SANDIA REPORT

SAND
Unlimited Release
Printed

Uniprocessor Performance Analysis of a Representative Workload of Sandia National Laboratories' Scientific Applications

CHARLES LAVERTY

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online


Sandia National Laboratories

UNIPROCESSOR PERFORMANCE ANALYSIS OF A REPRESENTATIVE WORKLOAD OF SANDIA NATIONAL LABORATORIES' SCIENTIFIC APPLICATIONS

BY

CHARLES LAVERTY, B.S.

A thesis submitted to the Graduate School

in partial fulfillment of the requirements

for the degree

Master of Science in Electrical Engineering

New Mexico State University

Las Cruces, New Mexico

June 2005

"Uniprocessor Performance Analysis of a Representative Workload of Sandia National Laboratories' Scientific Applications," a thesis prepared by Charles David Laverty in partial fulfillment of the requirements for the degree, Master of Science in Electrical Engineering, has been approved and accepted by the following:

Linda Lacey
Dean of the Graduate School

Jeanine Cook
Chair of the Examining Committee

Date

Committee in charge:

Dr. Jeanine Cook, Chair

Dr. Erik DeBenedictis

Norris Green

Dr. Steven Stochaj

VITA

February 26, 1980	Born in Taos, New Mexico
1998	Graduated from Cimarron High School, Cimarron, New Mexico
1998-2002	B.S, New Mexico State University, New Mexico
2002-2005	Graduate Assistant College of Engineering New Mexico State University

Field of Study

Major Field: Electrical Engineering (Computer Engineering)

ABSTRACT

UNIPROCESSOR PERFORMANCE ANALYSIS OF A REPRESENTATIVE WORKLOAD OF SANDIA NATIONAL LABORATORIES' SCIENTIFIC APPLICATIONS

Master of Science in Electrical Engineering

New Mexico State University

Las Cruces, New Mexico, 2005

Dr. Jeanine Cook, Chair

Throughout the last decade computer performance analysis has become absolutely necessary to maximum performance of some workloads. Sandia National Laboratories (SNL) located in Albuquerque, New Mexico is no different in that to achieve maximum performance of large scientific, parallel workloads performance analysis is needed at the uni-processor level. A representative workload has been chosen as the basis of a computer performance study to determine optimal processor characteristics in order to better specify the next generation of supercomputers. Cube3, a finite element test problem developed at SNL is a representative workload of their scientific workloads. This workload has been studied at the uni-processor level to understand characteristics in the microarchitecture that will lead to the overall performance improvement at the multi-processor level. The goal of studying

this workload at the uni-processor level is to build a performance prediction model that will be integrated into a multi-processor performance model which is currently being developed at SNL. Through the use of performance counters on the Itanium 2 microarchitecture, performance statistics are studied to determine bottlenecks in the microarchitecture and/or changes in the application code that will maximize performance. From source code analysis a performance degrading loop kernel was identified and through the use of compiler optimizations a performance gain of around 20% was achieved.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

High performance computing has always been an integral part of our National Laboratories. As of November 2004, out of one hundred of the fastest computers in the world around twenty percent of them are located on National Laboratory grounds [1]. The workloads used on these supercomputers are scientific in nature, simulating various scientific phenomena such as projectile collisions and nuclear detonation, many of which the government uses to simulate physical problems that are not feasible to test in the real-world. Microarchitecture improvements can help to reduce the extensive runtime of these simulations. Performance analysis of these workloads is used to determine where bottlenecks occur in various microarchitecture components and/or where improvements to the application code will result in decreased execution time. From the various methods of performance analysis a study has been conducted on a representative workload of SNL's scientific applications. This study will aid in understanding the workload behavior throughout all phases of its execution and aid in identifying the causes poor performance, hopefully generating incite that will lead to changes in hardware or software, that enables speedup of these types of applications. These studies will provide the necessary information to enable generation of an analytic performance model at a uni-processor level which will be implemented in another multiprocessor model to help in the decision of future computing efforts at SNL.

2. BACKGROUND

A few concepts need to be understood in order to understand the performance of this representative workload at SNL. This section discusses the background concepts needed to understand the results presented in Section 6. This section is organized by presenting a short background to computer performance analysis in Section 2.1. Then a background of the microarchitecture of the Intel Itanium 2 architecture, which was the architecture that this analysis was conducted on, will be presented in 2.2. Section 2.2 and 2.3 will discuss some of the tools used to collect miscellaneous statistics in the performance study. Finally in Sections 2.5-2.7 discussion on background of the actual workload will be presented in order to help understand its performance characteristics.

2.1. Computer Performance Analysis

Computer performance analysis is the art of studying various workloads on different architectures to understand what can be done to maximize the performance of the workload on a specific architecture or to understand what architecture executes the workload the fastest and why. The main statistics that are studied for computer performance analysis are instructions per cycle (IPC) or cycles per instruction (CPI), execution time, as well as cache statistics, and stall statistics.

IPC is the number of instructions completed per cycle averaged over the entire application execution. For example, if two different processors

running the same application complete one instruction per cycle (max IPC per machine is one) and if the workload takes 1000 instructions to complete and on one processor it takes 2000 cycles (amount of time based on clock frequency) and takes 1000 cycles on another machine their IPC's are 0.5 and 1, respectfully. This states that the second processor is better due to the fact that it utilizes its architecture better even though the execution time (clock time) could be slower. So the combination of these two statistics is a major determination of how a processor performs.

Modern day microarchitectures have the ability to complete more than one instruction per cycle due to various architecture innovations such as the addition of multiple pipelines. Pipelines allow architectures the ability to break-up the execution time of an instruction into smaller pieces allowing for faster clock frequencies and for various pieces of different instructions to be in the pipeline at a single time. Performance degradation of these pipelined architectures comes from stalls in the pipeline, which occurs mainly when data is being brought in from disk or memory which is much slower than the processor. That is why cache statistics are also important in studying performance of architectures. Caches are small fast memory that help speed up the data access of disks or memory. The memory hierarchy of a contemporary computer is shown in Figure 1 where the registers are the closest to the CPU and the secondary storage is further away from the CPU.

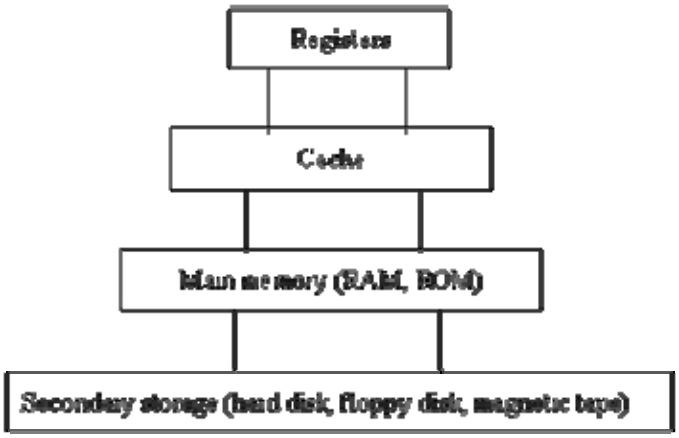


Figure 1 Processor Memory Hierarchy

The hierarchy starts with the slower secondary storage which runs at a speed of around 8ms and a bandwidth of around 20MB/sec [2]. The next level, main memory, runs at a speed of around 50ns and a bandwidth of 100MB/sec. The faster on-chip memories called cache run at a speed of a couple of nanoseconds and have a bandwidth in the gigahertz range. The highest level of the memory is the processor registers which run the fastest at around sub-nanosecond and at gigahertz bandwidth. The slower memory also is the cheapest as compared to the expensive on-chip registers and cache.

Many processors try to conceal the performance degradation contributed by the slow access time of caches by hiding the latency involved with a memory access. This is achieved by the use of processors that execute instructions out-of-order. This allows instructions following an instruction that is waiting for data to be computed while that instruction in waiting. When the instruction has its data from memory then the instructions are reassembled back to their program order. The key factor to hiding the

latency of a memory access comes down to the available parallelism within an application. In other words, if an instruction can not execute out-of-order then it is dependent on a previous instruction and if that previous instruction can not complete because it is waiting for one of its operands to be brought in from memory it also has to wait. There are always limits to the parallelism that can be extracted within an application due to the nature of its implementation.

The use of various performance analysis techniques enables designers to optimize the hardware design to excel performance of some workloads. There are three main approaches to performance analysis: Analytic methods, Simulation methods, and Direct Measurement methods. Analytic methods are techniques by which the behavior of the microarchitecture components is represented by mathematical equations or queuing models. The advantage of using analytic modeling is the time to achieve an answer is very small but it is very difficult to represent the behaviors and interactions of complex structures of modern day architectures because of their complexity. So the accuracy of such models is very limited.

Another method of performance evaluation is through the use of simulators. Simulators are very useful in performance analysis due to their robustness. However, a simulator is very difficult to implement because it is modeling a real world machine through software, making the development time costly. The accuracy of a simulator is always a concern because it is very difficult to model a processor through software. Most simulators provide

the designer with many performance metrics and simulator code can be modified to incorporate any changes to the microarchitecture that the designer desires such as modifying the cache organizations, memory bandwidth, and queue sizes. Simulators have one major drawback and that is the execution time. The runtime of an application on a cycle-accurate simulator usually takes 100 to 1000 times the native applications execution time.

The last of the performance analysis tools is direct measurement of physical systems. Through the use of on-chip hardware counters performance data can be collected during native execution of an application as well as during operating system activity. The problem with hardware counters is that in most modern computers processor real estate is limited and thus there are typically only four to eight counters that enable the study of an application, which leads to very limited performance studies. Also, there are some variations between runs of the same application due to overhead involved with the operating system and cache activity but normally is less than 0.5% error between runs.

In this paper the majority of the performance analysis is done through the use of hardware performance counters. Intel's® Itanium® 2 microarchitecture provides one of the most extensive performance monitoring units for capturing performance data. The next section provides an overview of this architecture.

2.2. Intel® Itanium 2®

The Itanium 2 microarchitecture was a collaborative effort between HP and Intel and was released in July 2002. It is a 64-bit, VLIW (very long instruction word) architecture executing up to six instructions at a time. The VLIW in the Itanium 2, consists of groups of three instructions; these groups are called bundles. The three instructions in a bundle are independent and can, therefore, execute in parallel on the multiple functional units in the Itanium 2. It has four floating-point units, two capable of executing one FMA per cycle while the other two perform other floating-point operations such as comparisons, but only two floating-point operations can be executed in parallel [3] . Also the Itanium 2 has two integer, three branch, and four memory execution units in its parallel execution pipelines, for a total of 12 functional units. The Itanium 2's pipeline is shown in Figure 2. Each instruction bundle is encoded by the type of resource that can execute in parallel (i.e. memory, floating-point, and branch instructions comprise a MFB bundle). The Itanium 2 issues two bundles per cycle to its in-order core.

Through the use of VLIW technology, the Itanium 2 does not need a complex out-of-order pipeline to achieve performance improvement, allowing the architecture real estate to be used for a more complex memory system and a large set of architectural registers. The VLIW places much responsibility on the compiler to achieve the maximum instruction throughput. The microarchitecture highlights are shown in Figure 3 [4]. The cache

configurations of the Itanium 2 processor used in our studies are listed in Table 1. The cache hierarchy of the Itanium 2 populates most of the area on the chip since the processor core is smaller for an in-order processor compared to out-of-order cores normally used in today's microcomputers. Overall, the Itanium 2 is a good choice for running the performance analysis of this representative application of SNL's scientific applications because of the large caches, the high performance of scientific applications cited in various studies, and the large number of performance metrics that the performance monitoring unit allows for collection which is discussed in the next section.

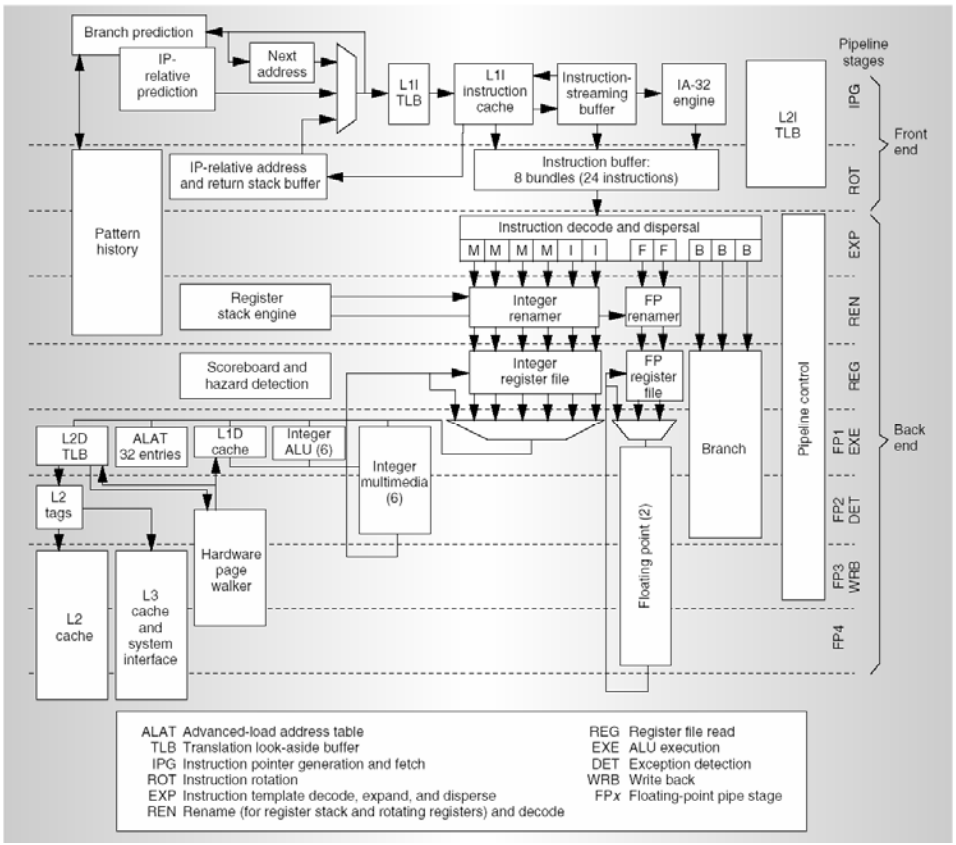


Figure 2 Itanium 2 Processor Pipeline

Table 1. Features of the Itanium 2 processor.

Design		
Frequency	1 GHz	
Pipe stages	8 in-order	
Issue/retire	6 instructions	
Execution units	2 integer, 4 memory, 3 branch, 2 floating-point	
Silicon		
Technology	180 nm	
Core	40 million transistors	
L3 cache	180 million transistors	
Size	421 mm^2	
Caches		
L1 instruction	Size	16 Kbytes
	Latency	1 cycle
	Protection	Parity
L1 data	Size	16 Kbytes
	Latency	1 cycle
	Protection	Parity
L2	Size	256 Kbytes
	Latency	5, 7, or 9+ cycles
	Protection	Parity or ECC*
L3	Size	3 Mbytes
	Latency	12+ cycles
	Protection	ECC
Benchmark results		
Spec CPU2000 score	810	
Spec FP2000 score	1,431	
TPCC (32-way)	433,107 transactions per minute	
Stream	3,700 Gbytes/s	
Linpack 10K**	13.94 Gflops	

* ECC: error-correcting code

** Performed with four processors

Figure 3 Itanium 2 Microarchitecture Features

Table 1 Itanium 2 Cache Configurations

	L1D	L2	L3
Access Time	1	5+	12+
Size	16 KB	256 KB	1.5MB
Line Size	64 bytes	128 bytes	128 bytes
Number of Lines	256	2048	24576
Associative Sets	4	8	12
Sets	64	256	2048
Update Policy	Write-Through	Write-Back	Write-Back
Banks	8 "groups"	16	1
Line Replacement	Not Recently Used	Not Recently Used	Not Recently Used

2.2.1. Performance Monitoring Unit

The Itanium 2's performance monitoring unit (PMU) is an important component of the architecture that allows a developer to tune their code to achieve maximum performance through the use of hardware performance counters. The PMU has the ability to track counts of around 500 different metrics which are collected on four different hardware counters. The metrics that can be counted range from simple memory statistics (misses, references, etc.) to branch miss-prediction rates to complex opcode matching (shown below). The Itanium 2's PMU is one of the most complex ever implemented which influences our choice of this architecture for performance analysis. One advantage to using the Itanium's PMU is its bubble counters that can be used to understand the major contributions of all stalls throughout application execution. A detailed explanation of the bubble analysis is included in Section 6.3. The events that the PMU can monitor can be broken down into various categories [5]. These categories are:

- ***Basic Events***: Clock cycles, retired instructions
- ***Instruction Dispersal Events***: Instruction decode and issue
- ***Instruction Execution Events***: Instruction execution, data and control speculation, and memory operations.
- ***Stall Events***: Stall and execution cycle breakdowns.
- ***Branch Events***: Branch prediction.
- ***Memory Hierarchy***: Instruction and data caches.
- ***System Events***: Operating system monitors.
- ***TLB Events***: Instruction and data TLB's.
- ***System Bus Events***: Events on the system bus

- ***RSE Events***: Register Stack Engine.

2.3. PERFMON

There are several software interfaces to the performance-monitoring unit including HP's PERFMON and Intel's Vtune. We chose to use both although the majority of the statistics were collected using PERFMON because of its ease of use and its ability to batch process the workload using the PERFMON software. PERFMON was a project developed by Hewlett Packard as a standard kernel interface for the Performance Monitoring unit of the Itanium and Itanium 2 architectures [6]. The software consists of a library called *libpfm* and a monitoring tool called *pfmon*. PERFMON provides full access to the PMU of the Itanium family of architectures. It provides the ability to monitor system or per-process sessions, as well as providing the capability of sampling events or just cumulative counts of the available metrics.

Pfmon has many features in which a user can customize performance counts. We used *pfmon* version 3.0 on our system. The basic usage of the pfmon command is shown in Figure 4. In this example, two metrics are counted, CPU cycles and instructions retired with the results of each being 436,368 and 513,437, respectively. With performance counters there is always a deviation from one run to another. For the command shown in Figure 4 a total of ten runs were completed and the percent error in instructions was less than 0.01 percent; for the CPU cycles the max percent error was around 0.65.

```
% pfmon -e cpu_cycles,IA64_inst_Retired ls /dev/null
   /dev/null
                    436368 CPU_CYCLES
                    513437 IA64_INST_RETIRED
```

Figure 4 PFMON Usage

In Figure 4 the *–e* specifies the event to be counted (up to four can be counted at a time) and *ls /dev/null* is the command to be monitored. Through the use of *-header* the output will include output useful information about your system and the performance session started by *pfmon*. A sample output using the *–header* option is seen in Figure 5. Some of the key information shown in Figure 5 is the cache hierarchy configuration of the system as well as how the output data of the sampling session is organized.

