In the case of compulsory misses, strictly speaking AMRM attempted to hide the effects, rather than reduce the number of actual misses.

12.2 Measurements

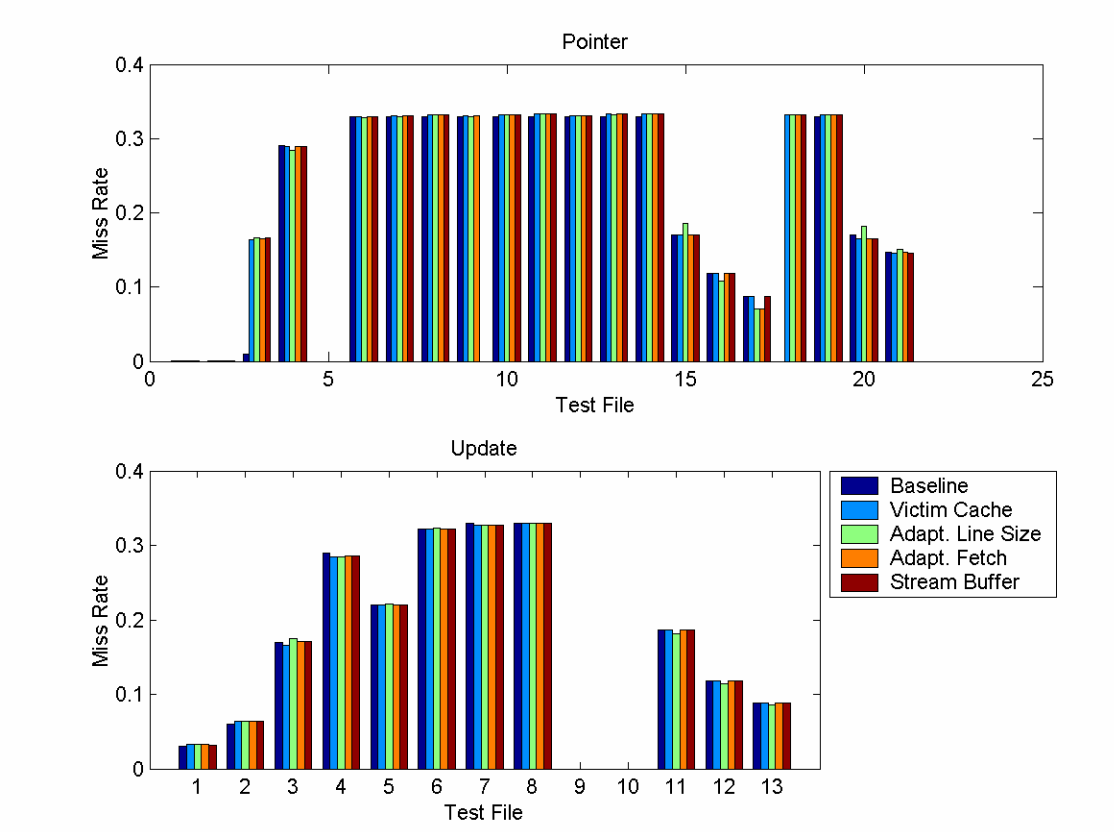
The AMRM team ran tests for most of the stressmarks, using the Sim-Outorder simulator derived from *Simple-Scalar*[61] with modified adaptive memory modules. The Simple-Scalar architecture was based on the MIPS-IV ISA[62].

The team generally did not simulate complete operation of each benchmark. Instead, operation was ceased after 3G operations. Rather than generate the initial data (which is exempt from timing considerations for stressmarks) separately, the team utilized a *fast-forward* simulation parameter to skip the initialization phase and begin metric collection at the kernel. So, 3G operations are performed exclusive of the initialization phase.

The only metric reported was cache miss rate. The baseline cache configuration included:

- L1: 32KB, 32B line, 2-way associative, 1-cycle latency, and least-recently-used replacement policy.
- L2: 1MB (unified data and instructions), 64B line, 2-way associative, 8-cycle latency, and least-recently-used replacement policy.

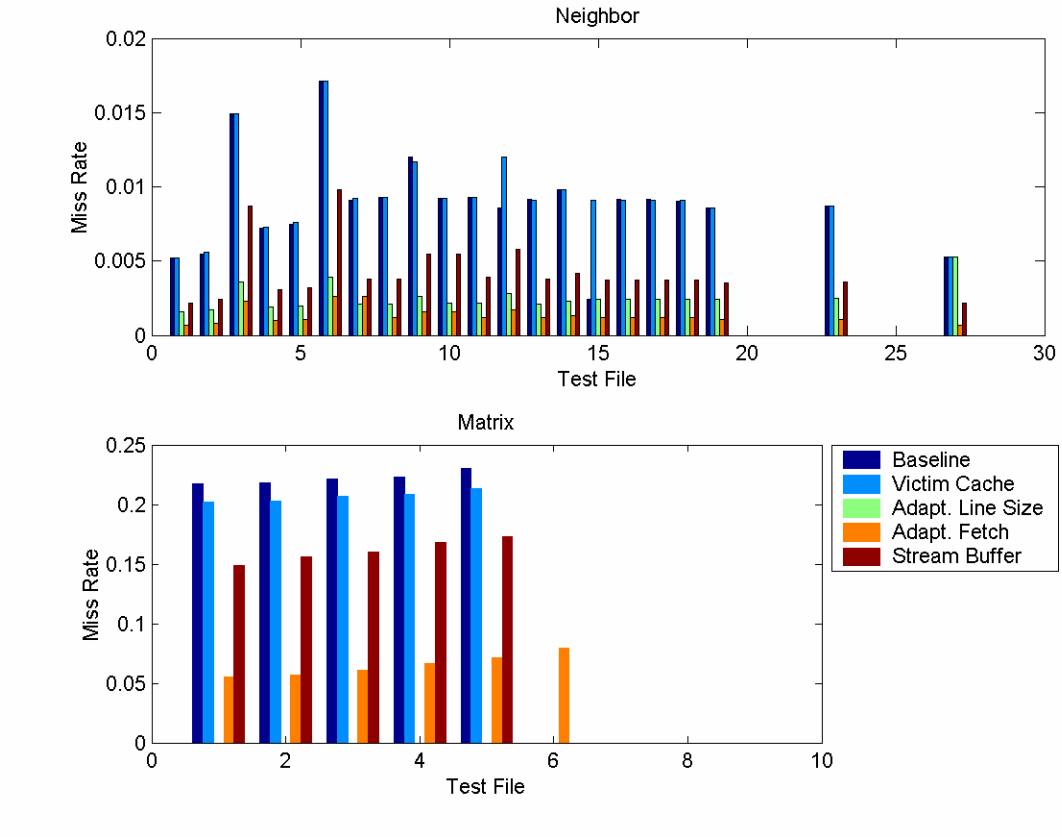
The following three graphs show the cache miss rates reported for each of the four configurations on the stressmarks.

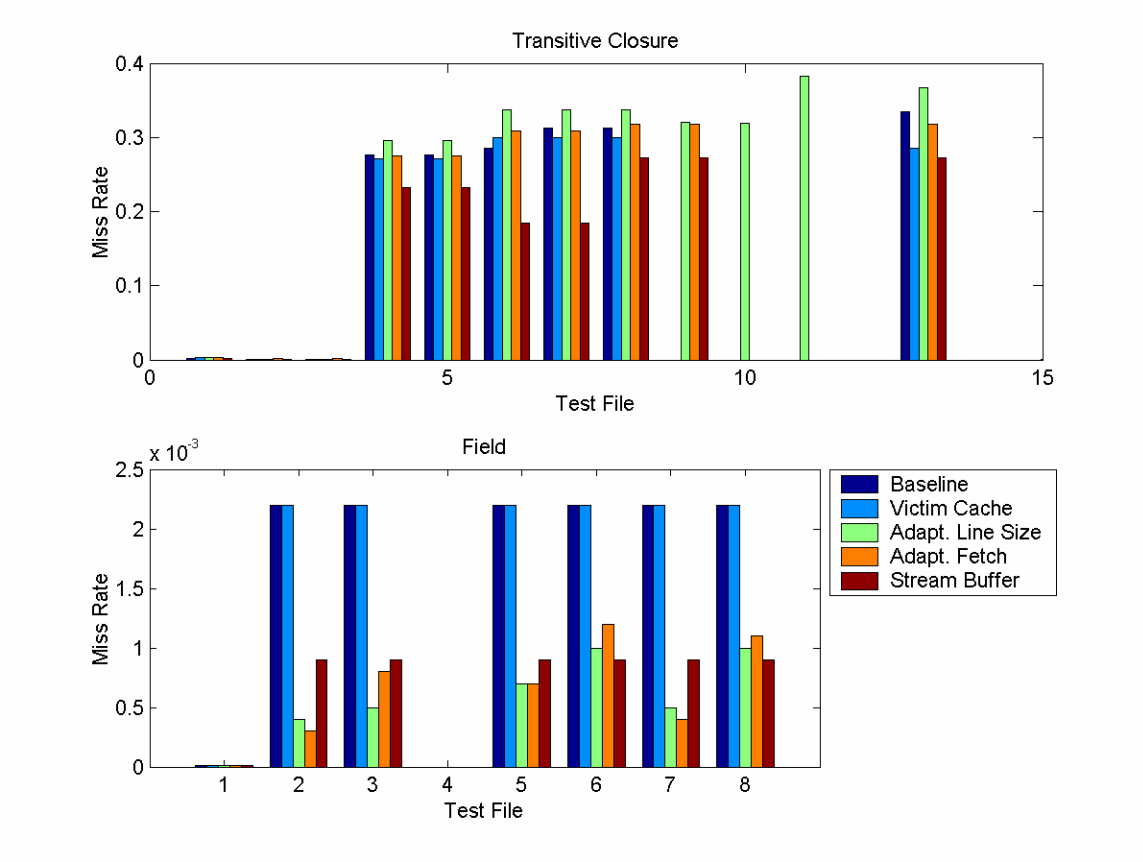


[61] D. C. Burger and T. M. Austin, *The SimpleScalar Tool Set, Version 2.0*, Computer Architecture News, 25 (3), June 1997.

[62] C. Price, *MIPS IV Instruction Set, revision 3.1*, MIPS Technologies Inc., January 1995.

For the Pointer and Update stressmarks, no real gain or loss is shown by any configuration. The access pattern for these stressmarks is essentially random, and the AMRM approaches do not address this pattern. Tests 15, 16, and 17 of the Pointer series, and 11, 12, and 13 of the Update series were window tests, where multiple words are accessed at each pointer destination address. This explains the lower miss rates for these tests. The tests with lower indices generally had smaller address ranges, again resulting in lower miss rates due to fewer capacity misses and greater locality.