Sampling events can be very useful information when studying a workload because you can determine execution phases within the program that cause performance degradation. Using pfmon, a sampling period in which two statistics are used can be specified, one event is used to choose when to sample the other. For example, the user can specify to sample *CPU_CYCLES* every one hundred thousand *IA64_INST_RETIRED*. From the number of cycles and the number of instructions, instructions per cycle (IPC) can be computed. IPC is a composite metric that is used to measure overall performance of a micro-architecture, which was discussed in Section 2.1. Figure 5 also shows an example of how to implement sampling (look for “command”). Plots of such interval data will be shown in Section 6. The PMU

of the Itanium 2 allows for various other software to interface with it such as Vtune which is described in the next section.

```
# date: Mon Mar  7 15:01:39 2005
#
# hostname: klipsch
#
# kernel version: Linux 2.6.9-5.EL #1 SMP Wed Jan 5 19:23:24 EST 2005
#
# pfmon version: 3.0
# kernel perfmon version: 2.0
#
#
# page size: 16384 bytes
#
# CLK_TCK: 1024 ticks/second
#
# CPU configured: 0
# CPU online    : 4
#
# physical memory         : 8526266368 bytes (8131.3 MB)
# physical memory available: 6630768640 bytes (6323.6 MB)
#
# host CPUs:  4-way 900MHz/1.5MB Itanium 2 (McKinley, B3)
#       PAL_A: 0.7.31
#       PAL_B: 0.7.40
#       Cache levels: 3 Unique caches: 4
#       L1D:    16384 bytes, line  64 bytes, load_lat   1, store_lat   3
#       L1I:    16384 bytes, line  64 bytes, load_lat   1, store_lat   0
#       L2 :   262144 bytes, line 128 bytes, load_lat   5, store_lat   7
#       L3 :  1572864 bytes, line 128 bytes, load_lat  12, store_lat   7
#
#
# captured events:
#       PMD4: IA64_INST_RETIRED
#       PMD5: CPU_CYCLES
#
# privilege levels:
#       PMD4: IA64_INST_RETIRED = user
#       PMD5: CPU_CYCLES        = user
#
# monitoring mode: per-process
#
#
# instruction sets:
#       PMD4: IA64_INST_RETIRED = ia32/ia64
#       PMD5: CPU_CYCLES        = ia32/ia64
#
#
# command: pfmon --smpl-module=compact-ia64 --short-smpl-period=1000000 --#long-smpl-
period=1000000 --aggregate-results -e #ia64_inst_retired,CPU_CYCLES --with-header --
verbose --smpl-#outfile=./output/cube_crs/CPU_CYCLES/stat_w8_d8_dofl --append
#/users/claverty/fei-2.10/chip/src/utest/cube3.exe -d / -i #/users/claverty/fei-
2.10/chip/src/utest/test/input
#
#
#
# recorded PMDs when IA64_INST_RETIRED overflows: PMD5
#
# short sampling rates (base/mask/seed):
#       IA64_INST_RETIRED 1000000
#       CPU_CYCLES none
#
# long sampling rates (base/mask/seed):
#       IA64_INST_RETIRED 1000000
#       CPU_CYCLES none
#
#
# sampling buffer entries: 2048
#
# description of columns:
#       column  1: entry number
#       column  2: process id
#       column  3: cpu number
#       column  4: instruction pointer
#       column  5: unique timestamp
#       column  6: overflowed PMD index
#       column  7: initial value of first overflowed PMD
#
#       when PMD4(IA64_INST_RETIRED) overflows:
#               column  8: PMD5
#
```

Figure 5 Sample Output of PFMON

2.4. Vtune

Vtune is a performance tool used to tune an application for maximum performance [7] through the use of profile and call graph results. The call graph and profile data allows Vtune to sort the functions by actual time spent throughout execution. This helps to identify sections of code where most of the execution is spent which are called hotspots. Once the hotspots have been identified by Vtune, it allows a user to double-click on a function within the call graph in the graphical user interface and backtrack to the actual source-code. Vtune also interfaces to the performance counters of the Itanium 2, Xeon, or Pentium and can perform sampling sessions just as PERFMON does on the Itanium 2 discussed in Section 2.3. Vtune is used in this work to help understand the application under study and to help determine sections of code that are used frequently within that application.

2.5. Trilinos

The applications that simulate large-scale physical systems are very mathematical in nature requiring the solution of many different linear and non-linear systems of equations, both time-dependent and independent [8]. At the core of these types of applications are various mathematical solvers that implement different algorithms to solve these systems of equations. Scalable solver algorithms and software development have long been an area of focus at Sandia National Laboratories. In the past, the development of these algorithms and applications was done by the individual scientist and each

created their own code even though many codes implemented the same underlying solver algorithm. The development of these codes was very time consuming and expensive. The Trilinos Project was created at Sandia to develop a reusable set of solvers to help reduce development time and overall expense. The Trilinos Project is a highly evolved system of libraries or packages that has been created as a basis for future complex application development and is currently in use at Sandia National Laboratories. Trilinos is used by many of the scientific applications at Sandia. The representative workload used in this work also uses the Trilinos set of solvers to solve its equations.

The various packages used by Trilinos are each independent of one another although some packages can be used in conjunction with other Trilinos packages. Some of the packages used in this work include:

- **Aztec00 -** *provides an object-oriented interface the well-known Aztec solver library. It also allows flexible construction of matrix and vector arguments via Epetra matrix and vector classes.*
- **Epetra -** *provides the fundamental construction routines and services that are required for serial and parallel linear algebra libraries. Epetra provides the underlying foundation for all Trilinos solvers.*
- **ML -** *is a multigrid preconditioning package intended to solve large sparse linear systems of equations arising from primarily elliptic Partial Differential Equation discretizations.*

The representative Sandia application that we use in this work (which is described in Section 5) primarily uses the Epetra and the AztecOO packages. The Epetra package is the primary package used to create and fill matrices used in many scientific applications and the AztecOO package is the primary solver used in the representative workload. The Trilinos Project and its packages are described in detail in the Trilinos Tutorial [9], as well as other Trilinos documentation [10, 11].

2.6. Finite Element Method

The representative workload used in this work makes use of the finite element method to solve a physical system. Some background in the finite element method helps to understand the execution of the representative application,

2.6.1. Overview

The basis of the math used in scientific applications is the use of Partial Differential Equations (PDE's). PDE's are used to model physical phenomena though the use of equations that describe the relationship of physical quantities such as forces, temperature, chemical reactions, and velocity by partial derivatives. If the physical system is very large in nature it is generally not possible to obtain a solution satisfying the governing PDE's and so the process of subdivision of the physical system into smaller portions is used which is known as finite-element discretization. There are various ways to discretize partial differential equations. In this work we focus on the Finite

Element Method (FEM), as opposed to Finite Difference and Finite Volume Methods.

Finite element analysis (FEA) or FEM allows a large naturally occurring physical phenomenon to be represented by mathematical equations and to be solved in parts and then combined to solve the entire problem. Through this divide and conquer scheme a problem is divided into parts called elements and the regions where the elements connect are called nodes. There are various ways of constructing elements in a finite element problem. Elements that are one, two, or three dimensional in nature will show the definition of the physical object in more or less detail. Figure 6 shows various shapes of elements in various dimensions. The tetrahedral element is the most widely used because it can closely model any physical shape whereas the hexahedral element will best model a rectangular shape.

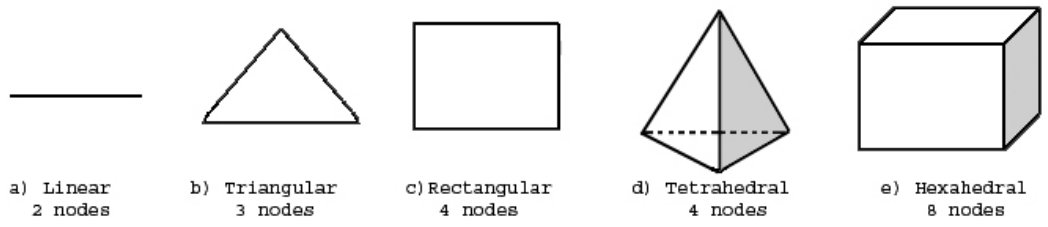


Figure 6 Element Types

Finite element analysis has six steps in the solution procedure:

1. Discretize the continuum
2. Select interpolation functions

3. Find the element properties
4. Assemble the element equations
5. Solve the global equation system
6. Compute additional results

Once a physical system is realized the first step is to discretize the physical continuum into elements and nodes, which are stored as element connectivity lists usually stored in an array. The nodes are the vertices or corners of the elements shown in Figure 6. The nodal coordinates are also stored in an array. Step two is to select the interpolation functions that are used to track various interactions between the variables over the element such as displacement. Each element is defined using this interpolation function to describe its behavior between its endpoints (nodes) based on various equations describing the physical phenomenon. Once the element has been discretized and the interpolation functions have been realized, the matrix equations which relate the nodal values to their unknowns for the element need to be established. Once the equations are established for each element the assembly process begins in which the global equation is assembled element by element according to each element's connectivity list. The global equation represents the whole physical object that the FEM is modeling. Before the global equation can be solved boundary conditions must be implemented which describe the physical force or strains on the physical object. The global equation is typically a matrix-vector multiply in the form:

$$\{f\} = [A]\{x\} \qquad \textbf{Equation 1}$$

where *A* is the matrix that represents the known coefficients and *x* represents the unknown values (in vector form). *A* is also known as the global stiffness matrix. Once the solution is complete there are times when additional results other than coordinate displacements need to be computed such as temperature variations within the original object modeled.

In finite element analysis there are many different solution techniques and/or ways of generating/assembling the matrices used in the calculations but all generally result in the same final equation of a matrix-vector or matrix-matrix multiply [12, 13, 14, 15, 16].

2.6.2. Assembly

The assembly process of the finite element method is the process of assembling the global stiffness matrix from the individual elements to characterize the unified behavior of the entire system. This is carried out using the element connectivity lists, which makes up the global numbering system of the problem and states how the elements are connected to realize the actual physical object. When assembling the matrix the contribution of each element is added to the global matrix. An example of the assembly process of a linear system with one variable is shown in the following example.

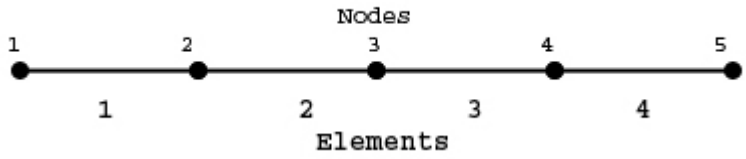


Figure 7 Assembly Example Problem

Figure 7 is a one dimensional physical system in which there is one degree of freedom (one-direction, *i.e.* x-direction). Figure 6 shows the physical system separated into four elements with five nodes joining the elements. The connectivity list of the finite element problem in Figure 7 is shown in Figure 8. Each element has two nodes numbered one and two as well as a global number associated with each node as shown in Figure 8.

	Node Numbers	
Element	Local	Global
1	1	1
	2	2
2	1	2
	2	3
3	1	3
	2	4
4	1	4
	2	5

Figure 8 Connectivity List

The element connectivity list shown in Figure 8 states how the example in Figure 7 will be assembled into a global matrix. Each element will contribute a 2x2 matrix to the global matrix because each element has two nodes associated with it. Each element contributes the 2x2 matrix shown in Figure 9. The value located in (1,1) (e.g., row one, column one) represents the contribution of node one and the value of (2,2) represents the contribution

of node two for each element. The values in Figure 9 are arbitrary in that the values were chosen for simplicity to show the assembly process. The negative values represent the connection between the nodes meaning nodes one and two are connected in the element.

$$\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$$

Figure 9 Element Contributions

The assembly process takes each individual element's 2x2 matrix and places it in the final matrix based on the column/row of the global node number shown in Figure 8.

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad a)$$

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 1+1 & -1 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad b)$$

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 1+1 & -1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad c)$$

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 1+1 & -1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \quad d)$$

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \quad e)$$

Figure 10 Assembly Process

The global matrices shown in Figures 10a-e each have five rows and columns because the object in Figure 7 has five nodes. When an element has shared nodes with another element the values associated with that node are additive

in the global matrix (Shown in Figure 10b, c, d). This is also shown in detail in section 3.6.4. Figure 10a shows the individual contribution of element one to the global matrix; Figure 10b shows the contribution of element two; Figure 10c shows element three; and Figure 10d shows element four's contribution. Each element's contributions are added to the global matrix. Figure 10e is the final *A* matrix which is the global stiffness matrix of the finite element problem. This final global matrix is normally a sparse matrix for large finite element problems. A sparse matrix is a matrix with few nonzeros usually situated close to the diagonal of the matrix. Sparse matrices are described in detail in Section 2.7.

The global matrix can take on various shapes and orderings based on the node numberings and shape of the physical system. For instance in Figure 10, the problem shape is a long bar with seven elements and a total of 32 nodes. The graph of the matrix is a narrow banded matrix (Figure 11) meaning the nonzero values are associated closely to the diagonal of the matrix. Figure 12 shows a three by three problem with six elements and 32 nodes. This matrix has a very different graph but has the same number of rows and although they are similar in size the performance changes in the multi-processor setting due to the extra communication between processors. Although this is not within the scope of this work, mesh generators and graph theory play an important role in performance and behavior of these finite element problems.

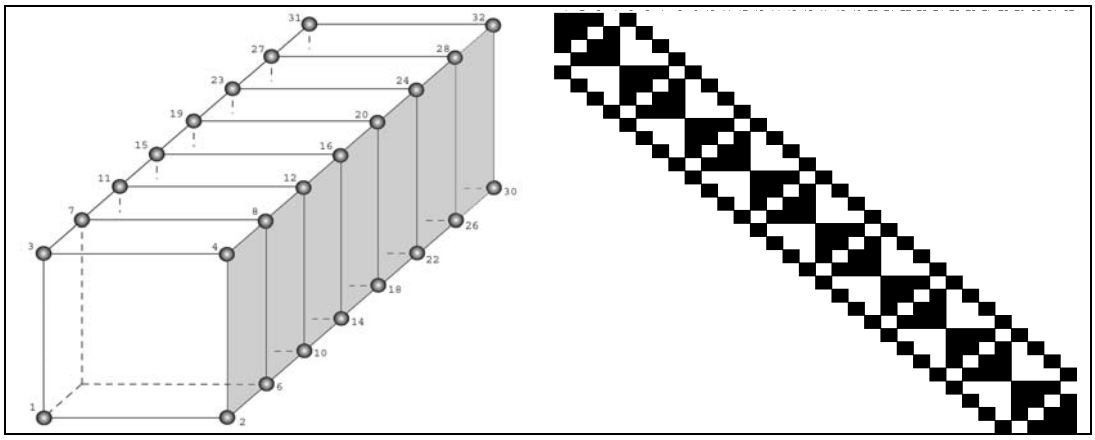


Figure 11 Beam Representation and Matrix Graph

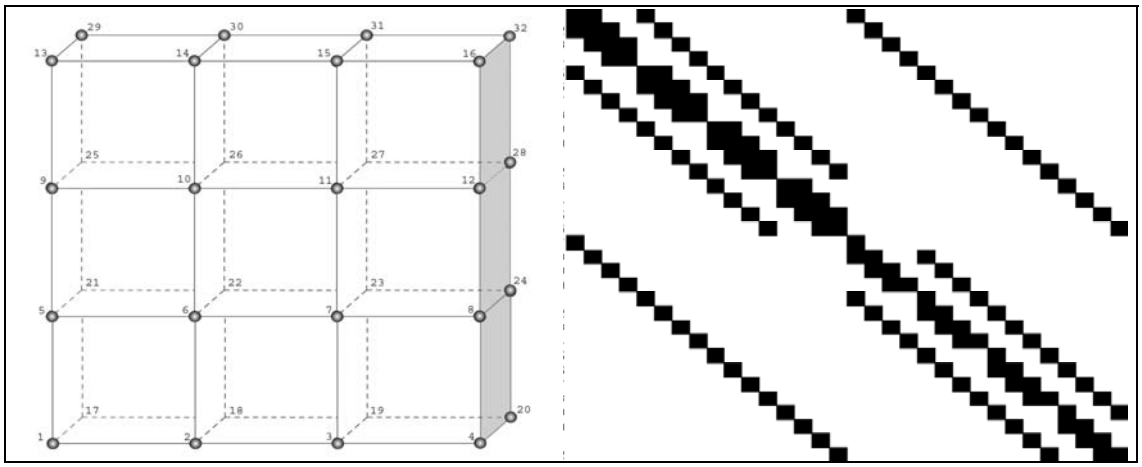


Figure 12 Plate Representation and Matrix Graph

2.6.3. Preconditioning/Solution

Once the *A* matrix which has been assembled into the global matrix form it can be solved. There are various ways to solve these large sparse matrices. For large sparse matrices it is not feasible to produce an exact solution since this would take an infinite amount of time to compute. Instead there are various ways to converge on an approximation of the exact result

through the use of various iterative and projection methods. The AztecOO package within the Trilinos set of solvers allows for six various “solver” routines [17]. These routines are:

- AZ_cg- Conjugate gradient
- AZ_gmres- Restarted generalized minimal residual
- AZ_cgs- Conjugate gradient squared
- AZ_tfqmr- Transpose-free quasi-minimal residual
- AZ_bicgstab- Bi-conjugate gradient with stabilization
- AZ_lu- Sparse Direct Solver

These routines provide various techniques for converging on the exact solution to the matrix-vector equations (Equation 1). These equations can be preconditioned to allow for faster convergence. Preconditioning these equations in matrix form requires another matrix (preconditioning matrix) to be multiplied in order to achieve faster convergence and is beyond the scope of this work. The AztecOO package provides five methods of preconditioning. These preconditioners are:

- AZ_Jacobi- *k* step Jacobi
- AZ_Neumann- Neumann series polynomial
- AZ_ls- Least-squares polynomial
- AZ_symm_GS- Non-overlapping domain decomposition (additive Schwartz) *k* step symmetric Gauss-Siedel

- AZ_dom_decomp- Domain decomposition preconditioner (additive Schwartz)

Although these are the preconditioners available in the AztecOO package, various other preconditioners exist within the Trilinos solvers, such as a multi-level preconditioner found in the ML package.

2.6.4. Finite Element Example

An example of finite element analysis is helpful in understanding the process of computing the final result in a finite element problem and proved extremely useful in understanding performance analysis data from our actual study of the representative application used in this work. This example will show how the assembly of the global matrix takes place from the element stiffness matrices. Figures 13-15 show the example implemented in COSMOS which is a finite element solver used in Solidworks.

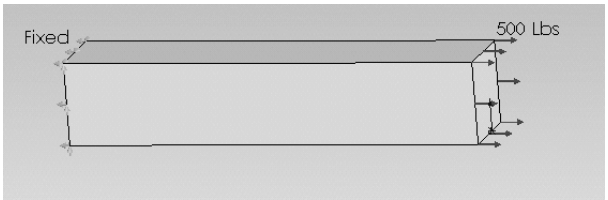


Figure 13 Cosmos Example

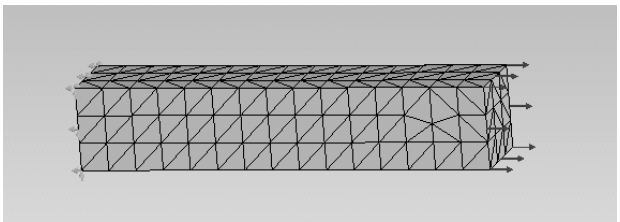


Figure 14 Cosmos Elements

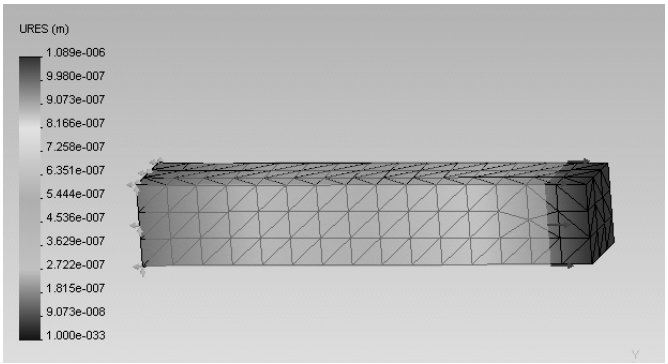


Figure 15 Cosmos Final Solution

In these figures the example is a beam shaped object with one side fixed and the other with a force of 500 ft-lbs in the opposite direction. The elements used within COSMOS are a tetrahedral shape and in this test problem there are a total of 848 elements and 1563 nodes. As seen in Figure 12 there is a displacement on the right of the figure due to the force of 500 lbs. The example that is shown next is a simplified version of the actual computation of the elements and how the elements are assembled to represent the whole system that is shown in the COSMOS example. The differences in this example and the COSMOS example are that the elements are triangular in shape as opposed to tetrahedral and the example is two-dimensional. Figure 16 shows the layout of the beam and the location on the “xy” plane.