12.3 Efficiency

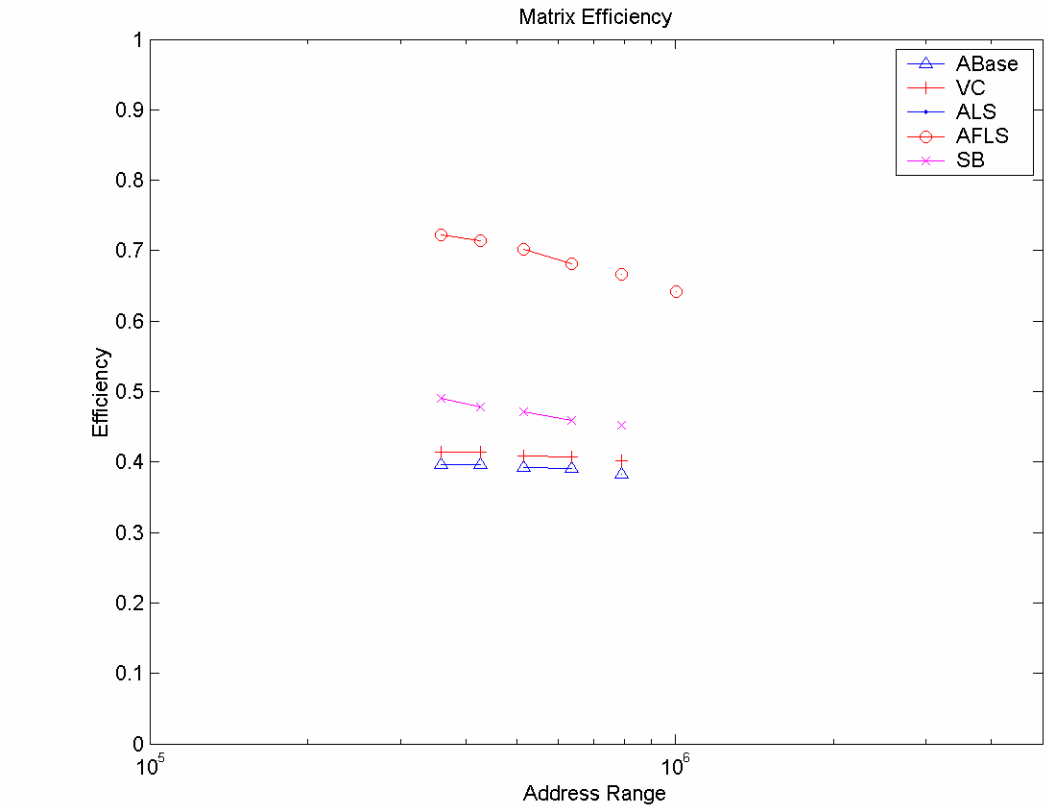
Based on the latencies of the cache hierarchy, we can calculate the efficiency of the memory system according to the following equations:

$$eff = \frac{L_{hit}(misses + hits)}{L_{miss}\,misses + L_{hit}\,hits}, R = \frac{misses}{misses + hits} \therefore eff = \frac{L_{hit}(R + (1 - R))}{L_{miss}\,R + L_{hit}(1 - R)},$$

where R is the miss ratio and L is the access latency. Elimination of all misses gives an efficiency of one. A reduction in L_{hit} is not considered in the context of AMRM, since the project goals assumed certain basic architectural elements such as a single cache hierarchy for each processor. The AMRM team generally utilized a value of 8 for L_{miss}; that value was used for this study, as well. Obviously, the efficiency gains reported here would increase with an increase in L_{miss}.

12.4 Matrix

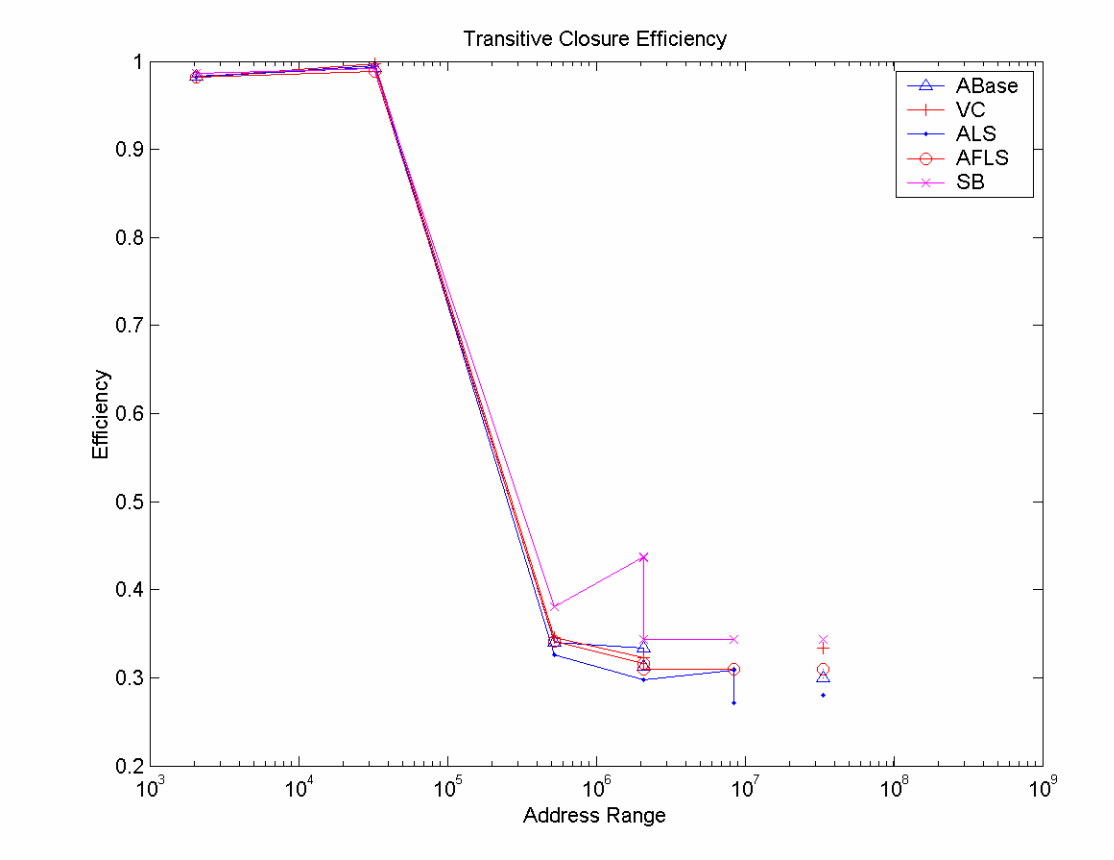
Calculated as described above, the efficiency of the configurations against address range is shown in the following graph.



Not many Matrix tests were run, owing to the long time required for simulation of this stressmark. Although the Adaptive Fetch Line Size configuration yields dramatically higher efficiency than the others, significant decline is seen with address range. It is unknown how the configurations would react to larger problems of this type.

12.5 Transitive Closure

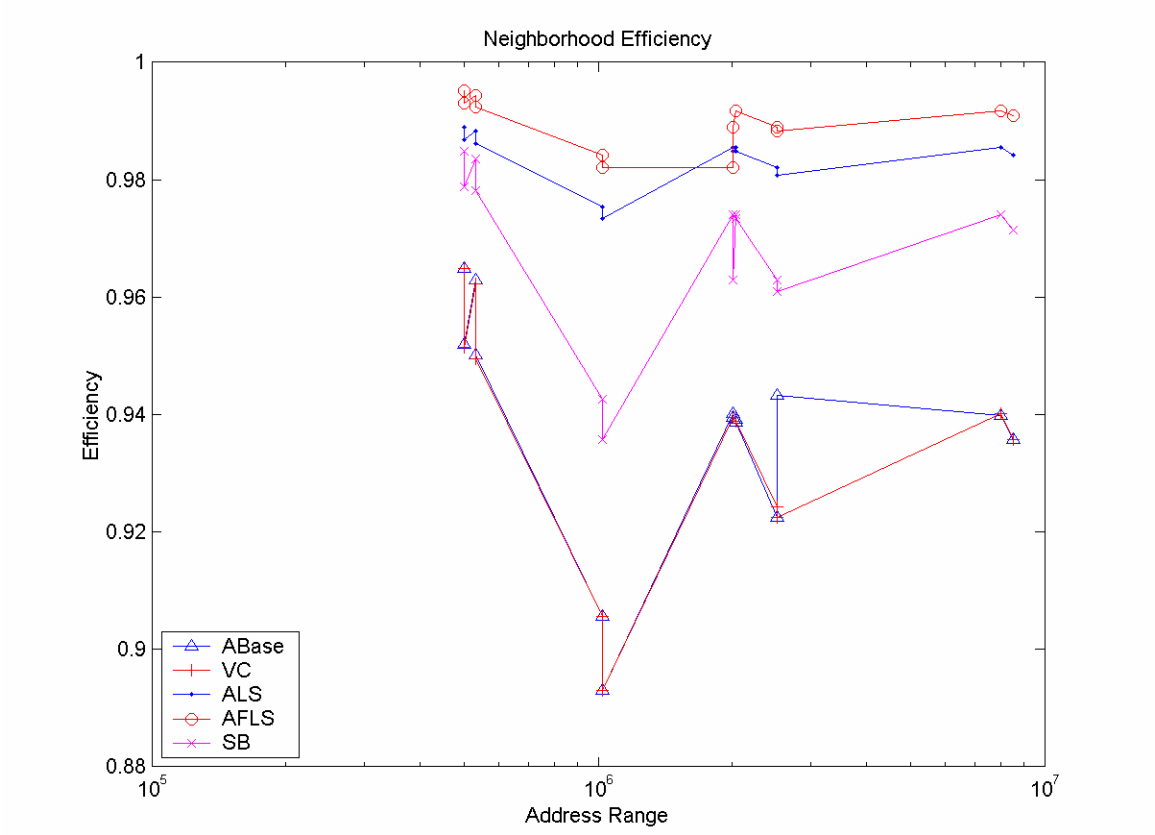
The efficiencies of the configurations for Transitive Closure tests are shown below.



Most obvious is the dramatic drop in efficiency once the problem becomes too large to fit mostly in cache. The Stream Buffer configuration gives improved results—as expected for this fairly regular stressmark—but not with great consistency. Since the larger address ranges correspond to more cycles-to-completion for this stressmark, small efficiency gains can net significant performance gains.

12.6 Neighborhood

The efficiencies for Neighborhood are shown against address range below. Note that even the baseline configuration shows high efficiencies for the small problem sizes tested. Many of the accesses for this stressmark are in unit-stride. Presumably, the histogramming portion of the code generates the bulk of the misses for these prefetch-enabled configurations.



12.7 Programming

The AMRM approach demands no source-level changes to code. Legacy code is therefore supported, and any gains possible with the system are equally available to legacy code and newly developed code.

12.8 Remarks

The AMRM team utilized a sampling simulator for its tests. We are unable to determine the accuracy of the simulation results, but we assume that they are sufficient for comparisons between the adaptation strategies.

AMRM demonstrated the ability to build adaptivity into the memory hierarchy without reconfigurable circuitry. The team claims that these adaptive mechanisms can simplify the memory architecture. For instance, use of an adaptive line-size in the cache can potentially replace various architectural mechanisms such as unit-stride prefetching, victim cache, cache bypass, and split-access type caches.

None of the approaches tested by the AMRM team offered any gain for the Pointer or Update stressmarks. We assume this is a fair indication that GUPS performance would not be improved, either.

The DIS stressmarks probably do not fully expose the value of the Victim Cache approach, since they exhibit relatively few conflict misses. This is an unfortunate product of the needs of the stressmarks, which were developed in part for ease of use and implementation. Presumably, the DIS benchmarks would have been a more appropriate test vehicle for the VC approach.

The other approaches did show efficiency gains, up to about 1.75 for the Matrix stressmark. Adaptive Fetch Line Cache, especially, gave good results. The Stream Buffer approach was strong for the more regular Transitive Closure stressmark.

The efficiency gains shown would theoretically increase in magnitude if L_{miss} were to increase. By all indications, for conventional systems, L_{miss} will continue to grow in the near future.

13 Advisor

The *Algorithms for Data Intensive Applications on Intelligent and Smart Memories* (ADVISOR) project[63] developed an algorithmic framework to enable effective and efficient mapping of data intensive applications onto smart-memory architectures. With these techniques, the value of the higher bandwidth and lower latency offered by advanced architectures is maximized.

13.1 Description

The Advisor project resulted in various algorithmic techniques for use with architectures developed under the DIS program. Bandwidth-aware algorithms are expected to result in superior performance compared with straightforward mapping, which does not consider the costs involved in on-chip and off-chip memory access. Memory performance can be improved by matching the application data access patterns to the data arrangement in memory. The following gives an overview of the reported techniques:

- *Application Level Memory Access Optimization:* For most applications, cache and TLB behavior have a significant effect on performance. 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. However, when used in conjunction with row-major or column-major layouts, tiling does not eliminate cache conflict misses. Tiling of matrix computations also does not address page locality in the TLB. The Advisor team described a cache-conscious data layout for matrix structures, involving a contiguous layout of the data elements within a tile. Use of block data layout for tiled computations achieves a significant reduction in TLB misses, and helps reduce self-interference misses for large matrices.

- *Cache-conscious data layout based on Perfect Latin Squares:* The Advisor team used Perfect Latin Squares as a mathematical framework for data distribution among parallel memory banks to minimize memory bank conflicts. The team applied these methods to uniprocessor memory hierarchies to minimize cache conflicts. The mapping allows conflict-free access to rows, columns, main diagonals, and minor sub-squares of square matrices. The team claimed an up-to-$O(N^2)$ reduction in cache conflicts for column access, compared to standard row-major layout.

- *Data Layout Optimizations for the Transitive Closure Stressmark* : The team developed the *Unidirectional Space Time Representation* (USTR) to uniquely address the complexities of transitive closure. Measurements relating to this method are given in this section.

- *Cache-Friendly Graph Representation:* The team applied a cache-friendly graph representation developed for Dijkstra's algorithm. The optimization combines the superior size of an adjacency list with the regular access pattern of an adjacency matrix, resulting in an up-to-2x performance improvement.

- *Tiling:* The basic goal of tiling is to reduce the work set size so that the problem will fit into the cache. By reordering the smaller problems or tiles, data dependencies can be satisfied. The team reported an up-to-10x improvement for the Floyd-Warshall algorithm.

[63] http://advisor.usc.edu

- *Graph Matching:* For dependencies that require possibly examining the entire graph during each step of the computation, the graph can be partitioned into sub-graphs that fit into the cache. By doing some local computations, the total work required can be reduced. The Advisor team reported that experimental results show performance improvements as highly dependent on the density of the graph, averaging between 2x and 6x.

13.2 Measurements

Advisor only tested one DIS stressmark: the Transitive Closure stressmark. The team ran a complete series for each of seven algorithms (a baseline, plus six alternatives) on each of four hardware platforms.

13.2.1 Algorithms

The algorithms are introduced in the following table.

Algorithm[64]	
Baseline *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.
FW TC *Floyd-Warshall with Tiling and Copying*	Tiling with copying is a standard cache-friendly optimization that can be performed using current research compilers. Because of this, tiling with copying are applied to the Floyd-Warshall algorithm. Due to data dependences current research compilers can only tile the inner two loops.
FW USC *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.
DJK Heap *Basic Dijkstra's*	The baseline Dijkstra's algorithm for the all pairs shortest path problem. Again, the best compiler optimizations available are utilized. A binary heap is used to implement the priority queue and store the graph as an adjacency list.
DJK CF *Cache-Friendly Dijkstra's*	In order to match the data access pattern of Dijkstra's algorithm to the data layout, the adjacency list is replaced with the adjacency matrix, and the binary heap is replaced with an array. A linear search is employed to find the minimum value. In this way, advantage is taken of data reuse at the cache line level, and prefetch is simplified.
SA BDL *Simple USTR Floyd-Warshall*	The basic Unidirectional Space Time Representation (USTR) developed by the Advisor team. This incarnation uses a systolic array implementation of the Floyd-Warshall algorithm.

[64] More detail can be found in M. Penner and V. K. Prasanna, *Cache-Friendly Implementations of Transitive Closure*, Proceedings of International Conference on Parallel Architectures and Compiler Techniques, Barcelona, Spain, September 2001. Also see J. S. Park, M. Penner, and V. K. Prasanna, *Optimizing Graph Algorithms for Improved Cache Performance*, Proceedings of International Parallel and Distributed Processing Symposium, Fort Lauderdale, Florida, April 2002.

FW USTR *Optimized USTR Floyd-Warshall*	A tiled implementation of the Floyd-Warshall algorithm, which also fits in the USTR. In order to eliminate the three passes present in the simple USTR implementation, the computation of tiles is reordered. A recursive iteration of this implementation was also tested. Only partial results were presented. See the notes regarding the data below.

13.2.2 Platforms

The platforms utilized are described in the following table.

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

13.2.3 Data Notes

Each implementation was compiled using the highest level of optimization available in gcc.

Times were collected using the system time function. Timer resolution was one microsecond for all machines except the Pentium III, for which it was one millisecond.

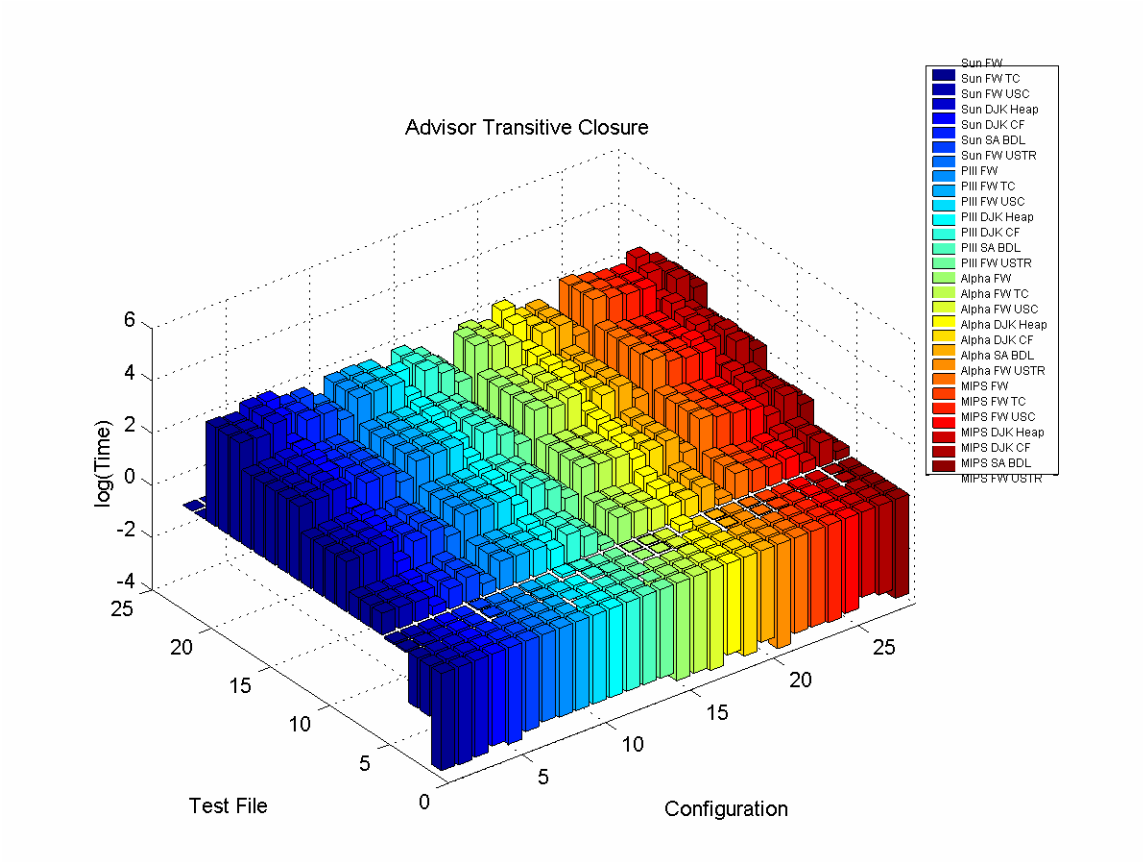
For our purposes, the results of the recursive implementation of the Floyd-Warshall algorithm and the optimized FW USTR were very similar. The results for the recursive FW USTR were not gathered for the MIPS machine. The results for the optimized FW USTR on the Alpha machine had a granularity of one second. Therefore, the recursive FW USTR times are substituted for the optimized FW USTR for the Alpha machine.

The MIPS machine shows unexpected variation in the results due to a high concurrent user load.

A total of 9 of the measurements were observed as irregular and removed or, in the case of obvious typographical errors, altered.

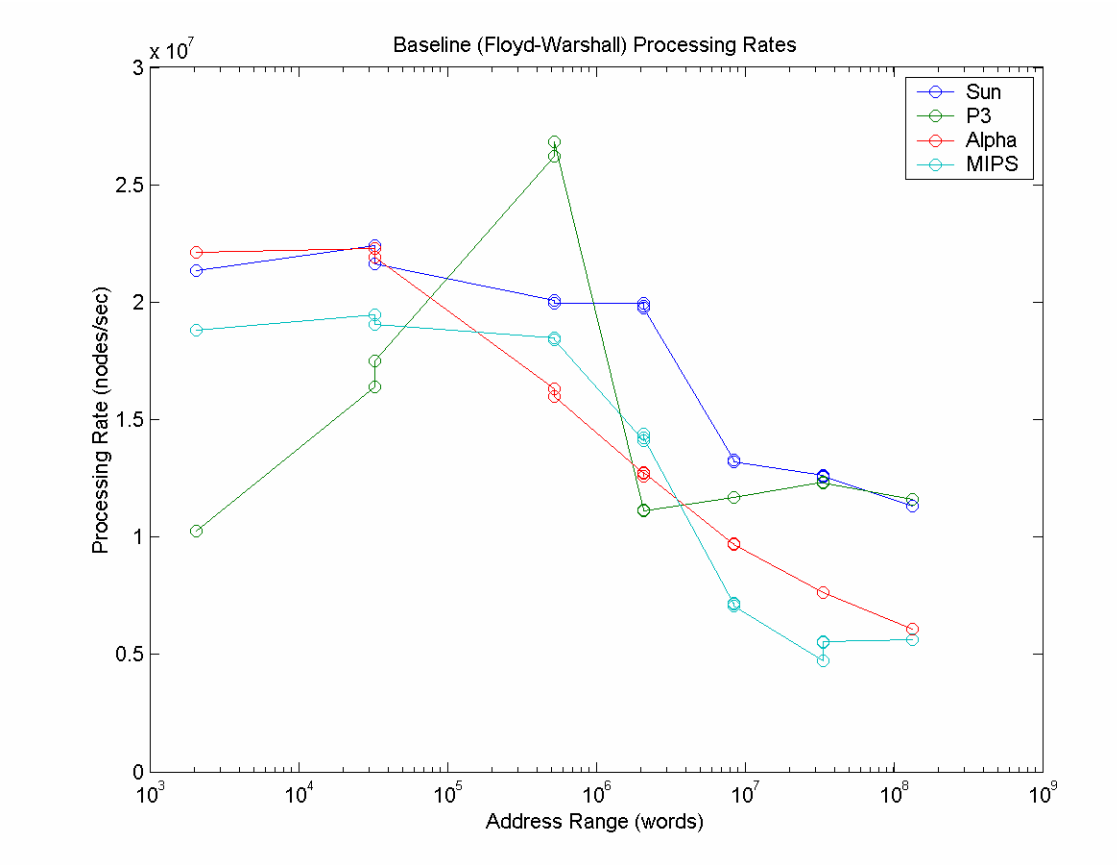
13.3 Transitive Closure

The chart below gives an overview of the results. Each time is shown as one bar. Each color represents a different configuration (i.e., combination of algorithm and platform). Problem sizes generally increase with test file number.