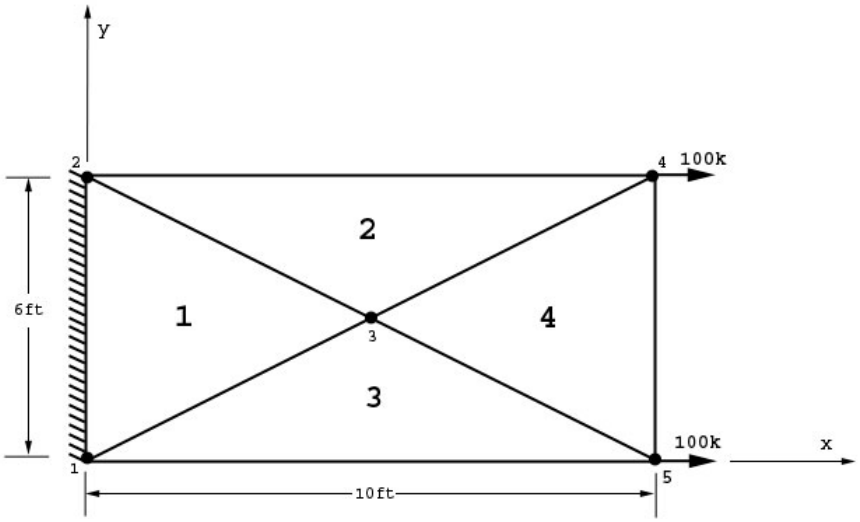


Figure 16 Thin Plate Example

This example shown in Figure 16 is a structural analysis problem of a thin plate with dimensions 6 feet wide, 10 feet long, and 1 inch thick. It is fixed along the y axis and is acting under pure tensile loads of 100 kips each, applied at two corners. This problem is trying to solve for the displacements in the x and y directions. Some other information that is needed to solve the finite element method of the plate is Young's modulus and Poisson's ratio values of 30,000 and 0.5, respectfully. The structure is already split into four elements label 1-4 with five nodes labeled 1-5. The governing matrix equation for the analysis of the structure is given by Equation 2:

$$[K]\{\partial\} = \{Q\} \qquad \textbf{Equation 2}$$

where the global stiffness matrix [*K*] is defined by Equation 3 which is the sum of each individual element's stiffness matrix:

$$[K] = \sum_{1}^{n} [k] \qquad \textbf{Equation 3}$$

The expression that defines the element stiffness matrix [*K*] is given by Equation 4 which is derived from an equilibrium equation in structural analysis. Its explanation is beyond the scope of this work but is expressed as:

$$[k] = \{B\}^T [D]\{B\} \cdot t \cdot \frac{dx \cdot dy}{2} \qquad \textbf{Equation 4}$$

The displacement-strain matrix {*B*} is defined for a triangular element defined by nodes *i, j, m* and is expresses in Equation 5, in which x_i, x_j, x_m, y_i, y_j, and y_m are the nodal coordinates in the "*xy*" plane of nodes *i, j,* and *m,* respectfully. The thickness of the plate is defined by *t* in Equation 4 and the area of the element is given by *A* in Equation 5 which is also equal to twice the value of $dx \cdot dy$.

$$\{B\} = \frac{1}{2A}\begin{bmatrix} yj-ym & 0 & ym-yi & 0 & yi-yj & 0 \\ 0 & xm-xj & 0 & xi-xm & 0 & xj-xi \\ xm-xj & yj-ym & xi-xm & ym-yi & xj-xi & yi-yj \end{bmatrix} \qquad \textbf{Equation 5}$$

The elasticity matrix for a plane stress analysis problem in a two-dimensional setting is given by Equation 6 and its derivation is also beyond the scope of this work.

$$\{D\} = \frac{E}{1-v^2}\begin{bmatrix} 1 & v & 0 \\ v & 1 & 0 \\ 0 & 0 & \frac{1-v}{2} \end{bmatrix} \qquad \textbf{Equation 6}$$

The general nodal displacement matrix {*∂*} is shown in terms of *u* and *v* which correlate to *x* and *y*, respectfully. The term *u1* in Equation 7 correlates to the

"x" displacement for node 1 and *v1* correlates to the *"y"* displacement for node 1 and so on.

$$\{\partial\} = \begin{Bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ u_3 \\ v_3 \\ u_4 \\ v_4 \\ u_5 \\ v_5 \end{Bmatrix}$$ **Equation 7**

The boundary condition of 100 kips is applied to nodes 4 and 5 as defined in Figure 16 and is illustrated by the Equation 8 in which the terms also correlate to the terms in Equation 7.

$$\{Q\} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 100 \\ 0 \\ 100 \\ 0 \end{Bmatrix}$$ **Equation 8**

The generation of the global stiffness matrix is achieved by computing the stiffness matrix associated with each element and then assembling them into the global matrix. Figure 17 shows the coordinates of the nodal points of the structure.

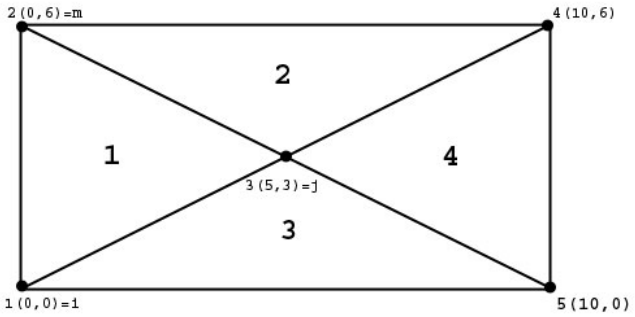


Figure 17 Thin Plate Nodal Coordinates

The generation of element stiffness matrix is shown only for element 1 as each other element is found using the same method. The displacement-strain matrix {*B*} of element 1 is shown below.

$$\{\mathbf{B}\} = \frac{12}{2 \cdot 30 \cdot 144} \begin{Bmatrix} (3-6) & 0 & 6 & 0 & -3 & 0 \\ 0 & -5 & 0 & 0 & 0 & 5 \\ -5 & (3-6) & 0 & 6 & 5 & -3 \end{Bmatrix} \qquad \textbf{Equation 9}$$

Equation 9 shows the displacement-strain matrix with the area equal to 30 times one inch (12/144) and the values of xi, xj, xm, yi, yj, and ym are the coordinates in the *"xy"* plane shown in Figure 17. Equation 10 shows the simplification of Equation 9:

$$\{\mathbf{B}\} = \frac{1}{720} \begin{Bmatrix} -3 & 0 & 6 & 0 & -3 & 0 \\ 0 & -5 & 0 & 0 & 0 & 5 \\ -5 & -3 & 0 & 6 & 5 & -3 \end{Bmatrix} \qquad \textbf{Equation 10}$$

The elasticity matrix [*D*] (Equation 11) is shown using Young's modulus (E in Equation 6) and Poisson's ratio (v in Equation 6).

$$\{D\} = \frac{30000}{1-(0.5)^2}\begin{bmatrix} 1 & 0.5 & 0 \\ 0.5 & 1 & 0 \\ 0 & 0 & \frac{1-0.5}{2} \end{bmatrix}$$ **Equation 11**

Equation 11 simplifies to the following matrix form of Equation 12:

$$\{D\} = \frac{30000}{0.75}\begin{bmatrix} 1 & 0.5 & 0 \\ 0.5 & 1 & 0 \\ 0 & 0 & 0.25 \end{bmatrix}$$ **Equation 12**

The calculation of the element stiffness matrix is determined by Equation 13:

$$[k]_1 = \{B\}^T[D]\{B\} \cdot t \cdot A$$ **Equation 13**

The constants of the matrix multiplication are precomputed in Equation 14:

$$\left\{\left(\frac{1}{720}\right)^2\left(\frac{30000}{0.75}\right)(1)\left(\frac{5x6x144}{2}\right)\right\} = 166.67$$ **Equation 14**

Equation 15 shows the substitution of the values of the matrices *B* and *D* in Equation 13:

$$[k]_1 = (166.67)\times\begin{bmatrix} -3 & 0 & -5 \\ 0 & -5 & -3 \\ 6 & 0 & 0 \\ 0 & 0 & 6 \\ -3 & 0 & 5 \\ 0 & 5 & -3 \end{bmatrix}\cdot\begin{bmatrix} 1 & 0.5 & 0 \\ 0.5 & 1 & 0 \\ 0 & 0 & 0.25 \end{bmatrix}\cdot\begin{bmatrix} -3 & 0 & 6 & 0 & -3 & 0 \\ 0 & -5 & 0 & 0 & 0 & 5 \\ -5 & -3 & 0 & 6 & 5 & -3 \end{bmatrix}$$ **Equation 15**

Equation 15 simplifies to Equation 16:

$$[k]_1 = (166.67)\times \begin{bmatrix} -3 & -1.5 & -1.25 \\ -2.5 & -5 & -0.75 \\ 6 & 3 & 0 \\ 0 & 0 & 1.5 \\ -3 & -1.5 & 1.25 \\ 2.5 & 5 & -0.75 \end{bmatrix} \cdot \begin{bmatrix} -3 & 0 & 6 & 0 & -3 & 0 \\ 0 & -5 & 0 & 0 & 0 & 5 \\ -5 & -3 & 0 & 6 & 5 & -3 \end{bmatrix}$$

Equation 16

And further to Equation 17:

$$[k]_1 = (166.67)\times \begin{bmatrix} 15.25 & 11.25 & -18 & -7.5 & 2.75 & -3.75 \\ 11.25 & 27.25 & -15 & -4.5 & 3.75 & -22.75 \\ -18 & -15 & 36 & 0 & -18 & 15 \\ -7.5 & -4.5 & 0 & 9 & 7.5 & -4.5 \\ 2.75 & 3.75 & -18 & 7.5 & 15.25 & -11.25 \\ -3.75 & -22.75 & 15 & -4.5 & -11.25 & 27.25 \end{bmatrix}$$

Equation 17

The contributions of *Kij* (nodes) to the global matrix are shown below with each being a 2x2 matrix corresponding to 2 degrees of freedom, one in the x direction and the other in the y direction. Recall degrees of freedom represent a variable within the problem that can represent direction, velocity, displacement, or any other physical variable that needed to solve these finite element problems.

For element 1:

$$K_{11} = \begin{bmatrix} 15.25 & 11.25 \\ 11.25 & 27.25 \end{bmatrix} \quad K_{13} = \begin{bmatrix} -18 & -7.5 \\ -15 & -4.5 \end{bmatrix} \quad K_{12} = \begin{bmatrix} 2.75 & -3.75 \\ 3.75 & -22.75 \end{bmatrix}$$

$$K_{31} = \begin{bmatrix} -18 & -15 \\ -7.5 & -4.5 \end{bmatrix} \quad K_{33} = \begin{bmatrix} 36 & 0 \\ 0 & 9 \end{bmatrix} \quad K_{32} = \begin{bmatrix} -18 & 15 \\ 7.5 & -4.5 \end{bmatrix}$$

$$K_{21} = \begin{bmatrix} 2.75 & 3.75 \\ -3.75 & -22.75 \end{bmatrix} \quad K_{23} = \begin{bmatrix} -18 & 7.5 \\ 15 & -4.5 \end{bmatrix} \quad K_{22} = \begin{bmatrix} 15.25 & -11.25 \\ -11.25 & 27.25 \end{bmatrix}$$

For element 2:

$$K_{22}=\begin{bmatrix}15.25 & 11.25\\ 11.25 & 27.25\end{bmatrix} \quad K_{23}=\begin{bmatrix}-12.5 & 15\\ 7.5 & -50\end{bmatrix} \quad K_{24}=\begin{bmatrix}-2.75 & -3.75\\ 3.75 & 22.75\end{bmatrix}$$

$$K_{32}=\begin{bmatrix}-12.5 & 7.5\\ 15 & -50\end{bmatrix} \quad K_{33}=\begin{bmatrix}25 & 0\\ 0 & 100\end{bmatrix} \quad K_{34}=\begin{bmatrix}-12.5 & -7.5\\ -15 & -50\end{bmatrix}$$

$$K_{42}=\begin{bmatrix}-2.75 & 3.75\\ -3.75 & 22.75\end{bmatrix} \quad K_{43}=\begin{bmatrix}-12.5 & -15\\ -7.5 & -50\end{bmatrix} \quad K_{44}=\begin{bmatrix}15.25 & -11.25\\ 11.25 & 27.25\end{bmatrix}$$

For element 3:

$$K_{11}=\begin{bmatrix}15.25 & 11.25\\ 11.25 & 27.25\end{bmatrix} \quad K_{15}=\begin{bmatrix}-2.75 & 3.75\\ -3.75 & 22.75\end{bmatrix} \quad K_{13}=\begin{bmatrix}-12.5 & -15\\ -7.5 & -50\end{bmatrix}$$

$$K_{51}=\begin{bmatrix}-2.75 & -3.75\\ 3.75 & 22.75\end{bmatrix} \quad K_{55}=\begin{bmatrix}15.25 & -11.25\\ -11.25 & 27.25\end{bmatrix} \quad K_{53}=\begin{bmatrix}-12.5 & 15\\ 7.5 & -50\end{bmatrix}$$

$$K_{31}=\begin{bmatrix}-12.5 & -7.5\\ -15 & -50\end{bmatrix} \quad K_{35}=\begin{bmatrix}-12.5 & 7.5\\ 15 & -50\end{bmatrix} \quad K_{33}=\begin{bmatrix}25 & 0\\ 0 & 100\end{bmatrix}$$

For element 4:

$$K_{55}=\begin{bmatrix}15.25 & 11.25\\ 11.25 & 27.25\end{bmatrix} \quad K_{54}=\begin{bmatrix}-15.25 & -11.25\\ -11.25 & -27.25\end{bmatrix} \quad K_{53}=\begin{bmatrix}18 & 7.5\\ 15 & 4.5\end{bmatrix}$$

$$K_{45}=\begin{bmatrix}-15.25 & -11.25\\ -11.25 & -27.25\end{bmatrix} \quad K_{44}=\begin{bmatrix}15.25 & 11.25\\ 11.25 & 27.25\end{bmatrix} \quad K_{43}=\begin{bmatrix}-18 & -7.5\\ -15 & -4.5\end{bmatrix}$$

$$K_{35}=\begin{bmatrix}18 & 15\\ 7.5 & 4.5\end{bmatrix} \quad K_{34}=\begin{bmatrix}-18 & -15\\ -7.5 & -4.5\end{bmatrix} \quad K_{33}=\begin{bmatrix}36 & 0\\ 0 & 9\end{bmatrix}$$

Once all the element stiffness matrices have been found the assembly process can begin by summing all the contributions according to the following:

$$K = \begin{bmatrix} \sum K_{11} & \sum K_{12} & \sum K_{13} & \sum K_{14} & \sum K_{15} \\ \sum K_{21} & \sum K_{22} & \sum K_{23} & \sum K_{24} & \sum K_{25} \\ \sum K_{31} & \sum K_{32} & \sum K_{33} & \sum K_{34} & \sum K_{35} \\ \sum K_{41} & \sum K_{42} & \sum K_{43} & \sum K_{44} & \sum K_{45} \\ \sum K_{51} & \sum K_{52} & \sum K_{53} & \sum K_{54} & \sum K_{55} \end{bmatrix} \qquad \textbf{Equation 18}$$

Each term in Equation 19 contributes a 2x2 matrix to the final matrix. Only the nodes that are shared between elements will need to be summed. The global stiffness matrix is shown in Equation 19 which is a 10x10 matrix because there are five nodes each with two degrees of freedom for all the elements:

$$K = (166.67) \cdot \begin{bmatrix} 30.5 & 22.5 & 2.75 & -3.75 & -30.5 & -22.5 & 0 & 0 & -2.75 & 3.75 \\ 22.5 & 54.5 & 3.75 & -22.75 & -22.5 & -54.5 & 0 & 0 & -3.75 & 22.75 \\ 2.75 & 3.75 & 30.5 & 0 & -30.5 & 22.5 & -2.75 & -3.75 & 0 & 0 \\ -3.75 & -22.75 & 0 & 54.5 & 22.5 & -54.5 & 3.75 & 22.75 & 0 & 0 \\ -30.5 & -22.5 & -30.5 & 22.5 & 122 & 0 & -30.5 & -22.5 & 5.5 & 22.5 \\ -22.5 & -54.5 & 22.5 & -54.5 & 0 & 218 & -22.5 & -54.5 & 22.5 & -45.5 \\ 0 & 0 & -2.75 & 3.75 & -30.5 & -22.5 & 30.5 & 0 & -15.25 & -11.25 \\ 0 & 0 & -3.75 & 22.75 & -22.5 & -54.5 & 22.5 & 54.5 & -11.25 & -27.25 \\ -2.75 & -3.75 & 0 & 0 & 5.5 & 22.5 & -15.25 & -11.25 & 30.5 & 0 \\ 3.75 & 22.75 & 0 & 0 & 22.5 & -45.5 & -11.25 & -27.25 & 0 & 54.5 \end{bmatrix} \qquad \textbf{Equation 19}$$

The computation of the nodal displacements, {∂}, is based on Equation 20:

$$\{\partial\} = [K]^{-1}\{Q\} \qquad \textbf{Equation 20}$$

$$\{\partial\} = \begin{Bmatrix} u_3 \\ v_3 \\ u_4 \\ v_4 \\ u_5 \\ v_5 \end{Bmatrix} = \left(\frac{1}{166.67}\right) \cdot [K]^{-1} \begin{Bmatrix} 0 \\ 0 \\ 100 \\ 0 \\ 100 \\ 0 \end{Bmatrix} \qquad \textbf{Equation 21}$$

Because nodes 1 and 2 are fixed, the stiffness matrix is reduced to a 6x6 matrix. The other unknown displacements are computed with the previous equation. The reduced inverted stiffness matrix is shown below as:

$$[K]^{-1} = \begin{bmatrix} 0.012 & 0.013 & 0.01 & 0.03 & 0.004 & 0.023 \\ 0.001 & 0.029 & 0 & 0.056 & -0.001 & 0.052 \\ 0.014 & 0.065 & 0.052 & 0.142 & 0.028 & 0.13 \\ 0 & 0.039 & -0.014 & 0.096 & 0 & 0.077 \\ 0.004 & 0.023 & 0.019 & 0.06 & 0.047 & 0.051 \\ -0.001 & 0.052 & 0 & 0.111 & 0.003 & 0.118 \end{bmatrix}$$

Equation 22

$$\{\partial\} = \begin{Bmatrix} 1.397 \\ -0.139 \\ 7.988 \\ -1.354 \\ 6.624 \\ 0.279 \end{Bmatrix} \cdot \frac{1}{166.67} in.$$

Equation 23

This problem demonstrates the basic method of finite element analysis through solving a basic a simple structural analysis example. Changing the problem from triangular elements to hexahedral adds more complexity and the matrix size is much larger because each element now has eight nodes instead of three. So for this problem to be changed to hexahedral elements the number of nodes per element would change from three to eight and since there were two degrees of freedom, each element would contribute a 16x16 matrix to the global stiffness matrix instead of a 6x6 matrix as shown with the triangular elements.

2.7. Sparse Matrices

Sparse matrices are matrices that have few nonzero terms compared to zero terms, usually situated close to the diagonal of the matrix. Sparse matrices arise in many scientific/engineering applications. Some of the applications are structural analysis, networks, and fluid-flow. In Equation 24,

$$\{f\} = [A]\{x\} \qquad \textbf{Equation 24}$$

the "A" matrix is usually a sparse matrix in the application of Finite Element Analysis as seen in Section 3.6.4 and is also the main operation in many different iterative solvers such as Preconditioned Conjugate Gradient method. The benefit of sparse matrices is that only the nonzero terms and their locations need to be saved. There are several ways to store a sparse matrix as discussed in the next section.

2.7.1. Storage Schemes

The first and most obvious is the use of three arrays one for the column, another for the row, and finally one for the data. The types of these arrays are integer, integer, and double, respectively in most cases. Shown in Table 3 is an example of the column-row technique called coordinate format. This is shown in Table 2 using zero indexing [18]. The coordinate format shown in Table 3 is a format where the row and column locations of only nonzeros are stored. For example, the term "3.0" found in row one and column one of Table 2 (using zero indexing) is represented by a "1" in i-index, a "1" in j-index, and a "3.0" in value [all in location two of there respective

arrays]. All of the other non-zero elements in Table 2 can be represented by the three arrays in Table 3 in a similar manner.

Table 2 Example Matrix for Sparse Storage Schemes

1.0	0	0	2.0	0	0
0	3.0	0	0	4.0	0
0	0	5.0	0	0	0
6.0	0	0	7.0	0	8.0

Table 3 Example Sparse Matrix Coordinate Format Storage

i-index = (	3,	1,	0,	3,	2,	0,	1,	3),
j-index =(	5,	1,	3,	3,	2,	0,	4,	0),
value = (	8.0,	3.0,	2.0,	7.0,	5.0,	1.0,	4.0,	6.0)

Next is a storage-by-row technique called compressed row storage (CRS) which also consists of three arrays of the same type as the coordinate format. The only difference is that the row array is compressed to only contain pointers to the first non-zero data entry in each row contained in the data array. The only array that changes compared to the coordinate format shown above is the i-pointer array. The i-pointer for CRS format points to the location within the j-index array that is the first nonzero of each row. For example, the second term in i-pointer points to value “1” (remember zero indexing) in j-index so one would know when the next row began. Table 4 shows this technique. There also is a technique called compressed column storage that is implemented in a similar manner, except the column array is compressed rather than the row array as in the CRS format.