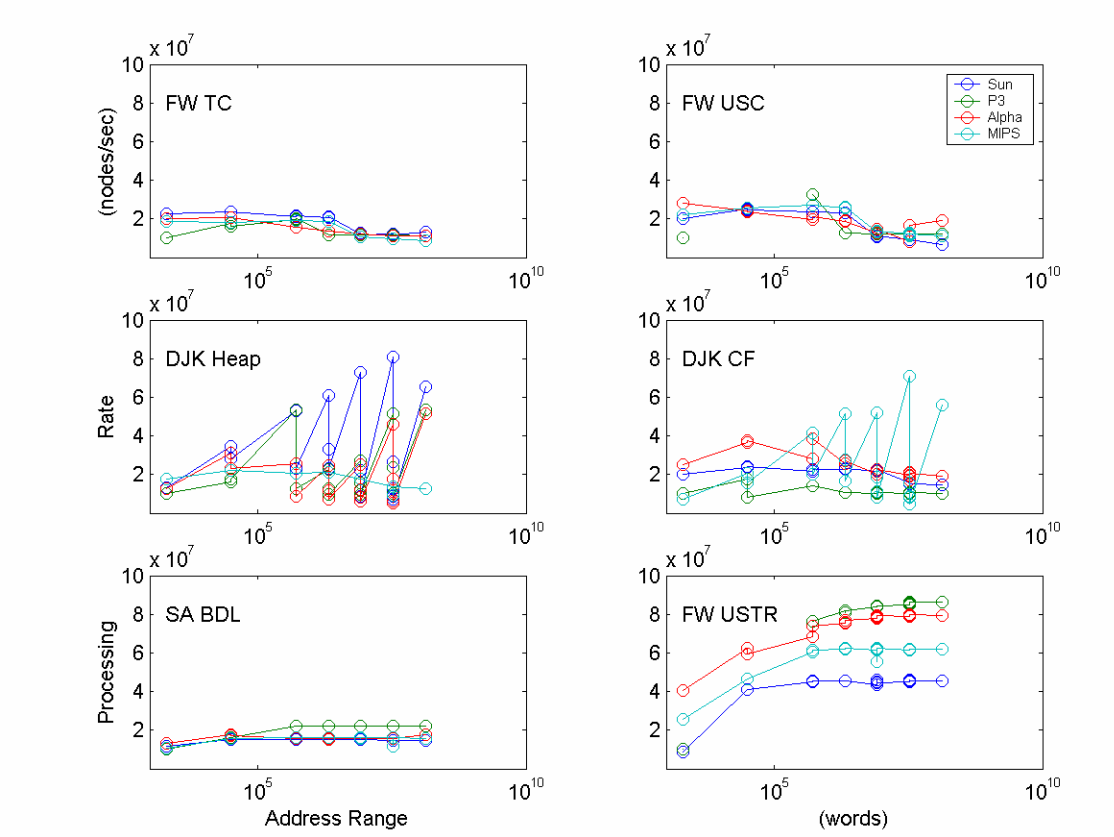
13.3.1 Baseline Processing Rates

This graph shows the baseline processing rates, graphed against address range. Note that all platforms show a decline when the problem no longer fits within nearby cache. The Pentium machine particularly exhibits a strong peak, suggesting a severe processor-memory mismatch.



13.3.2 Processing Rates

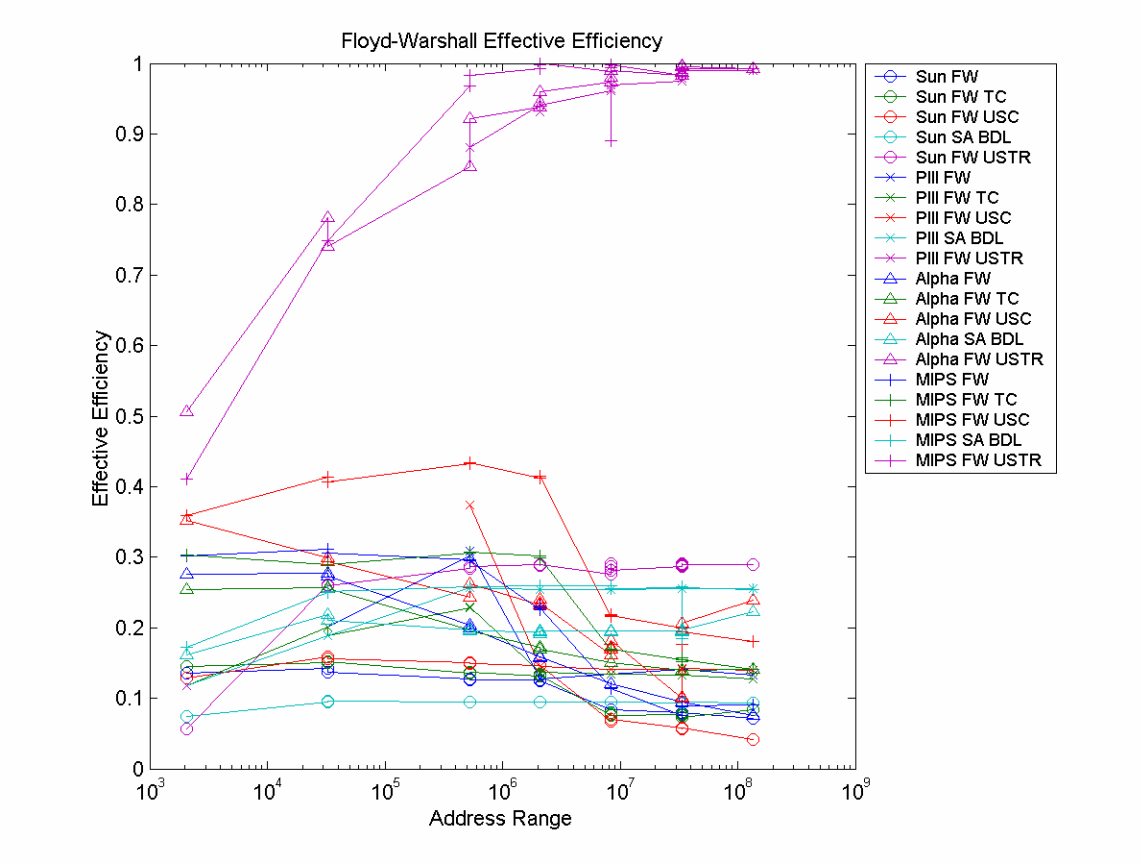
The graph below gives the processing rates for each of the algorithms under test. Each subgraph shows one algorithm; the four series represent the different platforms.



The FW TC version shows a performance envelope similar to that of the baseline. The FW USC version gives better performance, though a drop for larger problems is still evident. The two DJK approaches have dramatically different performance envelopes here because there is a performance dependency on the number of edges in the graph; the FW variants are mostly dependent on the number of nodes. Note that the MIPS machine reacts uniquely to the two DJK approaches, in that its performance pattern is contrary to the other three machines. Finally, both of the USTR representations deliver processing rates that do not drop off as cache is exceeded. The optimized (FW USTR) version does dramatically better than the simple (SA BDL) version, though both suggest improved efficiencies by expanding the performance envelope.

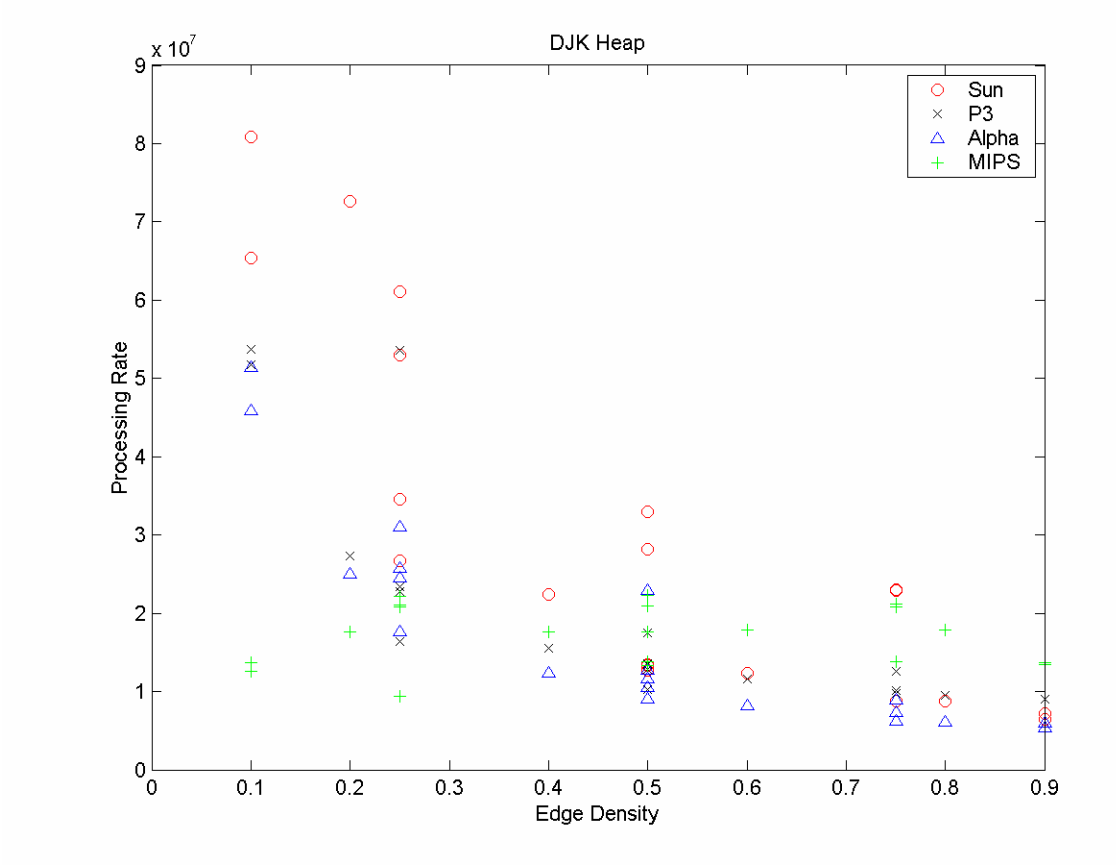
13.3.3 FW Efficiency

It is not strictly valid to calculate common efficiencies when the algorithms differ. However, although the variants of the Floyd-Warshall algorithms do not utilize the same paths to solution, the primary variation is in how data is arranged. The reader is cautioned to keep this in mind when viewing the following graph, which gives the efficiencies of the algorithms, based on the peak observed processing rates of each platform.



13.3.4 DJK Value

The graph below gives the processing rate for the DJK Heap configurations. Observe that processing rate declines as edge density rises. For large, sparsely-connected graphs, this approach could give better performance for a given architecture.



13.4 Programming

The Advisor team utilized off-the-shelf hardware for these tests. The programmability of the machines in general is not a question to be addressed here.

The optimizations described here were all developed and tested by hand. Although the net changes to the source code were relatively slight, a great deal of human labor was expended in applying the concepts to the Transitive Closure stressmark. The Advisor team outlined methods for automating selection of certain parameters within the scope of this problem, but the general use of these techniques would not be simple given a new problem to solve.