Table 4 Example Sparse Matrix CRS Format Storage

```
i-pointer = ( 0, 2, 4, 5, 8 ),
j-index = (  0,   3,   1,   4,   2,   0,   3,   5 ),
 value  = ( 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0 )
```

The last method is called variable block row format (VBR) which is used when there are large portions of the matrix that can be divided into smaller dense matrices (Table 5). This method has six arrays to hold the various information. The first array (row-pointer) is an integer array that holds pointers to the boundaries of the block rows (Table 6). The next array (column pointer) is also an integer array that holds pointers to the boundaries of the block column. Another array (value) is an array of doubles that contains the block entries of the matrix. Another integer array (index) holds the pointers to the beginning of each block entry stored in the value array. An integer array (block-index) contains the block column indices of the block entries (Table 7) in the matrix. The final array (block-pointer) contains pointers to the beginning of each block row in block-index and the value arrays. This method is the most difficult to implement and is shown graphically in Tables 5-6.

Table 5 Example VBR Matrix

```
     0    1     2    3     4      5     6    7   8
  +----------+----------------+------+-----------+
0 | 1.0  2.0 |                |  3.0 |           |
1 | 4.0  5.0 |                |  6.0 |           |
  +----------+----------------+------+-----------+
2 |          | 7.0  8.0   9.0 | 10.0 |           |
  +----------+----------------+------+-----------+
3 |          |                | 11.0 | 12.0 13.0 |
4 |          |                | 14.0 | 15.0 16.0 |
5 |          |                | 17.0 | 18.0 19.0 |
  +----------+----------------+------+-----------+
6
```

Table 6 Example VBR Storage Matrix

```
Row pointer  = (0, 2, 3, 6)
Column pointer  = (0, 2, 5, 6, 8)
Block pointer  = (0, 2, 4, 6)
Block index = (0, 2, 1, 2, 2, 3)
index  = (0, 4, 6, 9, 10, 13, 19)
value   = (1.0, 4.0, 2.0, 5.0, 3.0, 6.0, 7.0, 8.0, 9.0,
         10.0, 11.0, 14.0, 17.0, 12.0, 15.0, 18.0,
         13.0, 16.0, 19.0)
```

Table 7 Block Representation of Matrix

```
    0    1    2    3    4
  +----+----+----+----+
0 | b0 |    | b1 |    |
  +----+----+----+----+
1 |    | b2 | b3 |    |
  +----+----+----+----+
2 |    |    | b4 | b5 |
  +----+----+----+----+
3
```

To understand the VBR technique of storing sparse matrices let us look at the example of accessing the block row 1 in Table 5. First a lookup in the block pointer array and is needed to see where block row one appears in this case it is *block pointer* [1] = 2. This indicates that block two (b2) in Table 7 contains the first nonzero block from block row one and that it is from block column one as indicated by *block index* [*block pointer* [1]] =1. Second the *block pointer* [1] also indexes into *index*. That is *index* [*block pointer* [1]] = *index* [2] 6 = which points to *value* [6]. This is equal to *value* [*index* [*block pointer* [1]] =*value* [*index* [2]]. Where 6 is the location in *value* where the element, 7.0, is located.

3. RELATED WORK

There are various other works that have studied similar material relevant to this work. This section is organized by topic. First is some related work on the performance of Sparse Matrices, then finite element research, and finally performance studies on the Itanium 2.

3.1. Performance of Sparse Matrices

The performance of sparse matrix operations depends primarily on the memory hierarchy of the microarchitecture. Taylor [19] has studied the performance of these operations and has concluded that various memory organizations maximize the performance of sparse-matrix operations. Sparse matrices are stored in compressed form in which one data structure points to the position of the matrix data in another data structure. This form of indirect addressing allows the cache to be very effective for storing the data because of the spatial-temporal locality of the accesses. Spatial locality means that if a location in memory is accessed the datum that is close to that location is likely to be used in the near future. The term temporal locality states that if a location in memory/cache has been accessed recently it is likely to be reused in the near future (i.e., looping). The study by Taylor concluded that to maximize the performance of these sparse matrix problems, the cache organization would have to possess the following characteristics:

- Direct-Mapped Cache
- Cache Size of at least 1K words or 8K bytes
- Write-back Policy

- Pipeline depth for write equal to 2 (to allow for one-cycle minimal write for multi-word block)
- Block size equal to 16 words
- Invalidate data on a read
- 2 interface ports -1 read, 1 write
- 2 phase-clock to allow for simultaneous read and write

Another study by Temam and Jalby describes the performance of sparse algorithms on caches [20]. They concluded that cache size and the bandwidth of the matrix are closely dependent. When the bandwidth of the matrix is smaller than the cache size, spatial and temporal locality is well exploited with their scientific application. On the other hand, when the bandwidth is greater than the cache size, self and cross-interference degrade the reuse of the vector *x*, meaning the data within *x* gets overwritten by the vector *x* or the data within the A matrix before it gets reused. Also they report a performance increase when the line size is sufficiently large (around 128 bytes), exploiting the potential locality of the vector *x* especially in 3-dimensional finite element problems such as cube3 where the vector *x* is used more than in the 2-dimensional case.

3.2. Finite Element Research

Finite element workloads have been used as the basis for performance studies in other related works. The finite element workloads have mainly been studied at the multiprocessor level as of late because they can scale on multiple processors due the intrinsic value of matrix operations, such as [21] in which they design a high performance, high efficiency multi-processor computing engine for dynamic finite element analysis. In [22] they use a finite

element workload called DYFESM, which is a structural dynamics code which implements a finite element model using 8 stress and 5 displacement degrees of freedom per node. Within this problem they characterized the dominate loops and correlated the loops to the loop-based Livermoore Fortran Kernels Benchmark. They found that the dominate subroutine was for a preconditioned conjugate gradient solver in which it was performing a matrix-vector multiply. In [23] they conclude that the performance of the 3-D TGM finite element solver is directly related to the linear system solver, in which they found the conjugate gradient algorithm to be the most optimal which uses the same matrix-vector multiply found in [22]. To fully utilize the resources of an architecture running a application that contains the matrix-vector multiply found in these finite element workloads Taylor et al. propose an efficient scheme for storing sparse matrices and through the use of added hardware to the architecture to help in efficiently executing the proposed data structure. They demonstrated a 96% utilization of the floating-point units [24]. Also Vuduc et al. of UC Berkley discuss performance optimizations and bounds for a sparse matrix-vector multiply in which the results suggest that future performance improvements will come from two sources: 1) consideration of higher-level matrix structures, and 2) optimizing kernels with more opportunity for data reuse through higher level techniques [25].

3.3. Itanium 2 Performance Studies

The Itanium 2 microarchitecture is a newer architecture in which the basis for design was to maximize the performance of scientific applications that are mainly floating-point workloads. As seen by Purkayastha et al. [26] the floating-point performance of the Linpack benchmark on the Itanium 2 dominated the studies of modern 64-bit architectures (AMD Opteron, Apple G5) in which they uses a highly optimized Goto BLAS library. Also within the performance study in [26], they ran benchmarks of a 3d finite element code (MGF) and it also performed the best on the Itanium. Griem et al. also studied the Itanium 2 and propose a synthetic workload consisting of a sparse matrix-vector multiply to determine various characteristics of various other architectures besides the Itanium 2 [3]. The results of their studies state that because of the inability of the Itanium 2's L1 data cache to store floating-point variables some delays occur due to the register spills of larger working sets. But the Itanium 2 was able to hide memory latencies using a large register set and deep explicit prefetch queues.

Although many benchmarks contain some form of a finite element problem no current research has been found that focuses on uni-processor performance characterization of these finite element workloads that does not discuss the performance of the sparse matrix-vector multiply. Our research looks further into performance analysis of the Itanium 2 microarchitecture and how maximum performance may be achieved.

4. THESIS PROBLEM

The motivation for the work is to understand how this finite element application, “Cube3”, actually goes about creating and solving a system of equations and the performance characteristics associated with the application. By studying this application on the Itanium 2, which has the potential to perform well due to the large cache hierarchy and functional units, we hope to pin-point a performance bottleneck that can easily be identified and modified, be it hardware or software, which will lead to a performance improvement. Also, by studying “Cube3” on the Itanium 2 some micro-architectural characteristics can be used in future work of creating an analytic model used in a multi-processor model used at Sandia National Laboratories. This analytic model will be used in the future to help in the decision of what type of processor will have the best performance benefits used in the next generation of supercomputers.

5. WORKLOAD

National laboratories have some of the largest and most costly supercomputers ever designed and built. Because of the high cost of design and construction some studies of the workload executed on these computers are needed to understand how they spend most of their execution time. Bradley et al. [27] have studied the types of applications running on their system and have shown that the greatest amount of computing time has been in scientific workloads such as Finite element and physics based applications. Most of these programs use matrix operations which solve large numbers of differential equations. Therefore, a representative workload should include matrix assembly and various techniques of solving these matrices.

A workload that is intended to be representative of the many scientific codes was chosen as a basis of all scientific workloads at Sandia National Laboratories. This workload is a Finite Element problem that allows the user to specify various different techniques to solve the problem. The workload is described in detail in the next section. The overall study of this workload is to better understand the workloads on National Laboratories computers and also to provide information on what type of microarchitecture will maximize the performance of these types of applications.

5.1. The “Cube3” Application

The application chosen for this performance study is a Finite Element test problem, called “Cube3,” written and provided by Alan Williams of Sandia

National Laboratories in Albuquerque, New Mexico. “Cube3” is a test problem of a finite element interface (FEI) written as an abstraction layer between engineering/scientific software and math solvers [28]. The FEI is a linear system assembly library used for assembling sparse matrices in applications that use unstructured meshes. The finite element interface provides a layer of software to allow applications the ability to switch between various solvers without changing application code. The various linear solvers that can be used by the finite element interface are Trilinos, PETSc, FETI-DP, HYPRE, SPOOLES, Prometheus, and others. The “Cube3” test problem is an arbitrary problem written to test the performance of linear system assembly and solution. Although “Cube3” only mimics a finite-element problem it was chosen to represent Sandia’s workloads because of its simplicity and the genuineness of the assembly and solve processes within the “Cube3” workload. The test problem mimics a finite element application because it imitates the data produced from an application operating on a mesh of 8-node hexahedral elements (shown in Figure 18). The nodes are represented at each corner of the hexahedral element.

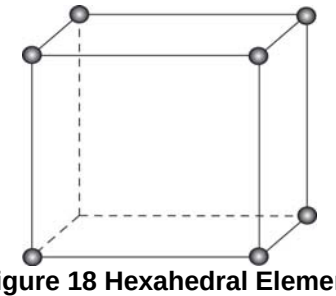


Figure 18 Hexahedral Element

The number of elements can be varied based on width, depth, and degrees of freedom based upon an input file used by the “Cube3” test problem which can be found in the Appendix. These degrees of freedom represent physical attributes such as pressure, temperature, velocity, or any other physical phenomenon that one wishes to compute. The number of elements is calculated by (width) · (width) · (depth). Figure 19 shows a graphical representation of a width equal to three and a depth of one.

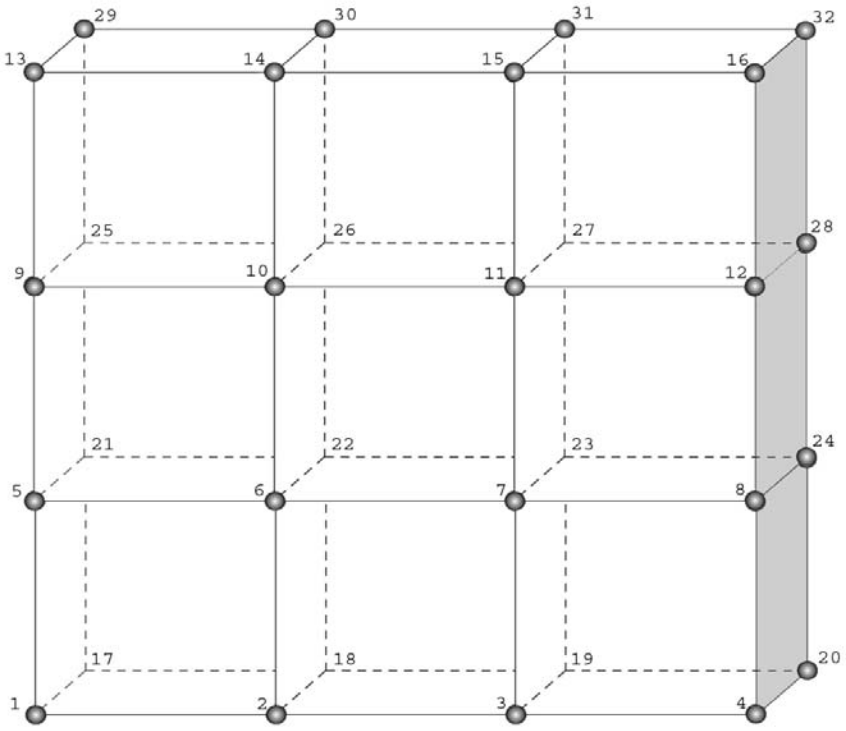


Figure 19 3x3x1 Hexahedral Elements

The number of equations in the linear system is equal to the number of nodes multiplied by the number of degrees of freedom per node where the number of nodes is equal to (“width”+1) · (“width”+1) · (“depth”+1). As stated above, each node can be defined to have a specified number of degrees of

freedom. This test problem usually runs on large multi-processor systems using a MPI (Message Passing Interface) to achieve better performance. In the parallel setting this problem is spilt across the depth of the cube. In other words the depth is divided by the number of processors specified during runtime. The shared nodes, which are nodes at the boundary of where the problem is split, appear on both of the processors. The “Cube3” workload provides a good test problem of the various underlying solvers and the how the problem is assembled, which is a major reason for use as a representative workload.

6. METHODOLOGY

The performance study of the representative workload, “Cube3,” is determined first by establishing a problem size that will be studied. The problem size should be chosen to maximize the utilization of the cache under study, in this case the cache of the Itanium 2. The cache statistics collected by the performance-monitoring unit decide what problem size to study since the data of the matrix and vectors of the Finite Element problem need to be held in the memory hierarchy to maximize the performance of the application. By executing “Cube3” with various problem sizes, we identify a size sufficiently large enough such that the working set can not be fully held in cache. The first step in varying problem size is to determine a maximum number of equations that can successfully complete on the Itanium 2 microarchitecture. Once a maximum value is found, then problem size studies are conducted based on varying the problem size via the input file to “Cube3”. These values are calculated by Equation 25 described in Section 5.1:

$$equations = (w+1)\cdot(w+1)\cdot(d+1)\cdot(dof) \quad \textbf{Equation 25}$$

After determining an appropriate problem size, various studies during runtime are performed to understand more about the “Cube3” application. These studies include studying the various phases throughout the execution of “Cube3.” This is conducted through the use of call graph and profile results in correlation to the interval data collected throughout the execution of “Cube3.” Vtune is used to generate the call graph data and profile data. The

interval IPC graphs are achieved through the use of the PERFMON software by outputting the number of cycles completed every one million instructions and then resetting the counter so that it is no longer a cumulative counter. This will give a list of the number of CPU cycles every one million instructions. By graphing the interval IPC data the phases with poor performance can be located by low segments of the IPC.

Experiments are then conducted on how varying the shape of an object that “Cube3” is studying will effect the various phases of execution. By holding the number of equations constant for a few different shapes (i.e. beam, cube, or thin plate) various performance attributes can be studied. Instruction mix of the shape under study is also collected to help to categorize the application as an integer or floating-point application and how the shape effects the instruction mix. In addition, cache statistics are studied to understand if the shape of object changes the cache miss rates. These studies help in understanding the application as the problem changes as it does in a real world application.

Once an overall understanding of the “Cube3” application is achieved a detailed performance study is conducted based on the problem size studies. The performance study is conducted using stall counters within the Itanium 2 microarchitecture. To determine where the stalls in the pipeline occur Jarp [29] has determined a methodology for performing bubble analysis on the Itanium microarchitectures using the hardware performance counters. This

methodology provides a top-down approach to identify and understand bottlenecks in the micro-architecture. This bubble analysis methodology allows a user to determine if the major cause of performance degradation is due to data cache stalls, branch misprediction, instruction miss stalls, floating-point unit stalls, general register scoreboarding, or front-end flushes. Through this global stall analysis, we can identify problem areas in the micro-architecture.

The methodology of this bubble analysis examines all the stall contributions in the pipeline and then allows for further exploration of those major stall contributions. The first step in determining a bottleneck using Jarp's approach is to consider all stall cycles and determine the cause of the stall. The Itanium 2 micro-architecture has two major components, the front-end (instruction decode and dispatch) and the back-end (execution). Within the Itanium 2 microarchitecture there are five main units which can cause stalls:

- Back-end stalls caused by an exception/interruption or branch misprediction flush (PMU event *be_flush_bubble_all*)
- Back-end stalls due to Level 1 data cache or Floating Point Unit (*be_l1d_fpu_bubble_all)*
- Back-end stalls due to the execution stage of the pipeline (*be_exe_bubble_all*)

- Back-end stalls due to the register stack engine (*be_rse_bubble_all)*
- Back-end stalls due to the Front-End (*be_exe_bubble_fe*).

Within each of these categories are vary sub-counters to provide a finer granularity of the stall causes. The stall counters and their sub-counters are shown in Table 10. The major contributions of the second sub-counter of "Cube3" will be described in detail in the results section.

Table 8 Itanium 2 Stall Counters

Total Stall Counter	Sub-Counter	Second Sub-Counter
Back_end_bubble_all	Be_flush_bubble_all	Be_flush_bubble_bru Be_flush_bubble_xpn
	Be_L1d_fpu_bubble_all	Be_L1d_fpu_bubble_l1d Be_L1d_fpu_bubble_l1d_dcurecir Be_L1d_fpu_bubble_L1d_tlb Be_L1d_fpu_bubble_L1d_stbufrecir Be_L1d_fpu_bubble_L1d_fullstbuf Be_L1d_fpu_bubble_L1d_L2bpress Be_L1d_fpu_bubble_fpu
	Be_exe_bubble_all	Be_exe_bubble_grall Be_exe_bubble_grgr Be_exe_bubble_frall Be_exe_arcr_pr_cancel_bank
	Be_rse_bubble_all	Be_rse_bubble_overflow Be_rse_bubble_underflow
	Back_end_bubble_fe	none

"Cube3" also has the ability to change the various solver algorithms used to solve the linear system through the use of the AztecOO package within Trilinos. By varying the different methods of solving the "Cube3" problem, a comparison of the solvers is performed to see the benefits of

each. This could lead to performance improvements of one or all due to the various solution methods performance characteristics. This is done through the use of the call graph/ profile data and the use of the interval data. Also, “Cube3” has the ability to change various storage methods (Crs and Vbr) through the input file as well as changing from the use of no preconditioner to the use of various preconditioners. A comparative analysis of the various methods will help to understand the different characteristics of each. This study will also use the interval IPC data to understand the changes in the phases and to see the performance benefits of each.

The Itanium 2 is a VLIW microarchitecture which is an in-order core which relies on the compiler to perform all scheduling of parallel instructions. This means that the performance of applications relies mainly on the compiler in use. A comparative analysis is needed to see which compiler performs the best on the Itanium 2. The two compilers available on our Itanium 2 are the Intel Icc compiler and the Gnu Gcc compiler.

Throughout the performance analysis of “Cube3” a problem section of the code is determined within the solve phase of execution. Some optimizations to the code were implemented in conjunction with some compiler optimizations to see if performance gains are achievable within the compilers in use. The results of these studies are also presented in the results section.

7. RESULTS

The results section of this work present the work necessary to understand “Cube3” as well as understand the performance of it. Section 2 of the results helps in the understanding of “Cube3” whereas Sections 3 through 5 are focused on the performance associated with “Cube3” on the Itanium 2.

7.1. Problem Size

We began the problem size studies on the Itanium 2 microarchitecture by choosing a large cube size of width 72, depth 72 and six degrees of freedom which calculated to a total number of equations equal to 2,334,102. With this problem size the Itanium 2 ran out of memory so the study was backed down to a maximum number of equations of about half the size of 72x72x6 which was around one million equations. With a maximum problem size of one million equations the problem size was great enough to overwhelm the cache. For example a 100x100x1 problem size contains about 27 million nonzeros which overwhelmed the L3 cache of size 1.5 Meg. This number was large enough to alleviate any cold start misses and stress the cache hierarchy sufficiently. Various configurations of width and depth and degrees of freedom were calculated to have a maximum of one million equations. The only study that showed any conclusive data was the study of varying width with constant depth and degrees of freedom. Table 8 shows the problem sizes executed to get the plot found in Figure 20. The rest of the problem size studies and plots are contained in the Appendix.

In Figure 20, one can see that the maximum IPC correlates with the minimization of the cache miss rates in the level 3 cache (Level 1 miss rates are not shown due to the fact that floating-point data bypasses the L1 cache and also because the PMU does not have a counter for L1 data). The maximum performance is at a point when the problem size is 300x1x1 which equates to 181,202 equations. From that point on the caches start having conflict misses due to the fact that the problem size was too large to be entirely held in cache and therefore needed data was getting overwritten by other data which increases the miss rate leading to performance degradation.

The next study studied various problem sizes that have a maximum of around 180,000 equations because of the maximum IPC point was when there were around 180,000 equations in the study of problem size above. This study had inconclusive results (results shown in Appendix) because of the fact that all the studies of varying shape showed relatively the same statistics for IPC, L2 miss rates, and L3 miss rates as the problem shape was varied. The problem sizes that were studied were of sizes 300x1x1 (181,202 equations), 55x55x1 (175,616 equations), 45,000x1x1 (180,004 equations) due to the performance characteristics of various problem shapes shown in section 7.2.2.

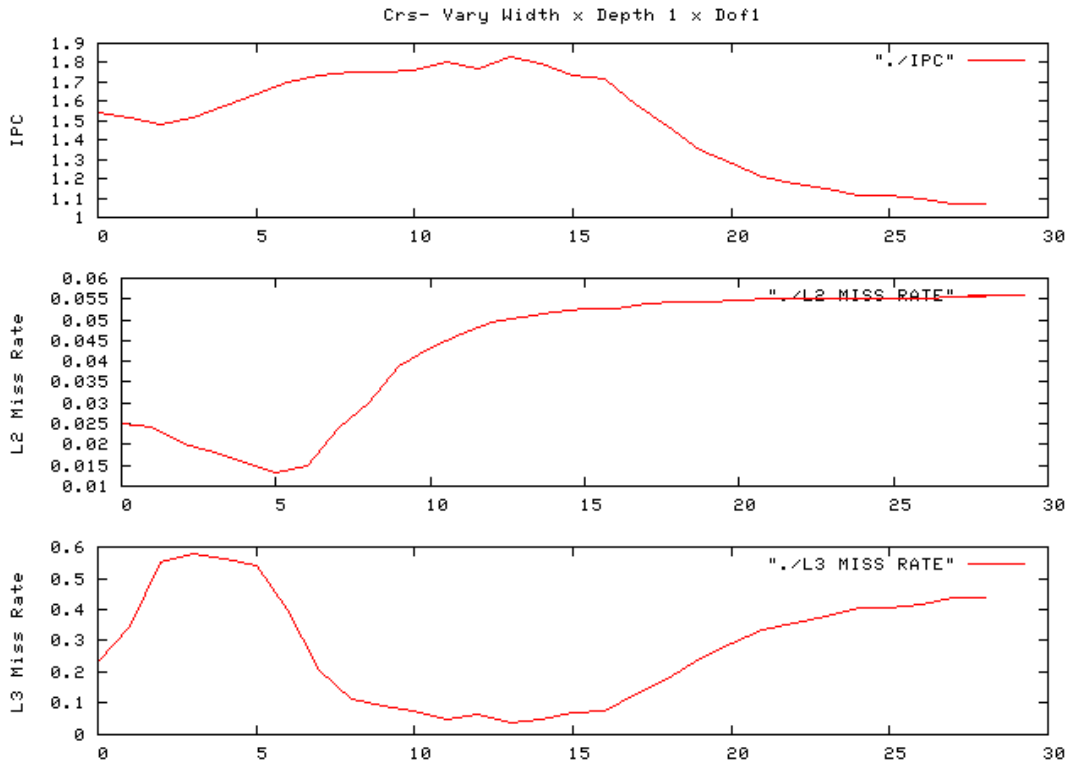


Figure 20 Vary Width CRS -IPC, L2&L3 Cache Stats

Table 9 Problem Sizes for Figure 20

Vary Width							
Depth	Width	Dof	Equations	Depth	Width	Dof	Equations
1	25	1	1352	1	400	1	321602
1	50	1	5202	1	425	1	362952
1	75	1	11552	1	450	1	406802
1	100	1	20402	1	475	1	453152
1	125	1	31752	1	500	1	502002
1	150	1	45602	1	525	1	553352
1	175	1	61952	1	550	1	607202
1	200	1	80802	1	575	1	663552
1	225	1	102152	1	600	1	722402
1	250	1	126002	1	625	1	783752
1	275	1	152352	1	650	1	847602
1	300	1	181202	1	675	1	913952
1	325	1	212552	1	700	1	982802
1	350	1	246402	1	725	1	1054152
1	375	1	282752	1	750	1	1128002

7.2. Runtime Description

The performance analysis of the “Cube3” workload depends greatly on the understanding of what occurs throughout the execution of “Cube3.” This section provides some incite to what is happening throughout the execution of “Cube3.” This section also shows where the major performance degradation of “Cube3” occurs.

7.2.1. Cube3 Phases

First, to understand the workload itself we generated the call graph using Vtune. The call graph shows the various functions that “Cube3” calls from the main procedure (Figure 21). There are twelve primary functions that the main function calls along with a few initialization functions that create a matrix graph and connectivity lists. Figure 21 only shows the first level of function calls due to the complexity and abundance of function calls within “Cube3”. Also shown is the three functions where “Cube3” spends most of its execution time (insert, sum in, and multiply).

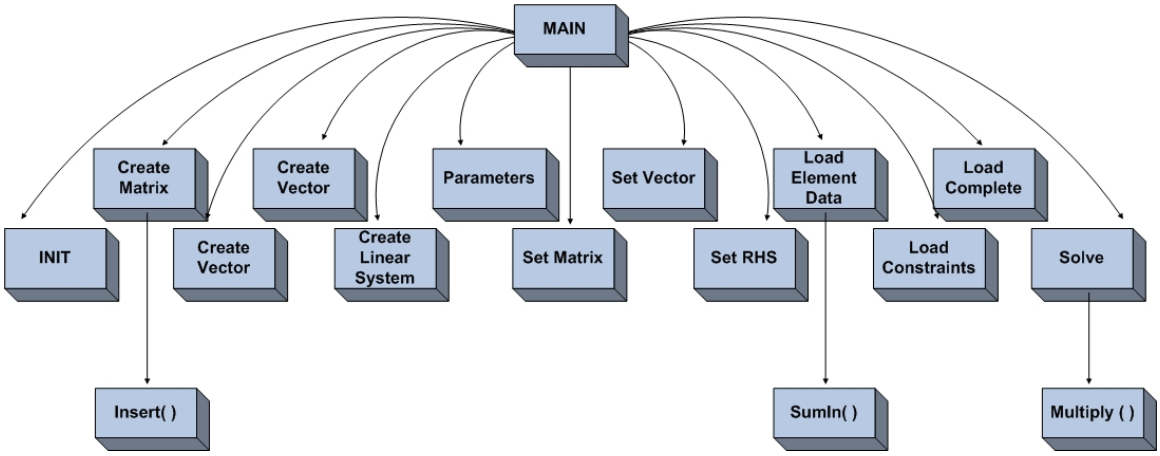


Figure 21 Cube3 Call graph

IPC interval data shows the “Cube3” phases of execution as seen in Figure 22. This figure shows the execution of a 55x55x1 problem size. We chose this size because the various phases are very evident throughout the execution. Figure 20 shows a low IPC execution phase toward the end of execution. To find out what part of the code is causing the performance degradation Vtune allows the user to map the call graph to the source code.

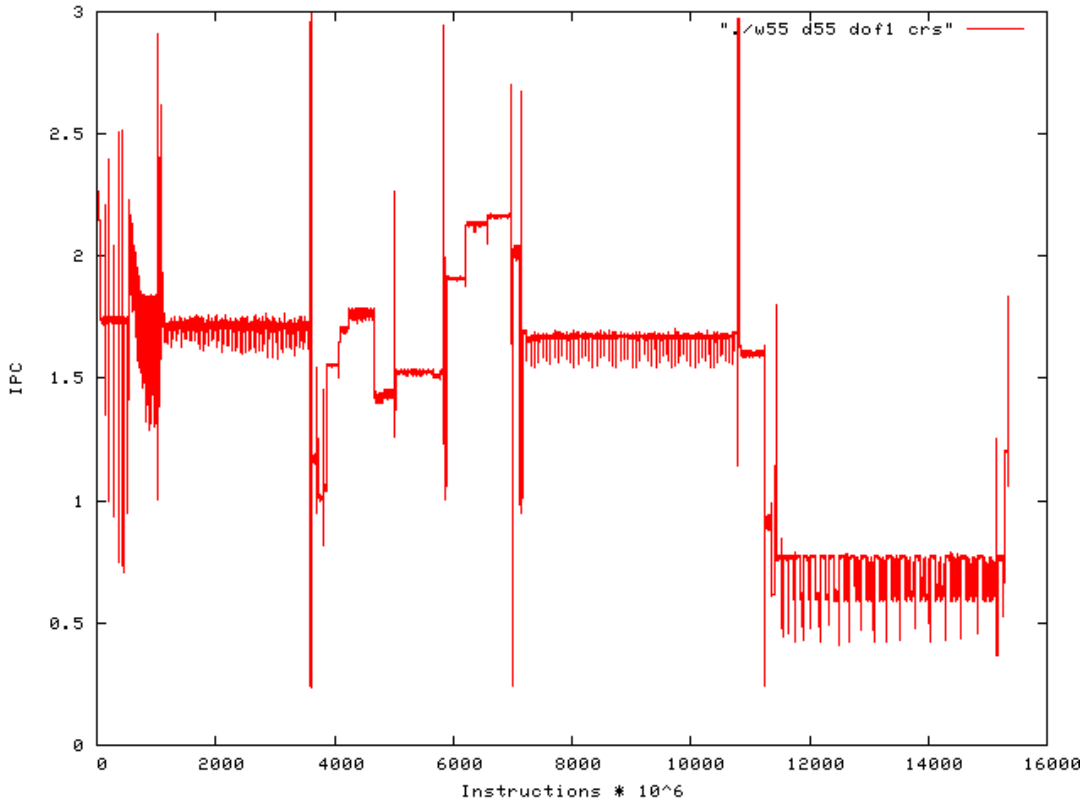


Figure 22 55x55x1 Interval IPC

The mapping of the call graph of Figure 21 and the execution data of Figure 22 is shown by Figure 23. There are three primary functions in which

“Cube3” workload spends most of its time: Create Matrix, Load Element Data, and Solve. The poorest performing is obviously the solve phase, which will be discussed later in the compiler optimization section.

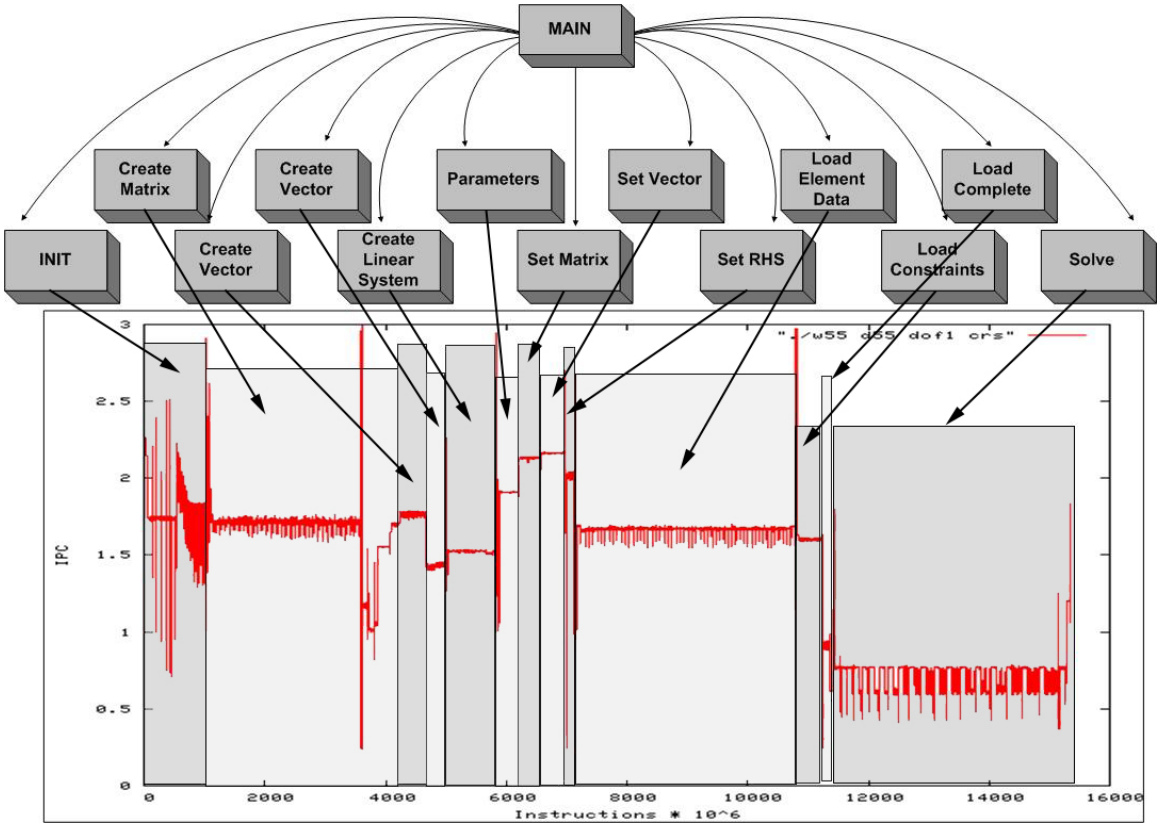


Figure 23 Call graph Mapping to Interval IPC

7.2.2. Varying The Shape of Cube3 Problem

These previous graphs are of a “Cube3” with CRS storage method using a GMRES solver. The following figures contained data generated by “Cube3” using a GMRES solver as we vary the problem shape and the storage method (CRS, VBR). In addition to Figure 22 (55x55x1), are two

graphs of which the shape has been varied from a cube to a beam to a plate. Figure 24 uses a beam (1x45000x1) and Figure 25 uses a plate (300x1x1).

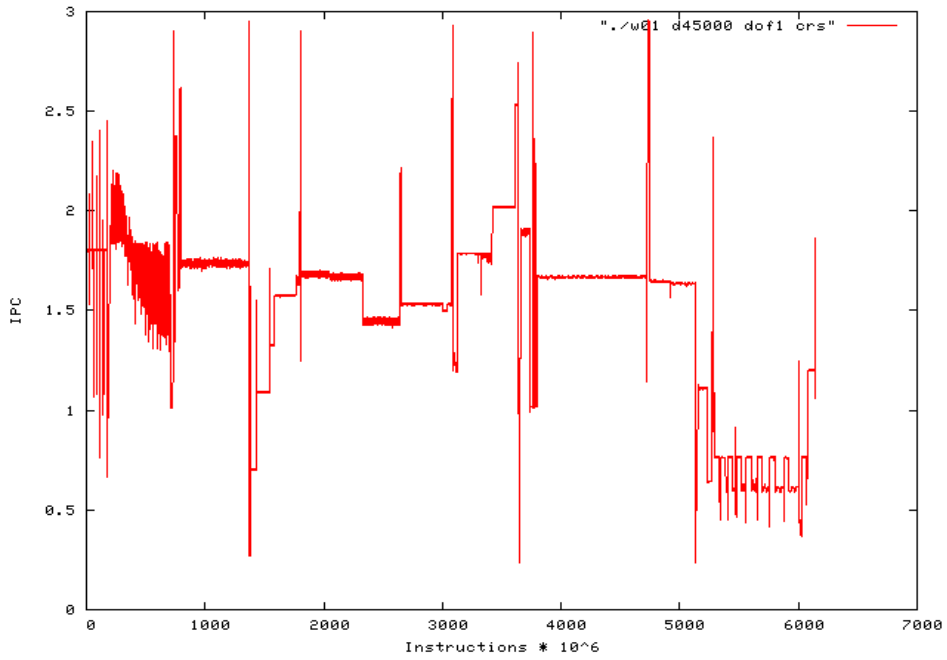


Figure 24 1x45000 Interval IPC

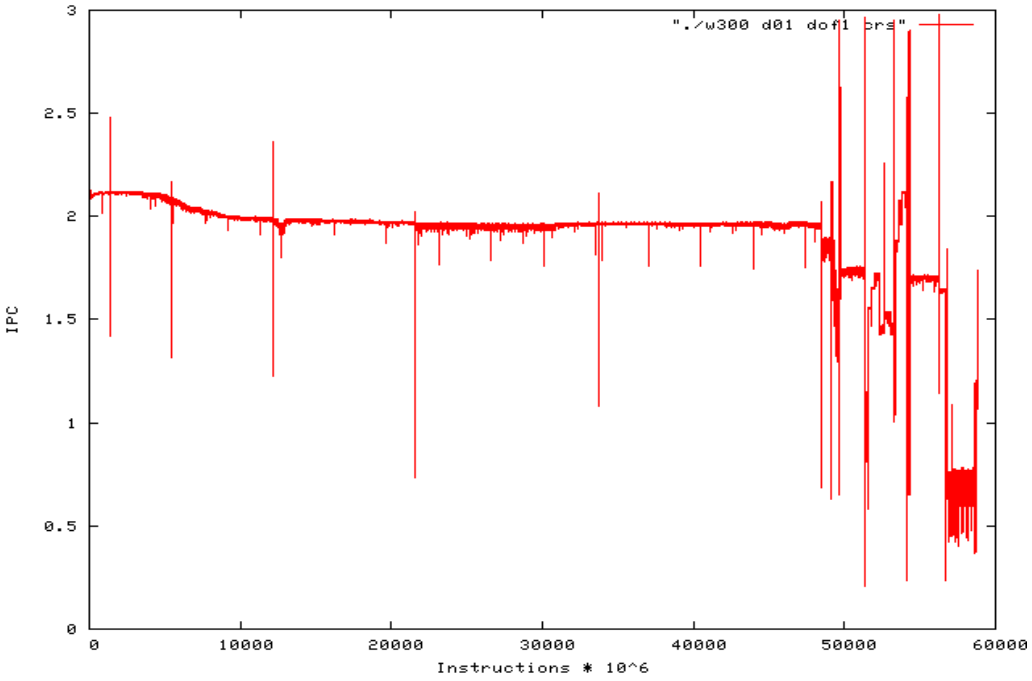


Figure 25 300x1x1 Interval IPC

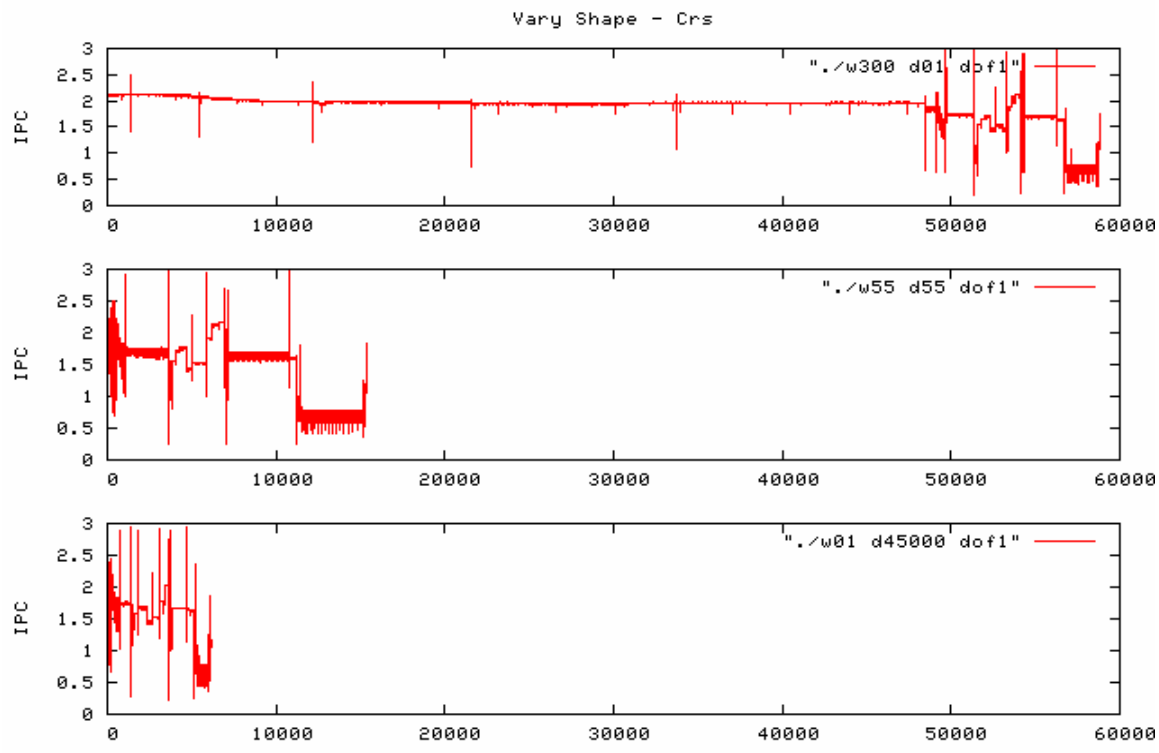


Figure 26 Vary Shapes - Interval IPC

The relative performance characteristics change depending on the shape of the problem due to the connectivity of the nodes and elements and the number of the nonzeros within the matrix. The 300x1x1 problem size seemed to utilize the cache the best in the results shown in Figure 20 in the previous section but studying the interval data it is clearly taking a lot longer than the other two problem shapes as evidenced by Figure 26. However, by studying the interval data graph and the call graph it seems that the time is spent in or before the creation of the matrix execution phase. This appears to be an anomaly and after coordinating with the author, we believe it is a problem with the code. The other two graphs of 55x55x1 and 1x45000x1 appear to have

the same execution phases but the beam (1x45000x1) completes much faster than the cube (55x55x1) due to the fact that the beam has more number of nonzeros in its “A” matrix. Talking further with the author of “Cube3” the actual performance depends on the number of nonzeros in the matrix and not the shape. Figure 26 shows the three shapes with the same x-axis to show the time spent completing each problem shape. Recall that these three problem shapes have approximately the same number of equations meaning the “A” matrices are about the same dimension but have different numbers of nonzeros.