13.5 Remarks

The Advisor project demonstrated performance enhancements available for those in a position to develop better software, with awareness of memory hierarchy costs. Developers can maximize the performance of their algorithms by increasing data reuse, decreasing cache conflicts, and decreasing cache pollution wherever possible. The project found several data layout and data access optimizations for the Transitive Closure stressmark, resulting in performance gains of up to 10.0 for the PIII architecture, and excellent efficiency curves. Tests were performed using off-the-shelf platforms under realistic circumstances. The team claims the same techniques can benefit a large class of algorithms, though we do not find that this would be readily demonstrable.

This project explored the potential of algorithmic enhancements, which do not require modification of hardware or operating system. Still, the cost of use of the new representations is very high, since they are labor intensive, and problem-dependent. While the tests on the one stressmark were comprehensive, only one was tested, and to do others would require significant labor.

14 Algorithmic Strategies for Compiler-Controlled Caches

The *Algorithmic Strategies for Compiler Controlled Caches* (ASC3) project[65] developed methods for managing programmable caches to overcome the performance hurdles germane to data-intensive applications. Hardware-based smart cache management was implemented and validated via simulation and emulation within the HP-Intel IA-64 program. The simulated devices monitored memory access patterns and automatically customized the cache management strategy to the application's needs.

14.1 Description

The ASC3 project concerned itself with application analysis and subsequent synthesis of program-specific memory management strategies. The analysis, or *cartography*, employs aggressive techniques drawn from diverse domains (e.g. pattern matching, computational learning, randomized sampling, and on-line algorithms) for discerning hidden data reference patterns from profiles and maps. Cartography is geared to characterize the behavior of an application as *quasi-regular*, *quasi-irregular,* or *irregular*. The characterization guides the synthesis of custom memory management strategies tailored for an application. The proposed range of solutions varies from completely static approaches for quasi-regular patterns, to complete run-time support for irregular applications.

Results were reported for two types of optimizations:

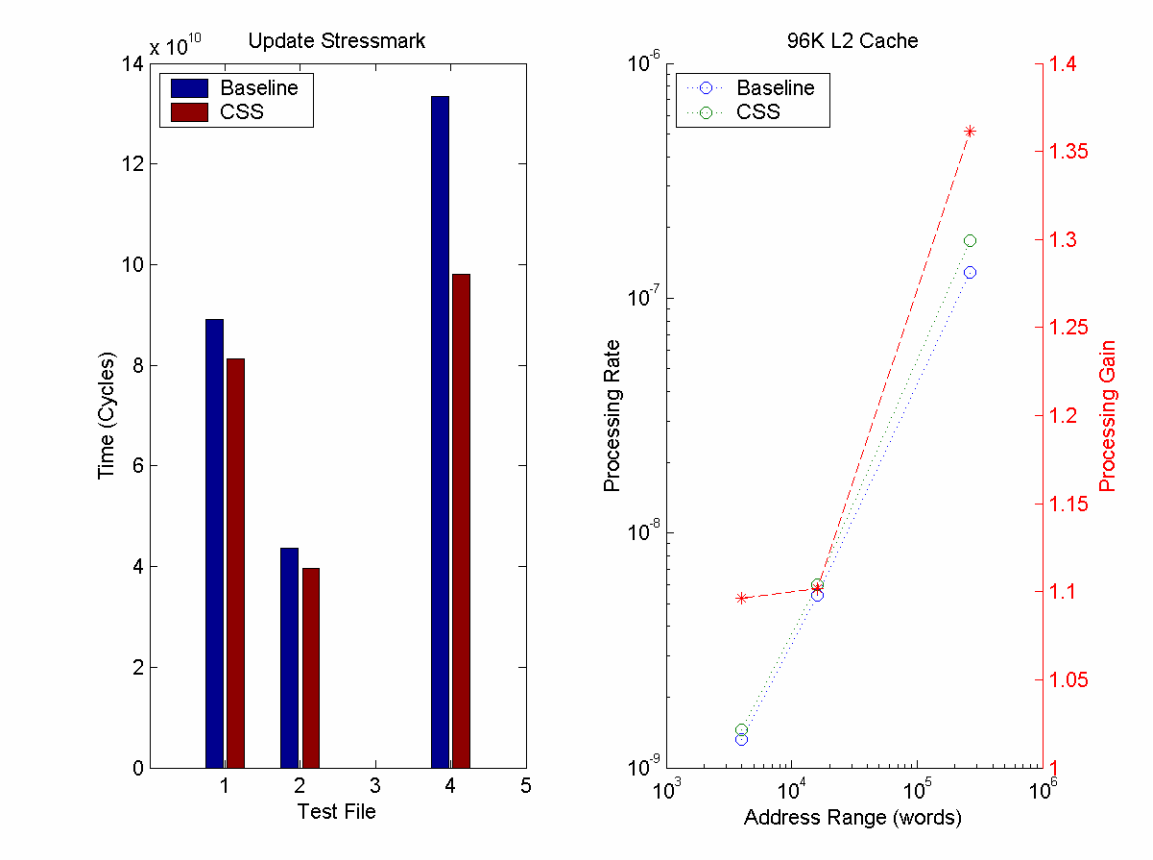
- Cache-Sensitive Scheduling. These results were given in the form of cycles-to-completion on a Trimaran-simulated machine with the following cache parameters:
  - cache #0 (based on IA-64). L1: I-16KB, D-16KB, 4-way set-associative, 32-byte lines. L2: 96KB, 6-way set-associative, 64-byte lines. The tests with the Update stressmark below utilized this configuration.
  - cache #1 (based on PIII). L1: I-64KB, D-64KB, 4-way set-associative, 32-byte lines. L2: 8MB, 8-way set-associative, 64-byte lines. The tests on the Matrix and Transitive Closure stressmarks below were based on this configuration. [66]
- Data Remapping. These results were given in the form of an execution speed-up factor. Gcc was used, with –O or –O3 flags, on three different processors:
  - 750MHz Pentium III with 256Kb L2;
  - 400MHz Pentium II with 512Kb L2; and
  - 400MhZ UltraSparc II with 2048Kb L2.

[65] http://www.crest.gatech.edu

[66] Conflicting information was reported about the configuration. The information supplied with the recorded data declared a 64KB L1 and an 8MB L2, while a separate summary report claimed a 16KB L1 and a 512KB L2.

14.1.1 Update

Results for Cache-Sensitive Scheduling of the Update stressmark are shown here. The bar chart shows raw time information, while the graph gives processing rates and gains (over baseline) versus address range.

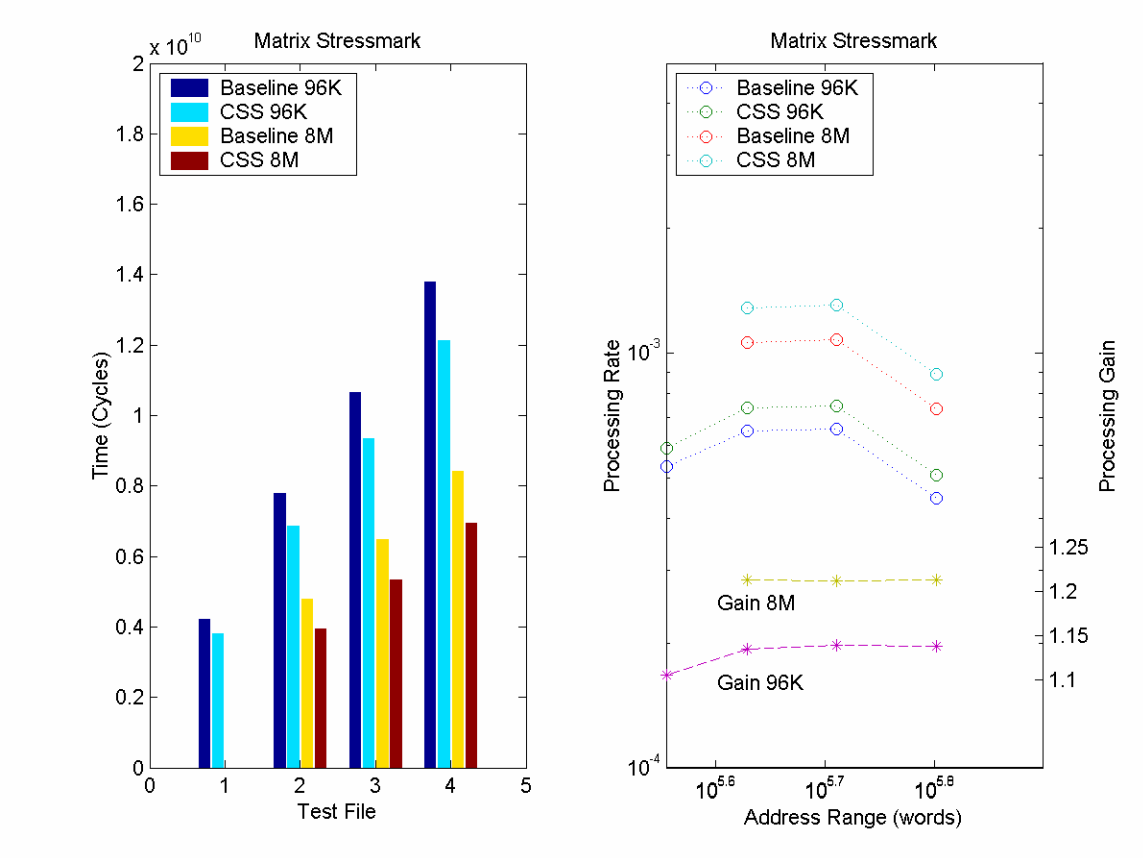


Note the very narrow range of addresses; the team tested small problems. Although the 96KB simulated L2 cache was likewise small, it could nearly or fully contain the entire problem for each of these tests. That the results indicate a 10% improvement for a problem fitting entirely into L1 is impressive. This stressmark has a high rate of compulsory misses for the small problems, and hiding of these misses is difficult due to the random nature of the problem. The performance gain climbed to about 37% when the problem fit mostly in L2.

It would be dangerous to interpolate performance for larger problems, given that the effects of CSS are not demonstrated here for problems that substantially do not fit within L2.

14.1.2 Matrix Stressmark

The graph below shows performance figures for Cache-Sensitive Scheduling of the Matrix stressmark. The bar chart shows raw time information, while the graph gives processing rates and gains (over baseline) versus address range. 96K and 8M refer to the number of bytes in the L2 cache.



Again, the address range is small; all problems tested would easily fit within the 8MB L2 cache. The gains shown could, therefore, have much to do with avoiding the penalties of L1 misses.