7.2.3. Varying Storage Techniques

Section 3.7 described various methods of storing sparse matrices (Crs and Vbr). By varying the storage method from compressed row storage to variable block row techniques the performance can be seen in Figure 27 to affect the load element data phase of the “Cube3” workload. The variable block row storage method has poor performance for these sizes mainly because of the complexity involved in retrieving the data from the various arrays presented in Section 3.7. For “Cube3” the data within the “A” matrix is accessed row-wise and that is why the compressed row storage method is the most popular implementation of sparse matrix data structures because the data is stored row-wise.

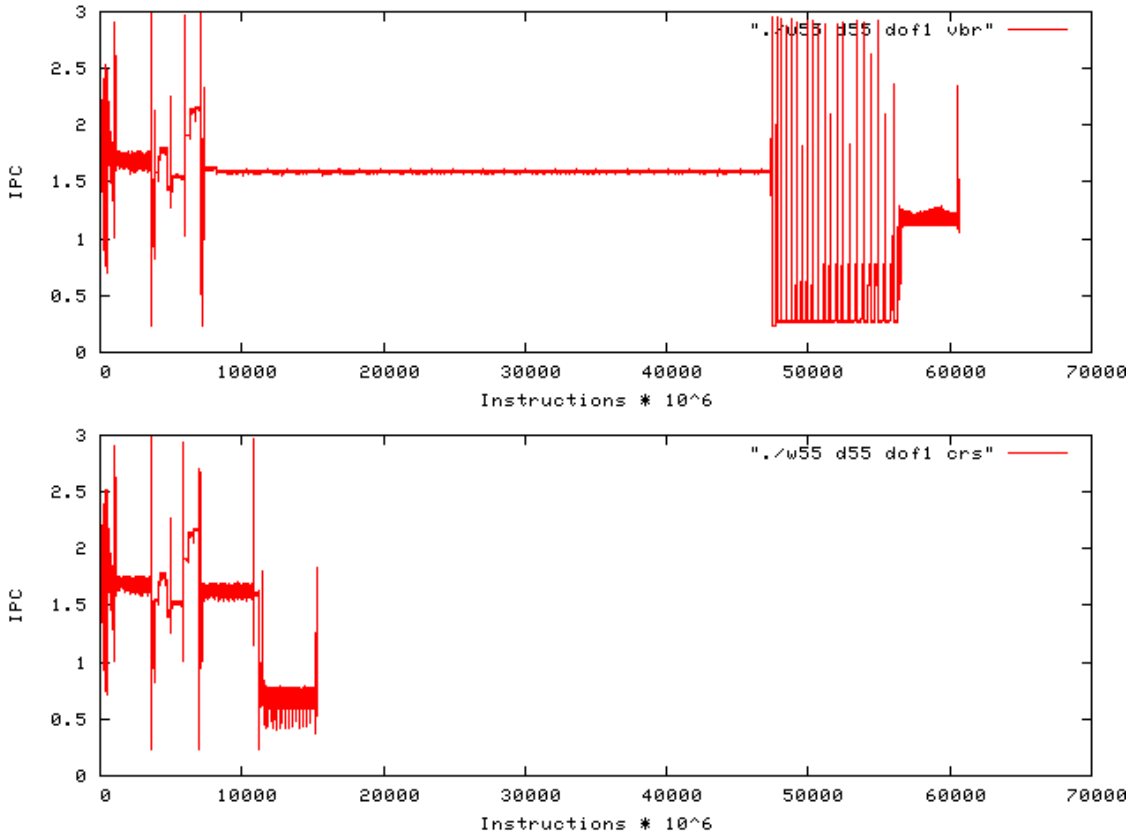
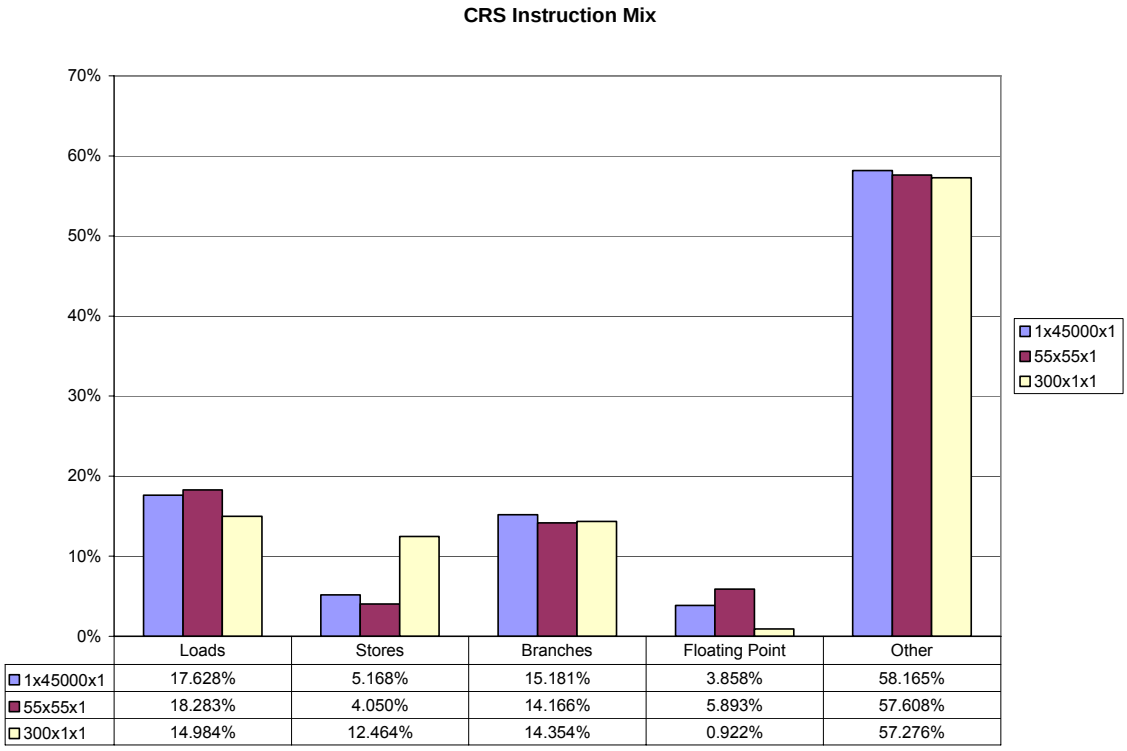


Figure 27 Crs and Vbr Methods 55x55x1 with Gmres Solver

7.2.4. Runtime Statistics

The instruction mix for the compress row storage method is shown in Figure 28. Due to the large amounts of data being manipulated the number of loads accounts for around 20 percent of the overall instruction mix. Also as can be seen from this data as the number of nonzeros increases (1x45000x1 to 55x55x1) the percentage of floating-point operations increases. So as the "A" matrix becomes more dense the more floating-point intensive the application becomes.



	Loads	Stores	Branches	Floating Point	Other
1x45000x1	17.628%	5.168%	15.181%	3.858%	58.165%
55x55x1	18.283%	4.050%	14.166%	5.893%	57.608%
300x1x1	14.984%	12.464%	14.354%	0.922%	57.276%

Figure 28 Crs Instruction Mix

The cache on the Itanium 2 should provide the best source of performance improvement over other architectures due to the large sizes of the caches. But to fully utilize the cache a problem size and shape must be chosen that stresses the cache (i.e. does not fit in the cache). This makes performance analysis difficult because depending on what a scientist is simulating, the problem size and shape changes. Table 9 shows the cache statistics for the three problem sizes. As shown in the table, 300x1x1 allows the cache to perform the best because the matrix creation process seems to be stuck allowing the cache miss rates to increase because it is accessing the same data. The variable block row storage method results are very similar to

the statistics found in Figure 28 and Table 9 and thus will not be shown here but can be found in the Appendix. Studying Table 10 helps to explain why the IPC is lower for the other problem sizes as compared to that of the 300x1x1 problem size. The IPC is lower because the miss rates are so high meaning that for most of the data being access a latency of 12+ cycles is encountered because the data has to be brought in from memory. The cache statistics reiterates that the problem size of 300x1x1 has an error as stated before and the other two problem sizes still do not take advantage of the caches due to their high L3 miss rates. As a conclusion some future work still needs to be done to exercise the caches better which will be discussed the future work section.

Table 10 Crs Cache Statistics

Cache Statistics			
Crs	**w01_d45000_dof1**	**w55_d55_dof1**	**w300_d01_dof1**
L1I miss rate	3.33%	3.02%	0.55%
L1I prefetch miss rate	17.10%	15.76%	17.64%
L1D miss rate	3.96%	3.86%	11.06%
L2 miss rate	3.22%	3.51%	4.77%
L2D miss rate	3.49%	3.79%	4.68%
L2I miss rate	0.26%	0.20%	9.78%
L3 miss rate	92.33%	93.84%	**8.42%**
L3D miss rate	93.11%	94.30%	**7.66%**
Cycles/L2 data miss	380.42	352.08	129.99
Cycles/L3 data miss	388.83	364.48	1476.13

7.3. Bottleneck Analysis

Using Jarp's methodology an analysis of "Cube3" with the three problem shapes (cube, beam, and plate) was conducted. The results of the stall analysis are shown in Table 11. For the three shapes, the majority of the stalls (around 80+ %) occur in the execution stage of the pipeline.

Table 11 Crs Stall Source

Crs	w01_d45000_dof1		w55_d55_dof1		w300_d01_dof1	
Counter Name	Count	Percent	Count	Percent	Count	Percent
BACK_END_BUBBLE_ALL	2320802441	100.00%	6696237200	100.00%	8333422043	100.00%
BE_FLUSH_BUBBLE_ALL	193117125	8.32%	354703907	5.30%	275111801	3.30%
BE_L1D_FPU_BUBBLE_ALL	49003323	2.11%	112511336	1.68%	629580214	7.55%
BE_EXE_BUBBLE_ALL	1947661879	83.92%	5954594293	88.92%	7154276379	85.85%
BE_RSE_BUBBLE_ALL	27673791	1.19%	51767374	0.77%	44776750	0.54%
BACK_END_BUBBLE_FE	99367068	4.28%	220832406	3.30%	202831524	2.43%

To further understand the cause of these stalls in the execution stage of the pipeline more statistics need to be evaluated. Looking closer at the Back-end Execution-Stage, the stalls can be broken down into more detailed statistics by using the PMU events within the *Be_exe_bubble* event shown in Table 10. The PMU can monitor what is causing the stalls within the execution stage of the pipeline; the statistics that can be collected are:

- Back-end stalls due to general register/general register or general register/load dependency (*be_exe_bubble_grall)*,
- Back-end stalls due to floating point register/floating point register or floating point register/load dependency (*be_exe_bubble_frall)*,

- Back-end stalls due to general register /general register dependency,
- Stalls due to ARCR dependency, PR dependency, Cancelled Loads, or Bank Switching (*arcr_pr_cancel_bank)*.

These statistics are shown in Table 12. The results show that depending on the shape of the problem (1x45000x1 to 55x55x1), more importantly number of nonzeros, the number of stalls due to general register to load dependency decreases and floating point to load/floating point register to floating point register increases mainly because the number of floating-point operations increases when there are more nonzeros within the matrix.

Table 12 Crs Execution Stage Stalls

Crs	w01_d45000_dof1		w55_d55_dof1		w300_d01_dof1	
BACK_END_BUBBLE_ALL	2320802441	100.00%	6696237200	100.00%	8333422043	100.00%
BE_EXE_BUBBLE_GRALL	969203079	41.76%	1635890701	24.43%	6344204662	76.13%
BE_EXE_BUBBLE_FRALL	980825829	42.26%	4311046217	64.38%	2199429333	26.39%
BE_EXE_BUBBLE_GRGR	1434	0.00%	99528	0.00%	1903	0.00%
BE_EXE_BUBBLE_ARCR_PR_CANCEL_BANK	2892140	0.12%	5779050	0.09%	4163059	0.05%

Grouping statistics to get a better understanding of the actual stall contributions from a global perspective give us the data shown in Table 13. Table 13 shows the contributions to each of the main causes of stalls; Table 14 shows a breakdown of three greatest contributions of the stalls in Table 13. The major cause of stalls is load integer and floating point dependency and/or floating point register floating point register dependency. However, Table 14 also shows that the branch stalls are due mainly to a branch misprediction bubbles (fe_bubble_bubble). Some of these stalls could be

alleviated with a bigger branch predictor such as a two-level tournament predictor.

Table 13 Crs Global Stall Counts

Stalls						
Crs	w01_d45000_dof1		w55_d55_dof1		w300_d01_dof1	
	Count	%	Count	%	Count	%
D-cache stalls	1017666288	37.046%	1749014158	23.247%	6974684612	63.882%
Branch mispredict stalls	513213458	18.683%	929897226	12.360%	736070519	6.742%
Instruction Miss Stalls	128795729	4.689%	312081523	4.148%	289025316	2.647%
RSE stalls	27673791	1.007%	51767374	0.688%	44776750	0.410%
Floating Point unit Stalls	1029290472	37.469%	4424269202	58.805%	2829911186	25.920%
GR Scoreboarding	1434	0.000%	99528	0.001%	1903	0.000%
Front-End Flushes	30372344	1.106%	56530081	0.751%	43582783	0.399%
Total	2747013516		7523659092		10918053069	

Table 14 Crs Major Stall Contributions

	Counter	w01_d45000_dof1		w55_d55_dof1		w300_d01_dof1	
Dcache	grall-grgr	969201645	95.238%	1635791173	93.526%	6344202759	90.960%
	BE_L1D_FPU_BUBBLE_L1D	48464643	4.762%	113222985	6.474%	630481853	9.040%
Branch	BE_FLUSH_BUBBLE_BRU	192861987	37.579%	354472948	38.120%	275201648	37.388%
	FE_BUBBLE_BUBBLE	270036171	52.617%	483820255	52.029%	391220995	53.150%
	FE_BUBBLE_BRANCH	50315300	9.804%	91604023	9.851%	69647876	9.462%
FP Units	BE_EXE_BUBBLE_FRALL	980825829	95.291%	4311046217	97.441%	2199429333	77.721%
	BE_L1D_FPU_BUBBLE_L1D	48464643	4.709%	113222985	2.559%	630481853	22.279%

7.4. Solution Techniques

"Cube3" has the ability to change the solver algorithms used to solve the linear equations established by the finite element problem. To understand which solver allows for the fastest solve time a comparative analysis with and without the use of preconditioning is presented in this section. Even though

the various algorithms used to solve these linear systems are beyond the scope of this work.

7.4.1. Varying Solvers

The AztecOO package in Trilinos provides various solvers as stated in Section 2.6.3. These solvers provide different iterative algorithms that allow convergence on the final solution of the linear system. Figure 29 shows the various interval data from the 55x55x1 problem size varying the different solvers. The 55x55x1 problem size is only shown because the execution phases can be seen distinctly and the changes in the solve phase can be determined. The results of using different solvers showed improvement only in the time spent in the solve phase due to faster convergence and not due to architecture performance improvements. All of the algorithms stated in section 2.6.3 all have the same characteristics in that they all use the same matrix-vector multiply; the only one that differed was the gmres solver which uses *dgemv* as a solver, which is a fortan Blas (Basic Linear Algebra Subprograms library used in many computer systems) function that performs matrix-vector multiply. The rest of the algorithms use a loop kernel within the AztecOO package of Trilinos which calculates the same matrix-vector multiply but is written in C++. Examining the performance data when using various solvers within AztecOO the only performance benefit is that some allow for faster solution convergence and hence fewer iterations are executed to converge on the final solution. The fastest performing solver is the conjugate

gradient method as shown in the lower graph of Figure 29. The slowest performing is the restarted general minimal residual (gmres), which is likely a result of the Blas libraries not being at an optimal level for the Itanium 2. The conjugate gradient method performed the best regardless of the problem shape as well as the Vbr storage format. Results are shown for the Vbr format in Appendix.

7.4.2. Vary Preconditioners

In addition to the solver algorithms, another method that allows for faster convergence on the final linear system solution is the method of matrix preconditioning. Preconditioning the matrix involves multiplying the “A” matrix by another matrix, which is called the precondition matrix, then that new matrix is sent to the solver. The various preconditioning methods that can be used by the “Cube3” workload are defined in the AztecOO solver library and also are stated in Section 2.6.3. Figure 30 is a graph that shows the performance of varying the preconditioners on a problem size of 55x55x1 using the conjugate gradient solver. Throughout the executions the k-step Jacobi allowed for the fastest convergence using only one step. As seen in Figure 30 the k-step Jacobi only spends around 1.5 billion instructions in the solve phase as compared to no preconditioning which spends around 3 billion instructions in the solve phase. The worst performance was observed using the domain decomposition preconditioner, which took the longest to complete

(almost 30 billion instructions just in the solve phase). The Vbr results concur with these results and can be found in the Appendix.

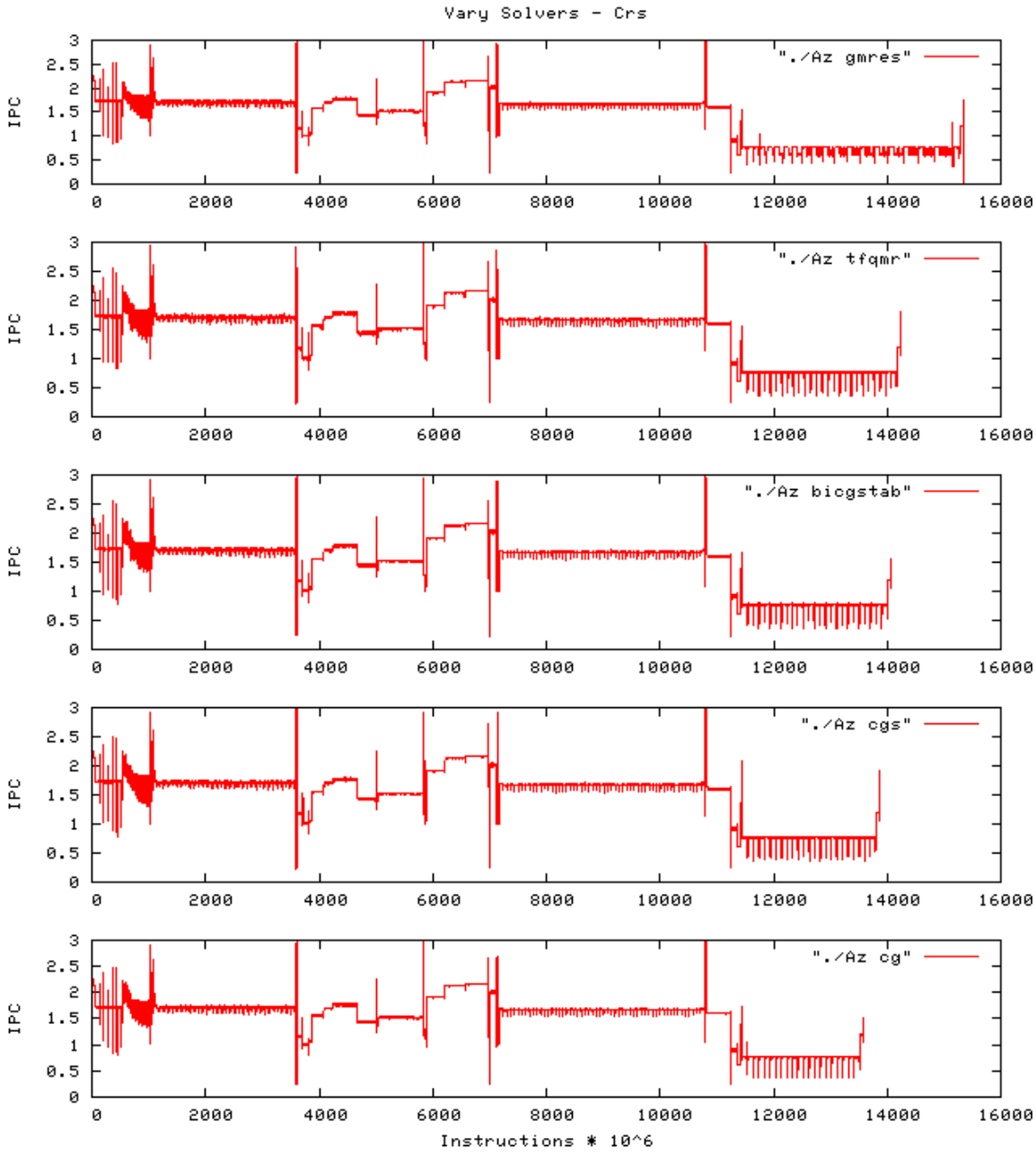


Figure 29 Crs 55x55x1 Varying Solver Methods

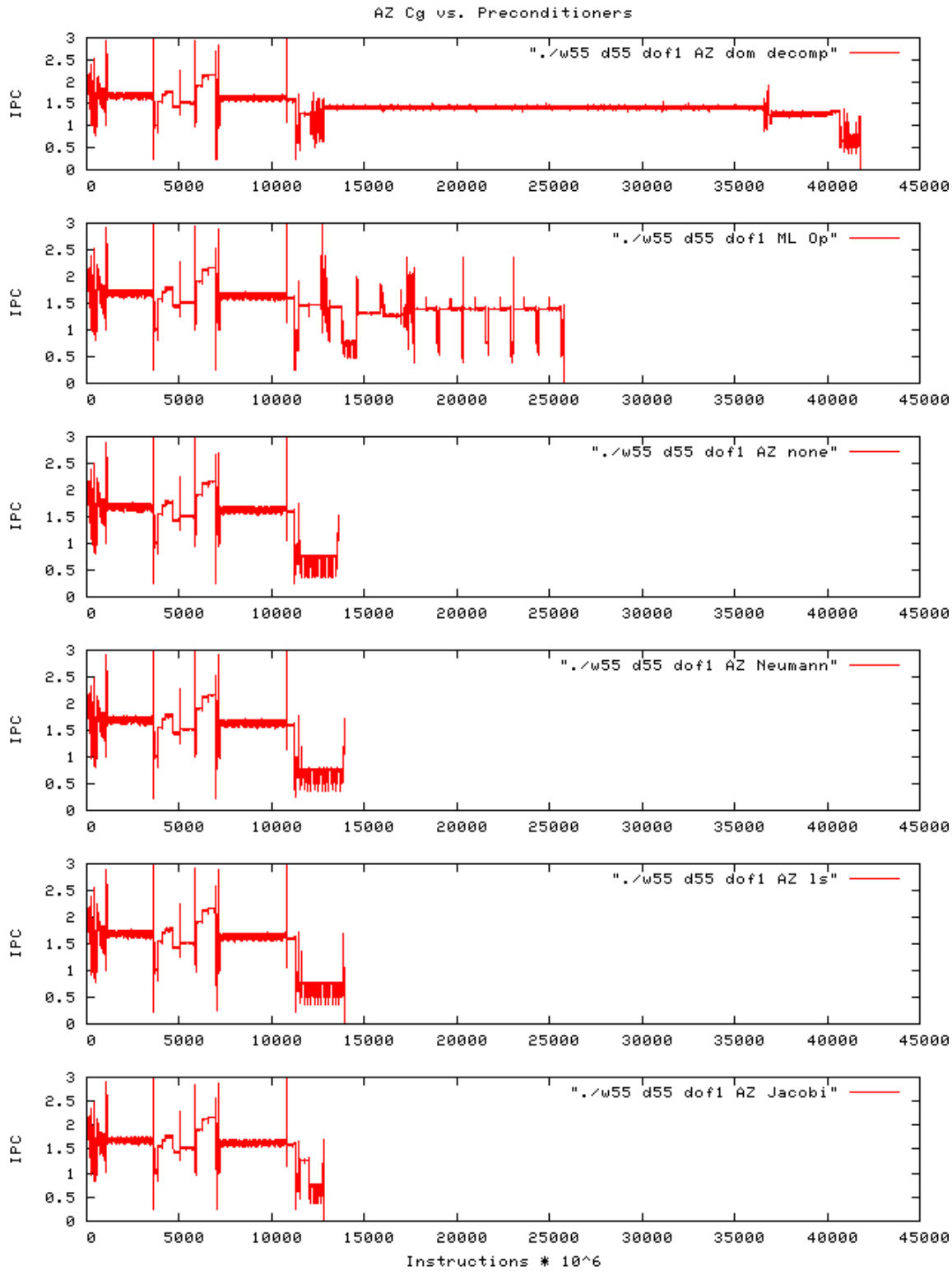


Figure 30 Crs 55x55x1 Varying Preconditioners with CG solver

7.5. Compiler Optimizations

The Itanium 2 relies mainly on the compiler to achieve performance improvement over other architectures. The performance of the application under study depends greatly on the compiler and the compiler optimizations used on the application. This section of the work studies some of the benefits of each.

7.5.1. Utilizing different Compilers

The two compilers that are available for use on the Itanium 2 at NMSU are the Gnu Gcc compiler and the Intel Icc compiler. The Gcc compiler has some built in functions that allow for performance improvement of an application. Such compiler techniques as loop unrolling and prefetching loop data help to improve the performance of an application. Loop unrolling is a concept that actually unrolls the loop iterations with the conjunction of register renaming so that the assembly code does not have any dependent instructions allowing the loops to run in parallel which improves the overall performance of this kernel. Prefetching loop data is a technique if supported by the architecture that issues prefetch instructions to fetch data used in large arrays to improve the performance of the loops. The Icc compiler does not have these capabilities available for ia64 instructions (64-bit Itanium instructions) but does have the capability to perform software pipelining that the Gcc compiler does not have. The technique of software pipelining is the method that a compiler uses by taking independent instructions from each iteration of the

original loop and creates another loop that only contains independent instructions with some setup and closing instructions to complete the same process as the original loop. Figure 31 shows the performance of the problem size of 55x55x1 when compiled with Gcc with no compiler optimizations versus the Icc compiled workload with software pipelining available as per Jarp [30]. Figure 31 shows that there is no performance gain using the Icc compiler versus the Gcc compiler. One reason may be that the Icc compiler was not installed correctly and is reverting to the Gcc libraries. Some future work in this area is needed to determine if the Icc compiler is actually working correctly.

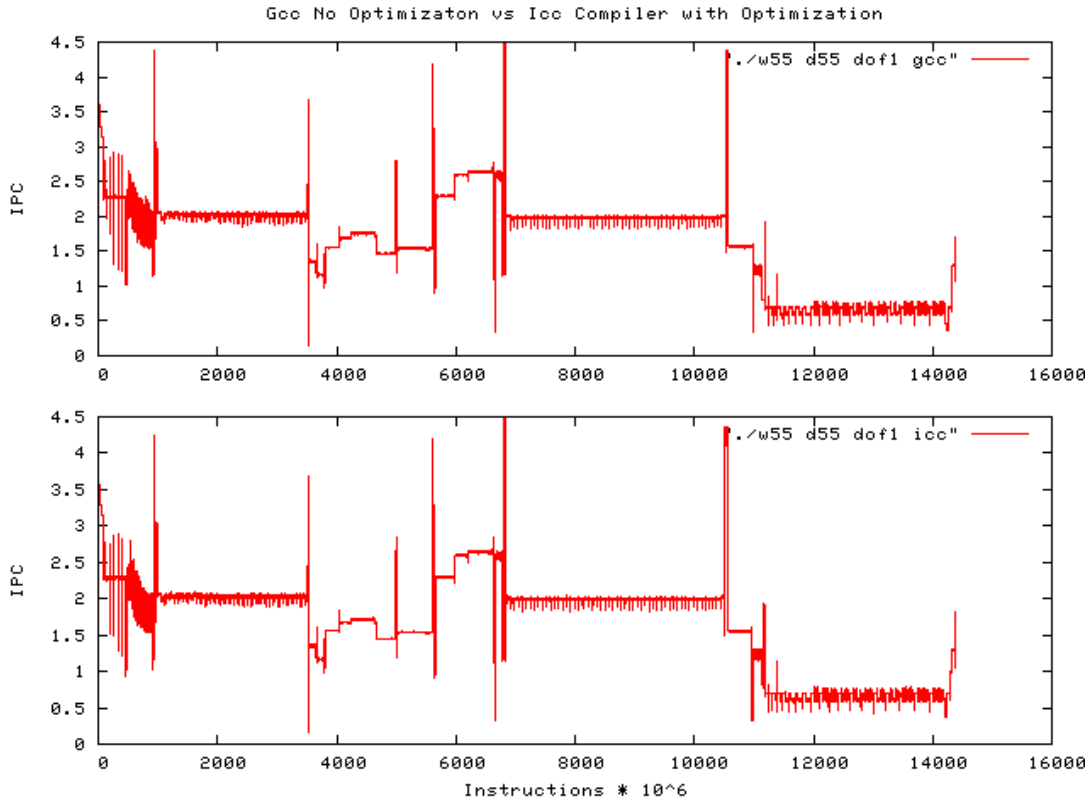


Figure 31 Crs 55x55x1 Varying Gcc & Icc Compilers

7.5.2. Loop Optimizations

The matrix vector multiplication is the primary cause of the poor performance found in the solve phase. This is due to instruction level dependency. Instruction level dependency is the term used when an subsequent instruction needs the data produced from an instruction before it. If the instruction that is needed is stalled in the pipeline because it is waiting for data from memory then the dependent instruction is stalled as well.

Some changes to the code were needed in order to see if loop unrolling was actually being conducted in the compiler to achieve maximum performance of the loop. The changes to the code were implemented in the multiply kernel of the AztecOO package (Epetra_CrsMatrix.cpp). The loop is shown in the following:

```
for(i = 0; i < NumMyRows_; i++) {
      double sum = 0.0;
      for(j = 0; j < NumEntries; j++)
        sum += RowValues[j] * xp[RowIndices[j]];

      yp[i] = sum;

    }
```

We modified the code by putting a number within the looping structure in order to recompile with loop unrolling allowing a higher level of parallelism to be extracted in the compiler. Hard coding the number of rows (NumMyRows) to an actual number instead of a variable allows the compiler to know exactly how many loops to unroll whereas before it could not perform loop unrolling because the number is not known at compile time. Once the number is hard coded and recompiled, this will show if the compiler is able to

extract some parallelism through the use of loop unrolling techniques or software pipelining. The best performance increase would be to hardcode the number of entries per row of the matrix (NumEntries) but unfortunately this is a variable and can not be hardcoded. The use of profile guided optimization can perhaps help the performance of this kernel. The problem with these optimizations is that in a real-world problem the number of loop iterations is never known at compile time and profile guided optimizations take a lot of time to implement because two compilations are needed, one to collect data and one that uses the data collected. In future work, a dynamic way of improving this loop-kernel is needed to achieve the best overall performance improvement in the “Cube3” application. The next section shows some of the compiler optimization results of this loop kernel.

7.5.2.1. Gcc vs. Gcc Optimized

The Gcc compiler allows a user to compile an application using loop unrolling and loop prefetching. The performance improvement in the solve phase is small but noticeable as shown in Figure 32 due to the loop unrolling and loop prefetching optimizations. Figure 32 shows the gmres solution interval data improvement and Figure 33 show the CG solution which only uses the matrix-vector loop kernel of AztecOO. The runtime of the non-optimized Gcc compilation (upper graph of Figure 33) using the conjugate gradient solver in CPU cycles was around 10 billion cycles and for the optimized Gcc compilation (lower graph of Figure 33) was around 8 billion

cycles. This attributes to a 20% gain of the overall execution time in the “Cube3” application shown in Figure 33.

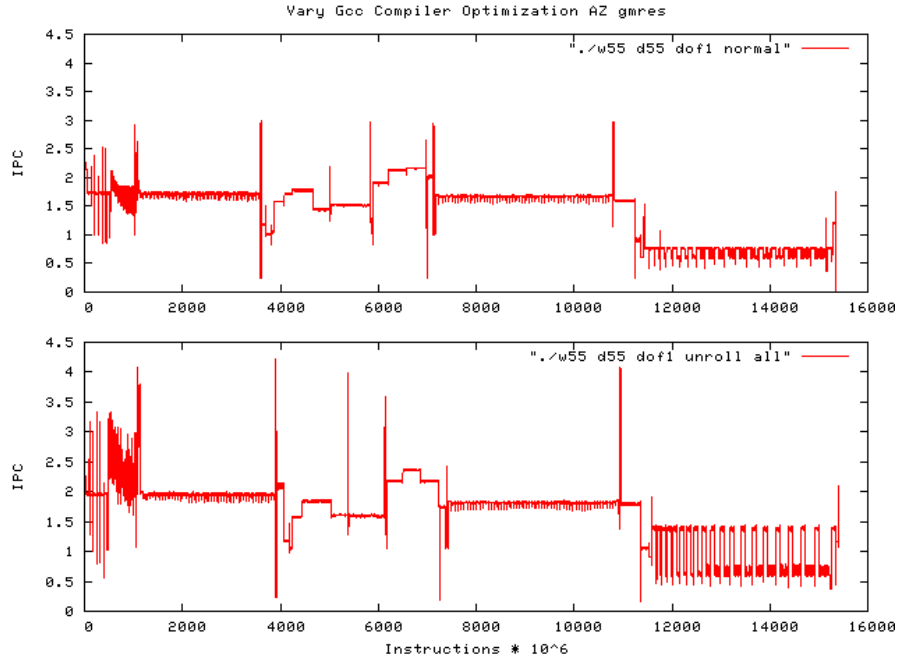


Figure 32 Gcc with Gmres Solver vs. Loop-Unrolling

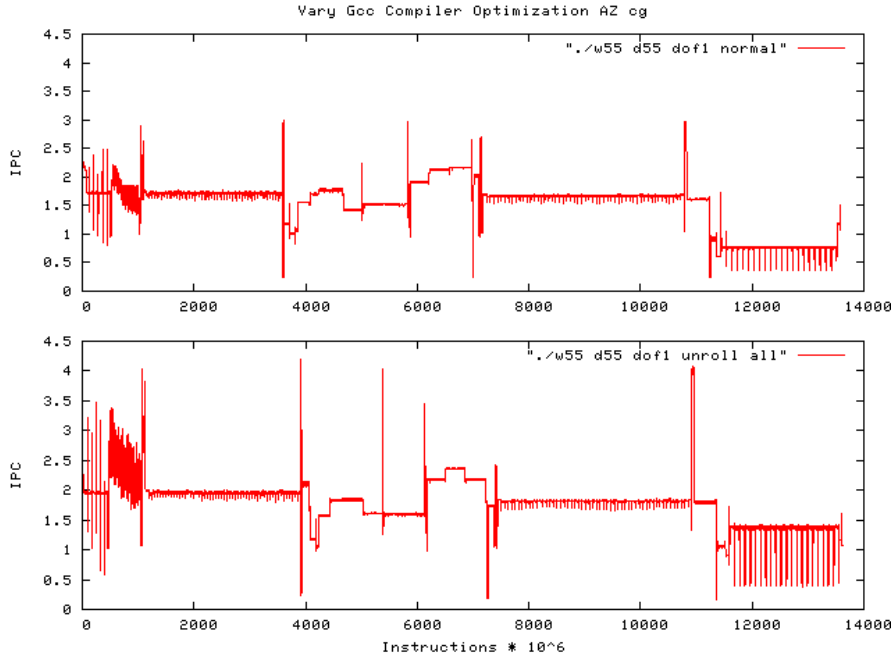


Figure 33 Gcc with CG Solver vs. Loop-Unrolling

8. CONCLUSION

The “Cube3” application proposed by Sandia National Laboratories as a representative workload of the scientific computing proved to be a difficult problem to study on the Itanium 2 microarchitecture. It was a difficult problem due to the fact that the performance varies as the shape and size changes of the problem due to the number of nonzeros within the “A” matrix and also an error in the code was discovered and hindered results of the 300x1x1 problem size. The error is currently being studied by the author of “Cube3.” When running “Cube3” on the Itanium 2 the only phase that consistently had poor performance was the solve phase. The section of the code that attributed to the poor performance of the solve phase was the matrix-vector multiply loop kernel found in the AztecOO package in Trilinos. As the problem size/number of nonzeros of the finite element problem increases the solve phase dominates the performance degradation of the application. Some studies were performed to improve the performance of this matrix-vector multiply kernel. The compiler techniques that were implemented to improve this kernel were:

- Adding *-fprefetch-loop-arrays* option in Gcc
- Adding *-funroll-loops* option in Gcc

Also, to minimize the execution time of the solve phase of the workload the use of the conjugate gradient algorithm with a *k*-step Jacobi

preconditioner proved to be the best in conjunction with the compiler options stated above.

In conclusion, once a technique has been implemented to help the matrix-vector multiply improve its performance then all of the solvers within the AztecOO package of Trilinos will improve, as well as any other solvers which have a matrix-vector multiply loop kernel similar to that of AztecOO. If a technique is found in the future to further improve this loop kernel then the finite element workload will be more dependent on the architecture under study. But at the time of this work the overall bottleneck of “Cube3” was the loop kernel and not any of the characteristics of the Itanium 2 micro-architecture.

9. FUTURE WORK

To best maximize the performance of the “Cube3” workload some additional techniques need to be implemented to maximize the performance of the matrix-vector loop kernel. The techniques need to focus on alleviating some of the data dependency within the loop so that more parallelism can be extracted allowing more instructions to be executed in parallel. This can be achieved by performing studies only on the loop-kernel, alleviating a lot of time involved with compiling and running “Cube3.” Some future research on the Icc compiler is needed to see if it is implementing software pipelining correctly, which could possibly improve this loop kernel. Also, the proposal of additional hardware/software techniques can maybe improve the performance as well.