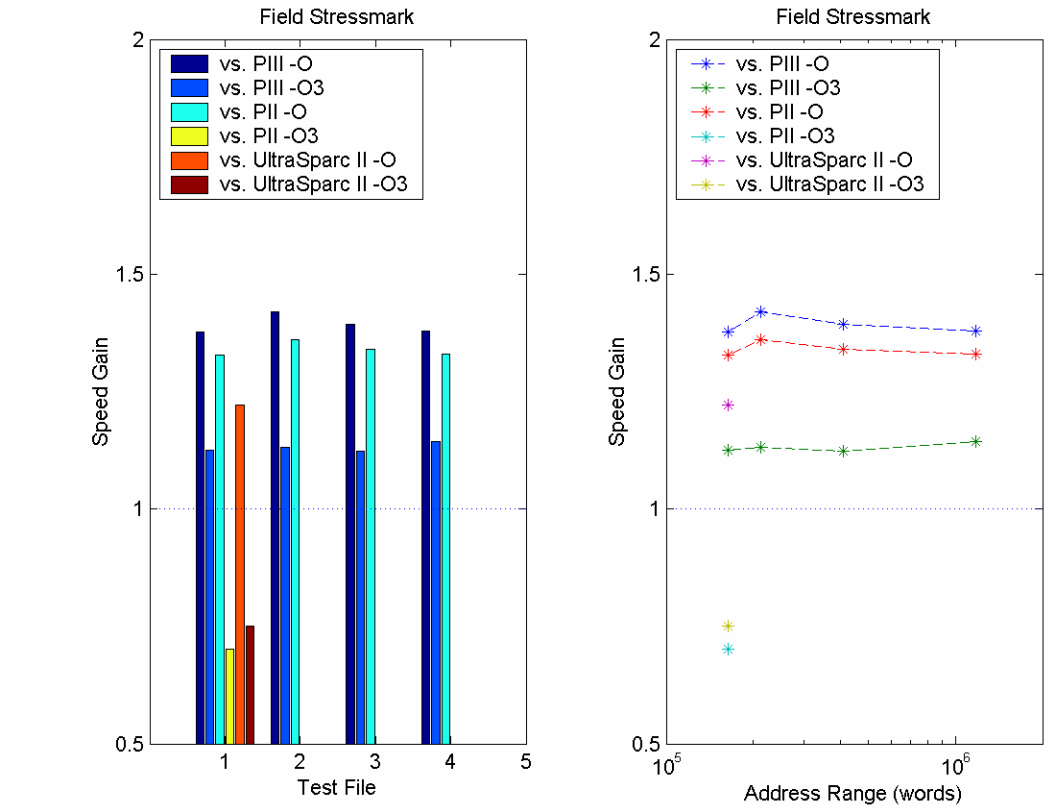
The processing gains shown are constant, even as the rates decline for increasing problem sizes. It is unknown whether these gains could be realized for very large problems, but the possibility is suggested since the gains for the 96K configuration do not drop for the larger tested problems.

14.1.3 Transitive Closure

The ASC3 team gave an isolated result for Transitive Closure. For test tc06, the team recorded 91.7Gcycles for the 8M L2 baseline configuration, and 89.9Gcycles for the CSS configuration. This represents a speed increase of about 2%.

14.1.4 Field Stressmark

For the Field stressmark, the ASC3 team applied Data Remapping. The recorded processing rate gains are shown below.



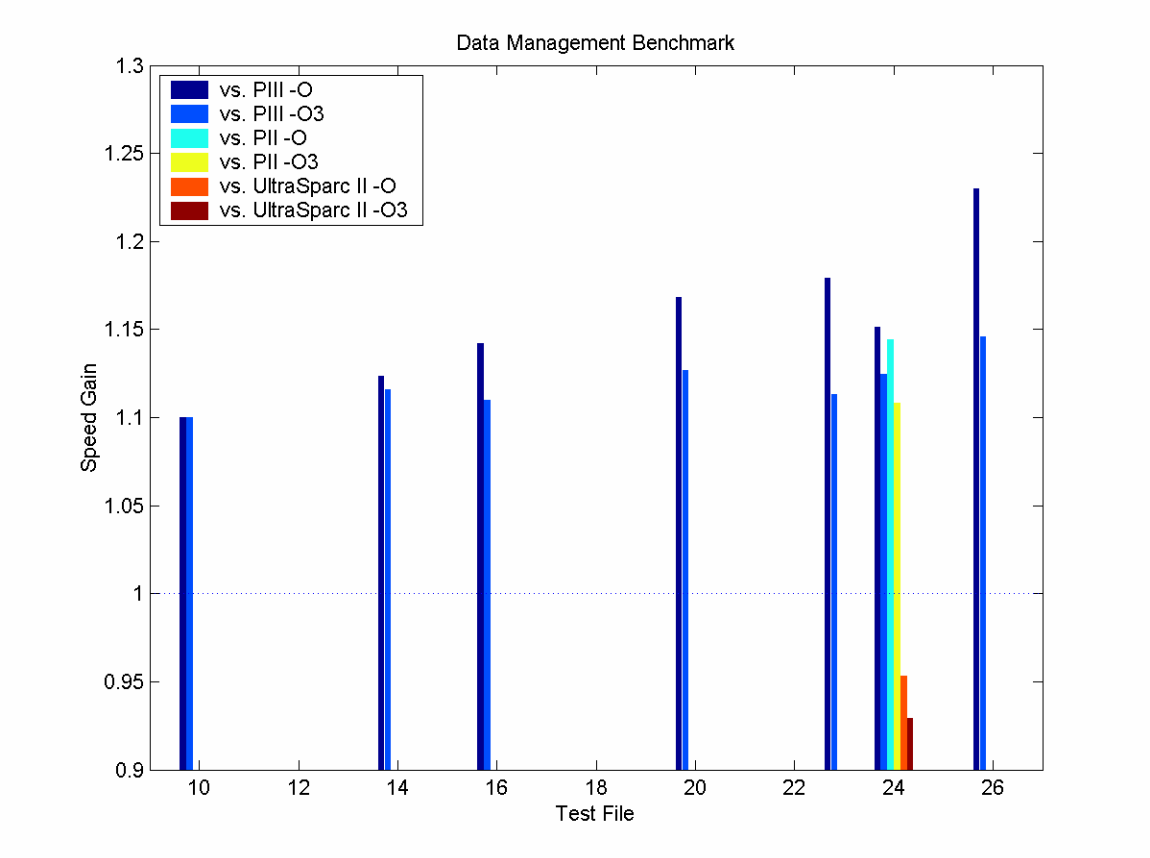
As with the previous three stressmarks, the tests shown here involved a small problem size. The largest cases here would not fully fit within the tested L2 caches, but the other cases would. Additionally, this stressmark is extremely regular, and prefetching tends to hide any off-cache latency problems.

It is interesting that the ASC3 team was able to demonstrate even modest speed gains for this highly regular stress-mark without hardware modifications. However, the better gains are shown when compared with a low level of compiler optimization. When using –O3, little gain is offered, and only for the PIII system. There is a performance loss compared to the UltraSparc –O3 and PII –O3 configurations. It seems likely that this is reflective of the better processor-memory harmony for those systems. In other words, this technique, as would be expected, is beneficial when there exist memory hierarchy inefficiencies.

Like the gains for the Matrix stressmark, the gains appear consistently across the (narrow) range of problem sizes tested.

14.1.5 Data Management Benchmark

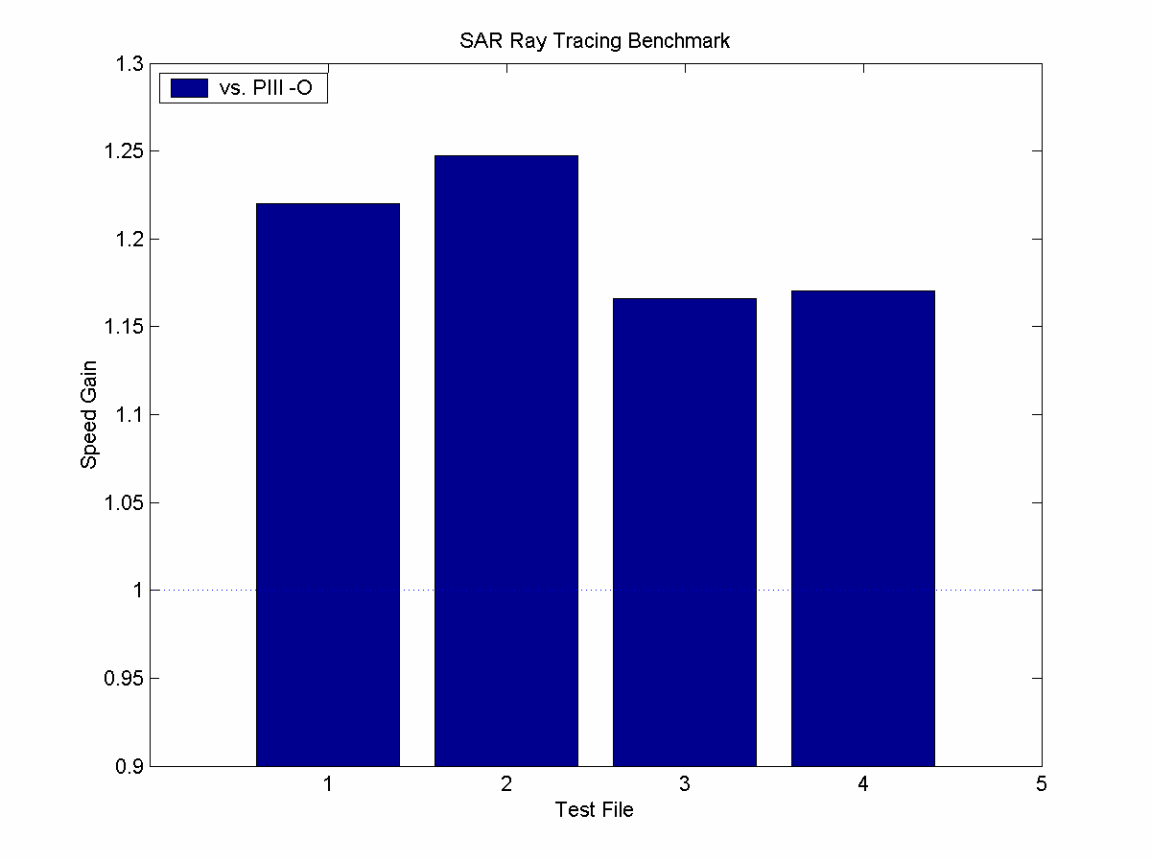
The speed gains reported for Data Remapping of the Data Management Benchmark are shown below.



Only one test was reported using the UltraSparc and PII configurations, but it seems likely that the UltraSparc tests would only show speed losses. The PIII –O configuration is expected to have the worst processor-memory mismatch, so the highest gains offered are not surprising here. For the PIII –O3 configuration, increases of 10 to 15% are seen with fair consistency. Test #26 keeps the largest database on average over time.

14.1.6 SAR Ray-Tracing Benchmark

The speed gains of four Ray-Tracing Benchmark tests utilizing Data Remapping are given below. Again, net increases of 15-20% are seen against what is expected to be the worst of the tested configurations (i.e., the configuration that has the most to gain from these algorithms). No measurements for other configurations were reported, but the other evidence within this section suggests that no noteworthy gains would be available for this benchmark on any of the other configurations, and substantial losses for the UltraSparc –O3 configurations seem likely.



14.2 Programming

The approaches were tested within the scope of finding program optimizations that could be found and implemented by compilers, so special programming techniques are not a consideration.

14.3 Remarks

The two approaches explored under ASC3 gave small (10-20% typical) gains in processing rate with no hardware modification.

Cache-Sensitive Scheduling showed modest gains for the highly irregular Update stressmark, and the semi-regular Matrix stressmark, but negligible gain for Transitive Closure. Most of the test data sets were able to fit within L2. It is unknown what level of gains would be offered by CSS if traditional compiler optimizations were employed as well. If the compiler did not fully optimize the code as a baseline condition, the gains reported here would be significantly larger than those realizable in practice.

The Data Remapping approach was able to show significant gains for the PIII architecture, but those gains were diminished when compared to those offered by current compiler optimizations. The Sparc architecture, which exhibits less of a processor-memory mismatch, saw a reduction in performance.

15 Program Analysis

Taking the collection of DIS benchmark results together, it is desirable to extract information relevant to the whole program. Specifically, it is desirable to know which approaches yielded good results for given problems, so that system designers or users may choose the appropriate technology for their tasks.

The DIS contractors returned results from a subset of the DIS benchmarks and stressmarks. This affected the ability to make comprehensive findings and develop broad conclusions. In the future, studies of performance correlated with problem variables, as well as fair comparison of approaches in various domains would be desirable.

This section attempts to draw together key elements from individual projects to form a simplified image of DIS benchmarking results.

15.1 Demonstrated Stressmark Gains

The tables below attempt to describe the performance gain demonstrated for each stressmark. These graphs, however, are necessarily approximate, because the performance offered in a given instance is subject to myriad circumstances. While interpreting the graphs, pay special attention to the following important notes:

- The graphs show *demonstrated performance gain*, but this value may be defined differently for different projects. In most cases, it relates to reduction of time-to-complete for a given problem. Gain values are approximate, and are derived from individual selected tests, averages of several tests, and estimations.

- The order of the projects as they appear in the graph is significant. It indicates a rough ranking, in our opinion, of the hardware implementation complexity of the system. At the very bottom of the graph are projects requiring software modification only.[67] Above those appear projects modifying small elements (e.g., memory controllers or cache behavior) of systems. Above those are the PIM-based architectures. At the very top are the memory-stream devices. These rankings are subjective, and no effort is made to find the relative distance between individual ranks.

- Each stressmark is divided into *Small Test* and *Large Test* categories. The boundary between the two was generally held to 1Mword, which is drastically smaller than we had hoped. Because of this, the reader is cautioned to understand that *Large Tests* in many cases are still not large at all, and in some cases may even fit within cache of modern systems.

- The vast majority of results presented here are based on simulations. It is very likely that actual measurements would deviate significantly, perhaps even from so-called 'cycle-accurate' simulations. The SimpleScalar simulator, for example, was utilized by several teams. It only superficially models the superscalar CPU, and it is inaccurate: it passes system calls directly to kernel, contains minimal memory timing functionality, gives dubious results, and typically reports radically higher (up to 1 Order) IPC than the actual CPU.[68]

- Of the teams with *in situ* processing approaches, only the Diva team provided data relative to multi-chip operation. The data below for other teams are based on single-chip configurations.

---

[67] Understand that software-only approaches do not necessarily cost less to utilize.

[68] Larry Rudolph, *Malleable Caches* Principal Investigator, during project review, March, 2001.

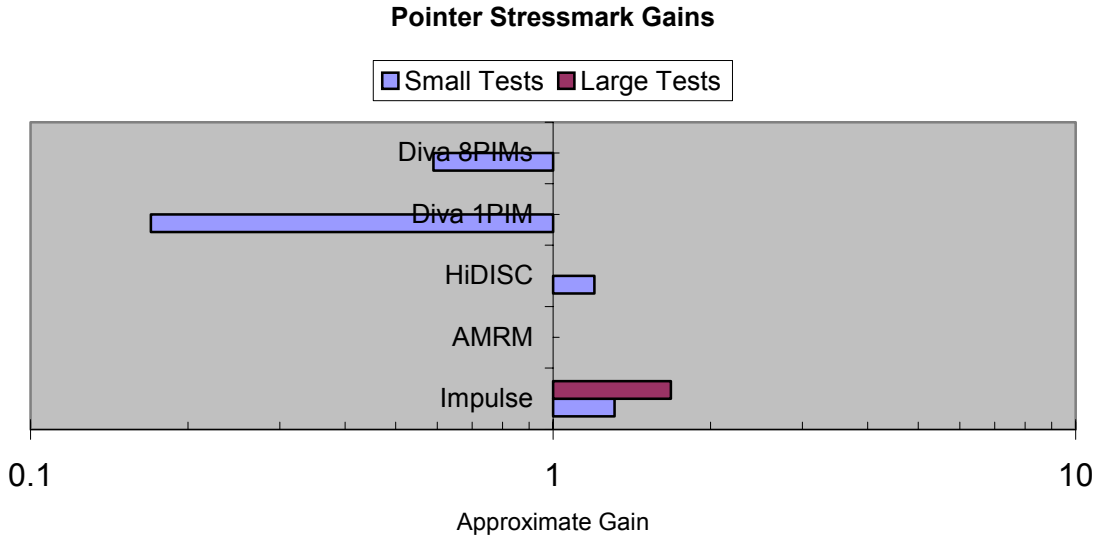
- Where results were not provided for a project, the project name does not appear in the graph. In cases where a project name appears but no bars show, the project did not demonstrate appreciable gain.

- In most cases, the absence of a *Large Test* bar where a *Small Test* bar is present indicates that only small tests were performed.

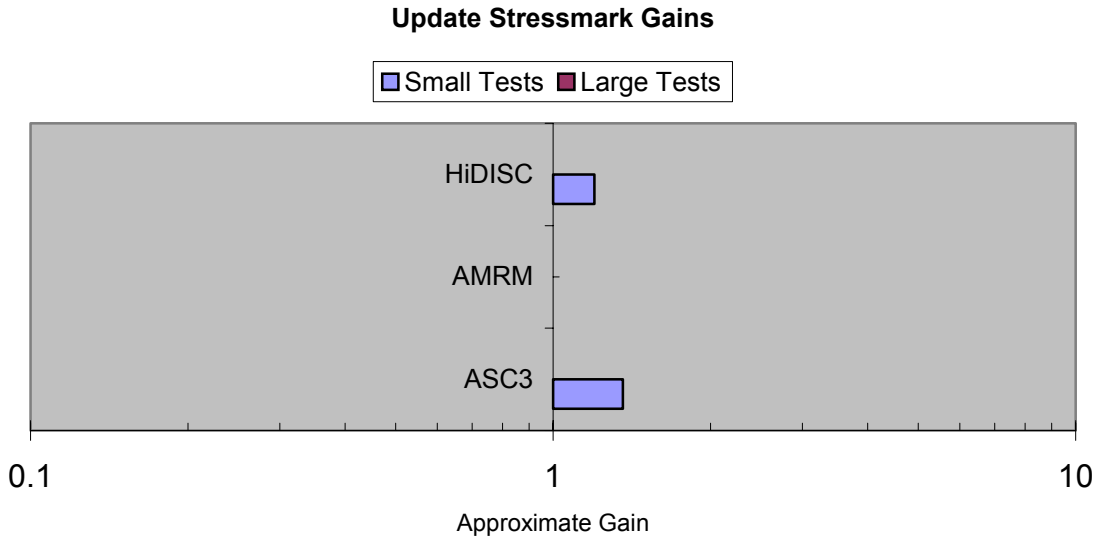
- Very few of the teams supplied any output files for verification of correct operation. It is assumed that the teams validated their own. However, if any data were supplied based on faulty runs, the results shown in this document could be skewed.

- No team declared that it deviated from numerical (data type) standards. It is unknown whether any team instituted more rigorous standards than those required.

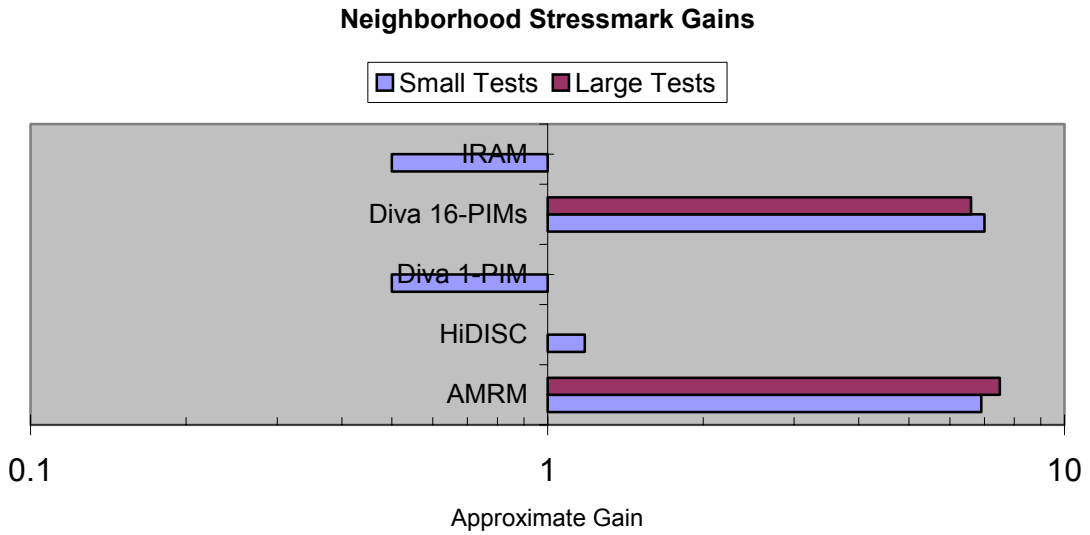
15.1.1 Demonstrated Pointer Stressmark Gains



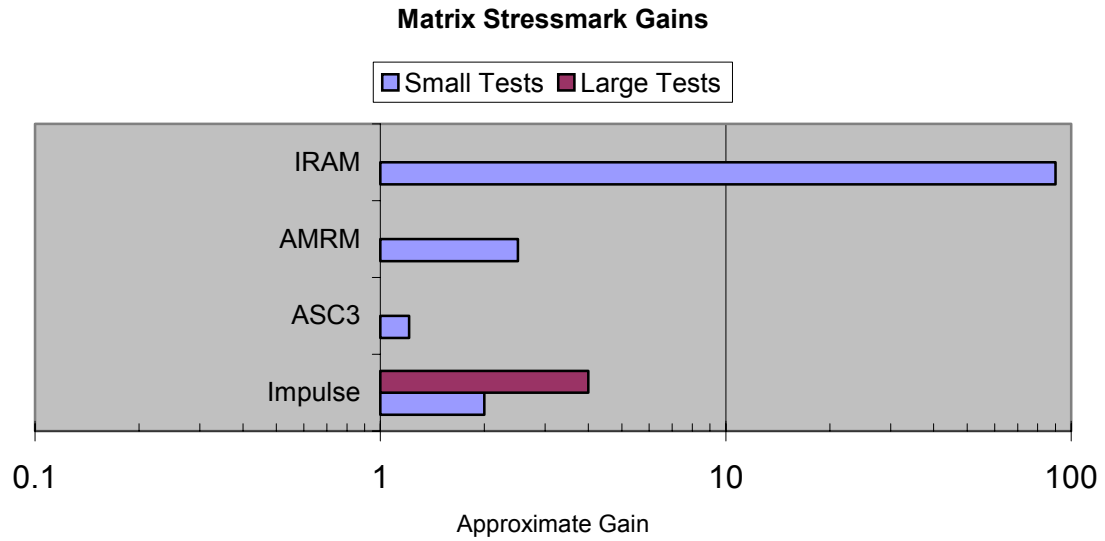
15.1.2 Demonstrated Update Stressmark Gains



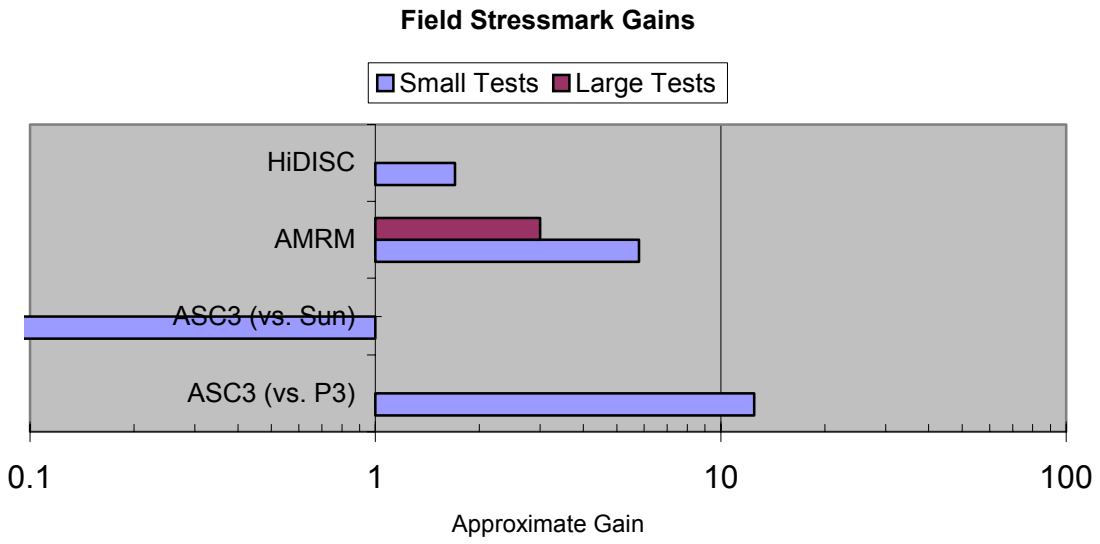
15.1.3 Demonstrated Neighborhood Stressmark Gains



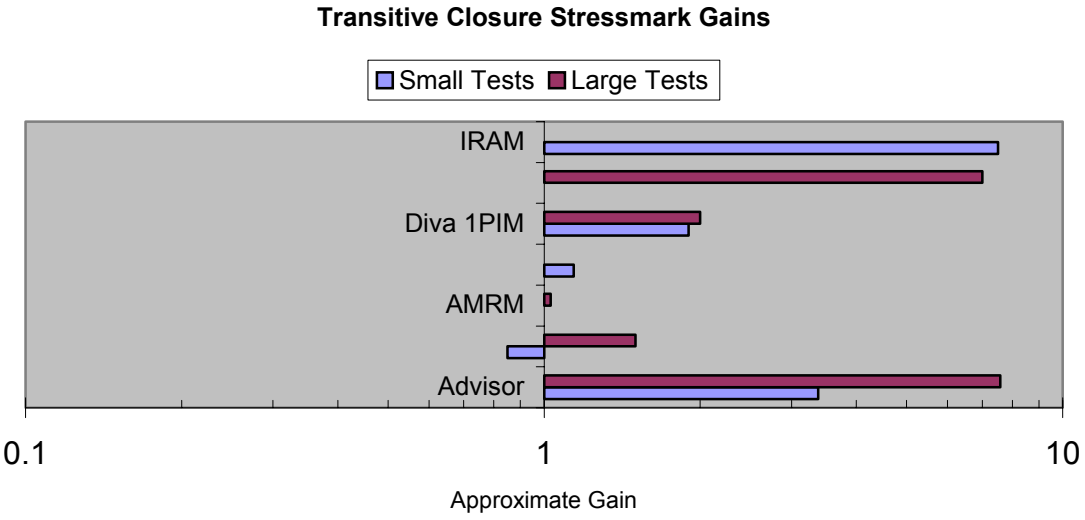
15.1.4 Demonstrated Matrix Stressmark Gains



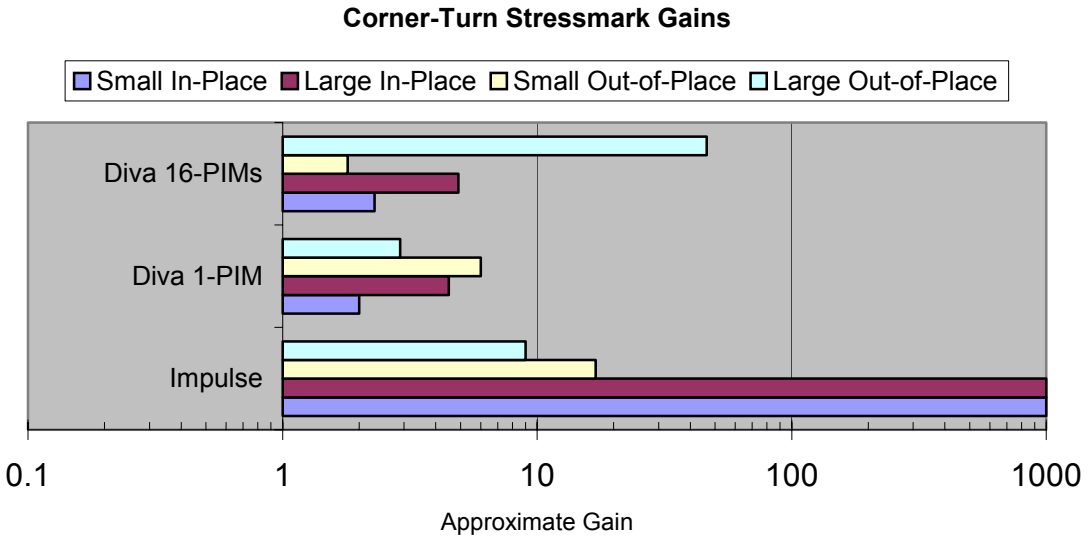
15.1.5 Demonstrated Field Stressmark Gains



15.1.6 Demonstrated Transitive Closure Stressmark Gains



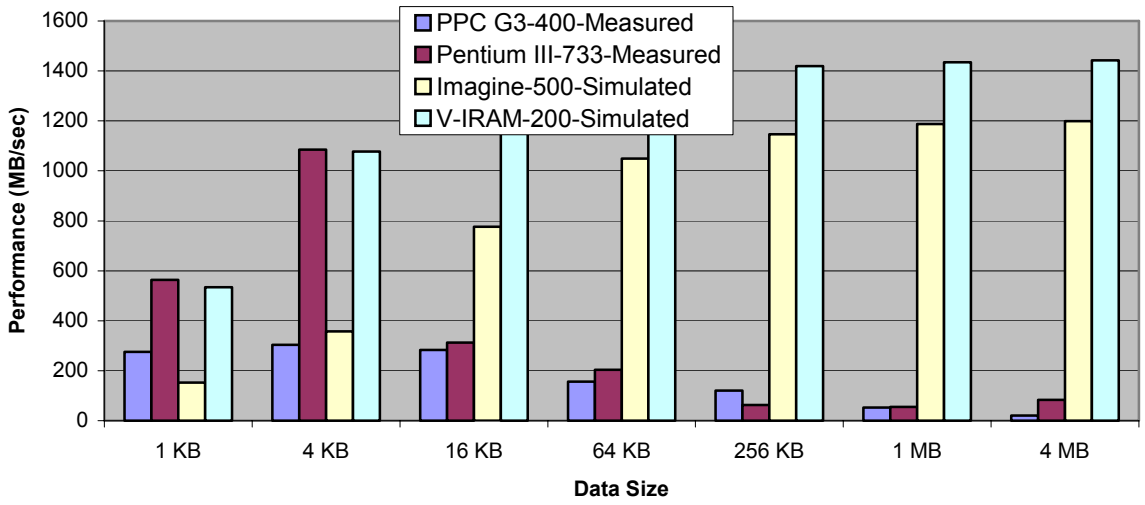
15.1.7 Demonstrated Corner-Turn Stressmark Gains



15.2 SLIIC Measurements

The *System Level Intelligent Intensive Computing* (SLIIC) project[69] is developing a system-level architecture that uses DIS PIM component technology to achieve two orders of magnitude of performance improvement on data-intensive radar processing applications. As part of this effort, the SLIIC team studied the performance of Imagine and IRAM components on radar processing applications. The team reported the following results for corner-turning. Note that the two OTS systems were measured, while the two DIS systems were simulated.

[69] http://www.east.isi.edu/projects/SLIIC/



15.3 Projections

In many ways, the DIS program is probably ahead of its time. The divergence of performance of processors and memory is real, but for the moment, the disparity can be hidden with ever larger and deeper caches. Systems ultimately become less efficient, but the pure speed offered by new processors is so great that there is a net performance gain even with the inefficiencies.

This may not always be true. For problems requiring global unpredictable access to data or movement of large data quantities, the processor-memory disparity is already a problem that cannot be hidden, and with memory latencies projected to increase to around 256 cycles in the next 1-2 years,[70] ever more algorithms will suffer data starvation. To compensate, designers may attempt to ensure that multiple tasks are always active, but that approach bears its own data starvation problems.

Even when the measured performance gain is slight, the dividends for future systems could be great. We anticipate that the approaches developed under the DIS program will increase in value exponentially over time.

[70] Larry Rudolph, *Malleable Caches* project review, 21 March, 2001.

16 ACRONYMNS

ASC3-Algorithmic Strategies for Compiler Controlled Caches

ADVISOR- Algorithms for Data Intensive Applications on Intelligent and Smart Memories

AFL- Adaptive Fetch Line Cache

ALS- Adaptive Line Size Cache

AMRM- Adaptive Memory Reconfiguration and Management

ATR- Automatic Target Recognition

BDL- Block Data Layout

CFAR- Constant False Alarm Rate

CMOS- Complimentary Metal Oxide Semiconductor

COTS- Commercial-Off-The-Shelf

CPU- Central Processing Unit

CSS- Case Sensitive Scheduling

DARPA- Defense Advanced Research Projects Agency

DBMS- Database Management System

DFT- Discrete Fourier Transform

DIS- Data Intensive Systems

DIVA- Data-IntensiVe Architecture

DoD- Department of Defense

DRAM- Dynamic Random Access Memory

DSIM- DIVA Simulator

DSPs- Digital Signal Processors

EM- Electromagnetic

FFT- Fast Fourier Transform

FIR- Finite Impluse Response

FW- Floyd-Warshall

GFLOPs- Giga Floating Point Operations Per Second

GLCM - gray-level co-occurrence matrix

GOPs- Giga Operations Per Second

GPO- Generalized Physical Optics

GPS- Global Positioning System

GUPS- Global Updates Per Second

HiDISC- Hierarchical Decoupled Instruction Stream Computer

IPC- Instructions per Clock-

IPTO- Information Processing Technology Office

IPR- Impulse Response

IRAM- Intelligent Random Access Memory

LRU- Least Recently Used

MoM- Method of Moments

MOPS- Millions of Operations Per Second

NRE- Non-Recurring Engineering

OLCD- Object Level Change Detection

OODB- Object Oriented Database

OPS- Operations per Second

OS- Operating System

OTS- Off-The-Shelf

PIM- Processor-in-Memory

PiRCs- PIM route components

PO- Physical Optics

PTF- Physical Theory of Diffraction

RAM- Random Access Memory

ROI- Region of Interest

RSIM- Rice Simulator for ILP Multi-Processors

SAR- Synthetic Aperture Radar

SB- Stream Buffers

SIMD- Single Instruction Multiple Data

SLIIC- System Level Intelligent Intensive Computing

SPD- Special Purpose Device

SPMV- Sparce Matrix Vector Multiply

SRAM- Static Random Access Memory

SRF- Stream Register File

TLB- Transaction Lookaside Buffer

URSIM- Utah RSIM

USTR- Unidirectional Space Time Representation

VC- Victim Cache

VIRAM- Vector Intelligence Random Access Memory

VLSI- Very Large Scale Integration