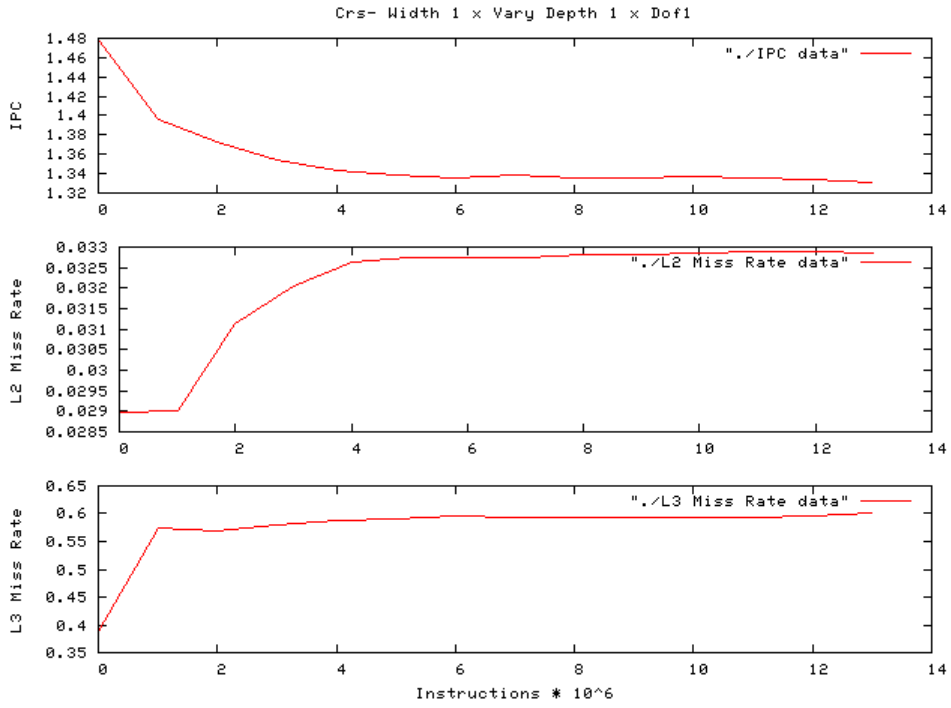
Once the performance of the matrix-vector multiply has been improved, then the architecture of the Itanium 2 needs to be re-studied to determine how “Cube3” performs because as of now the Itanium 2 architecture has not been stressed in any of the studies conducted. Also by running a workload of a dense matrix (all nonzeros) of different sizes can give more incite to the performance of the cache hierarchy and what problem size in terms of non-zeros in the matrix to choose to study the architecture to its fullest capability.

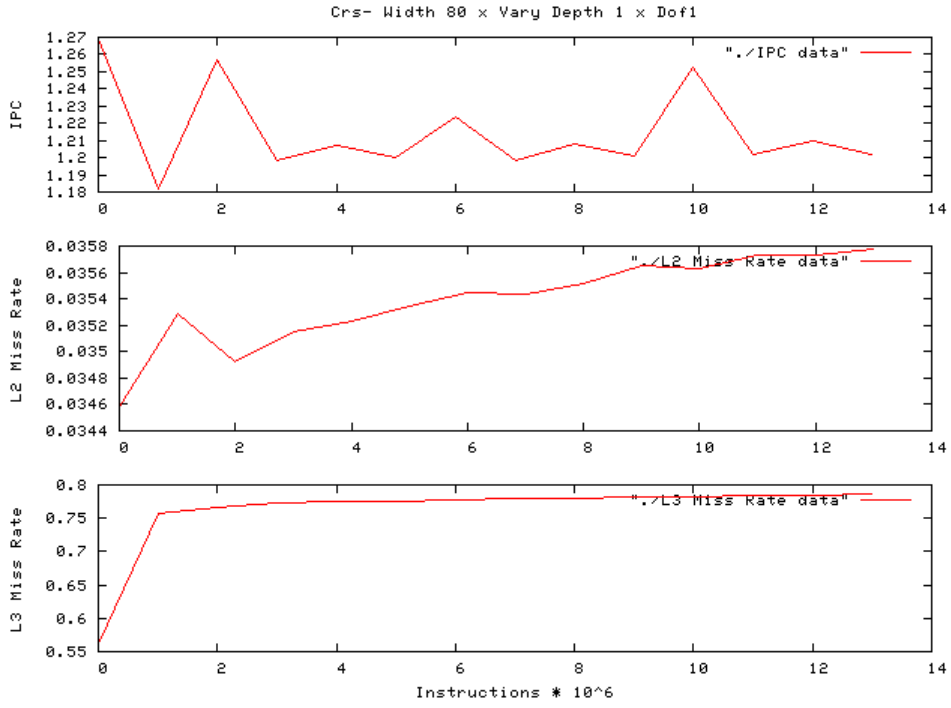
In addition to this performance analysis and future performance analysis studies, an analytic model needs to be implemented to help in the

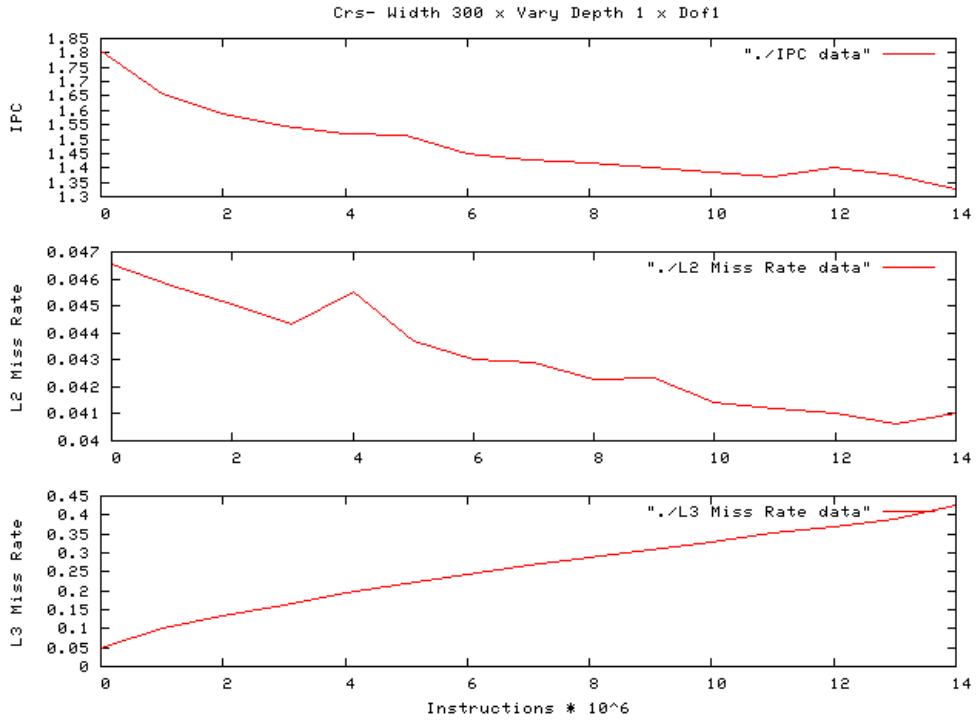
multi-processor analytic model used by Sandia National Laboratories which could have parameters based on the performance of a matrix-vector multiply and the techniques that the architecture implements to maximize the parallelism because this tends to be the most relevant performance loss as a whole.

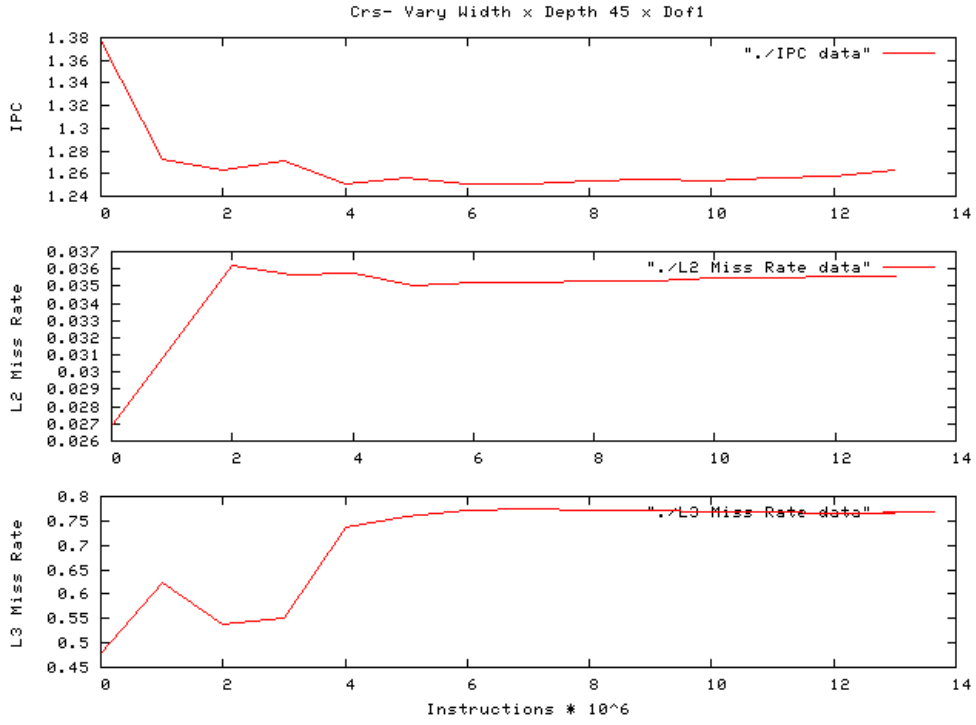
APPENDIX

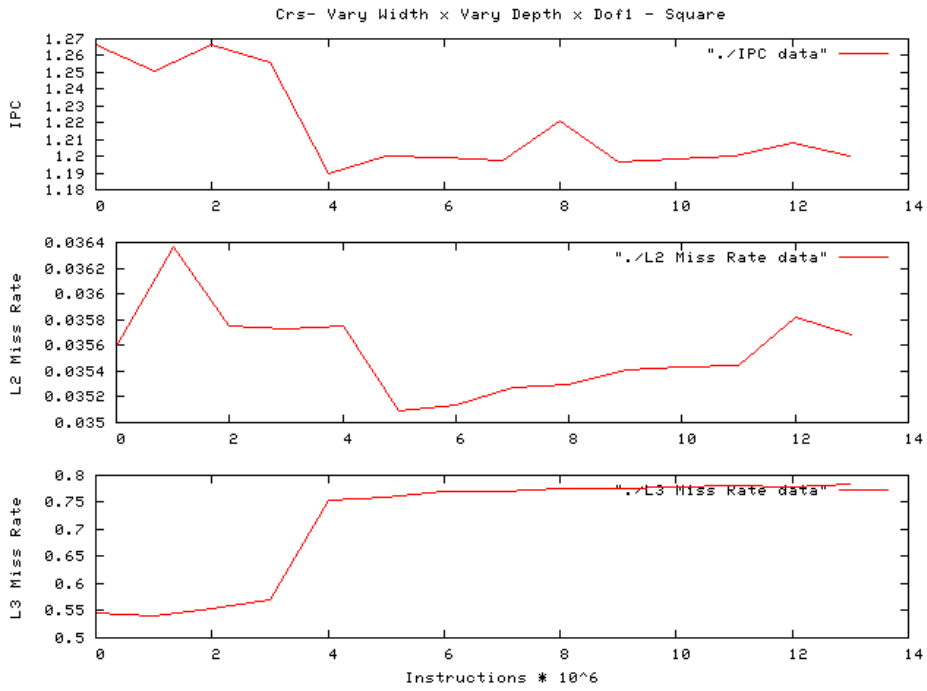
Problem Size Variation Graphs –CRS



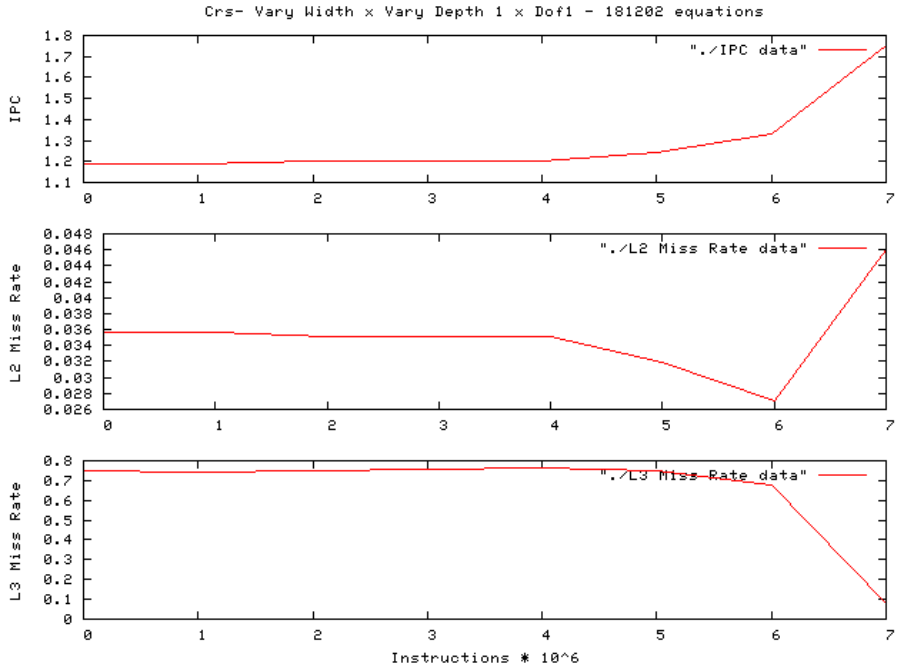


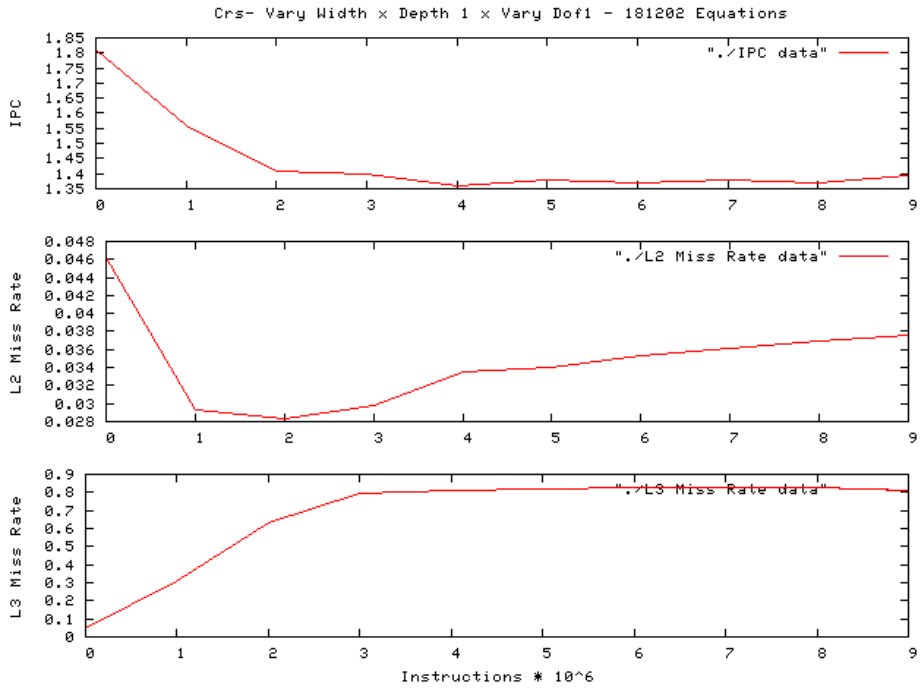

Crs- Width 300 x Vary Depth 1 x Dof1
IPC
"./IPC data"
L2 Miss Rate
"./L2 Miss Rate data"
L3 Miss Rate
"./L3 Miss Rate data"
Instructions * 10^6

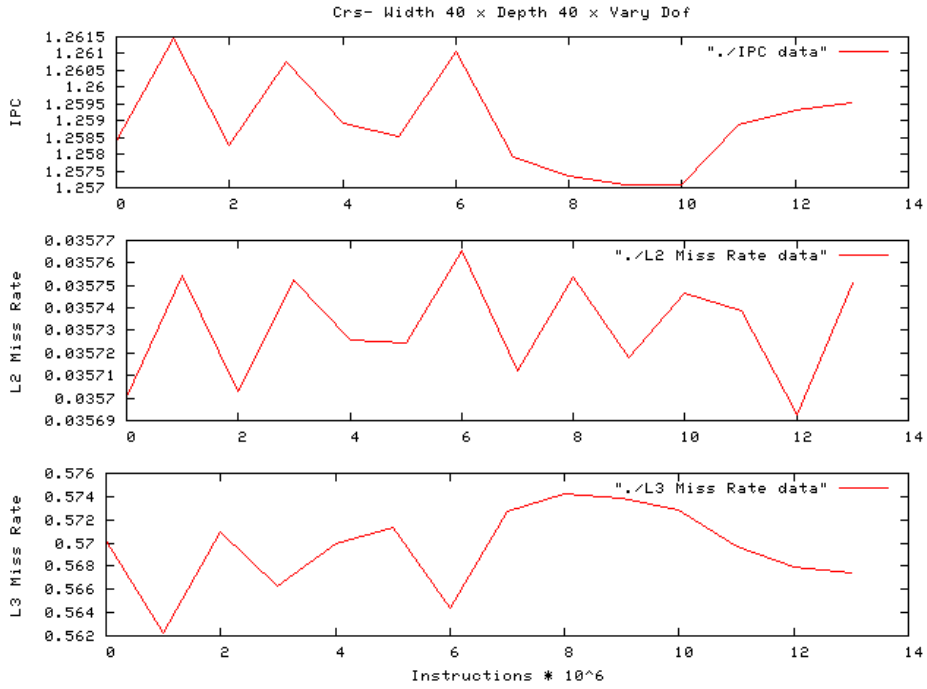

Crs- Vary Width x Depth 45 x Dof1
IPC
"./IPC data"
L2 Miss Rate
"./L2 Miss Rate data"
L3 Miss Rate
"./L3 Miss Rate data"
Instructions * 10^6

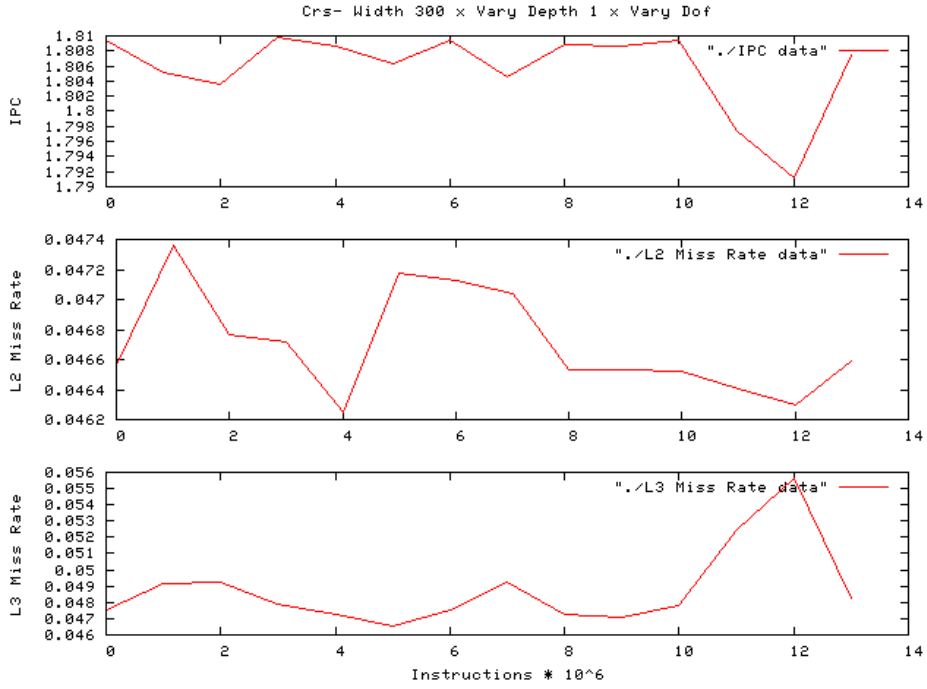

Crs- Vary Width x Vary Depth x Dof1 - Square
IPC
"./IPC data"
1.27
1.26
1.25
1.24
1.23
1.22
1.21
1.2
1.19
1.18
L2 Miss Rate
"./L2 Miss Rate data"
0.0364
0.0362
0.036
0.0358
0.0356
0.0354
0.0352
0.035
L3 Miss Rate
"./L3 Miss Rate data"
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0
2
4
6
8
10
12
14
Instructions * 10^6

Problem Shape Variation Graphs –CRS

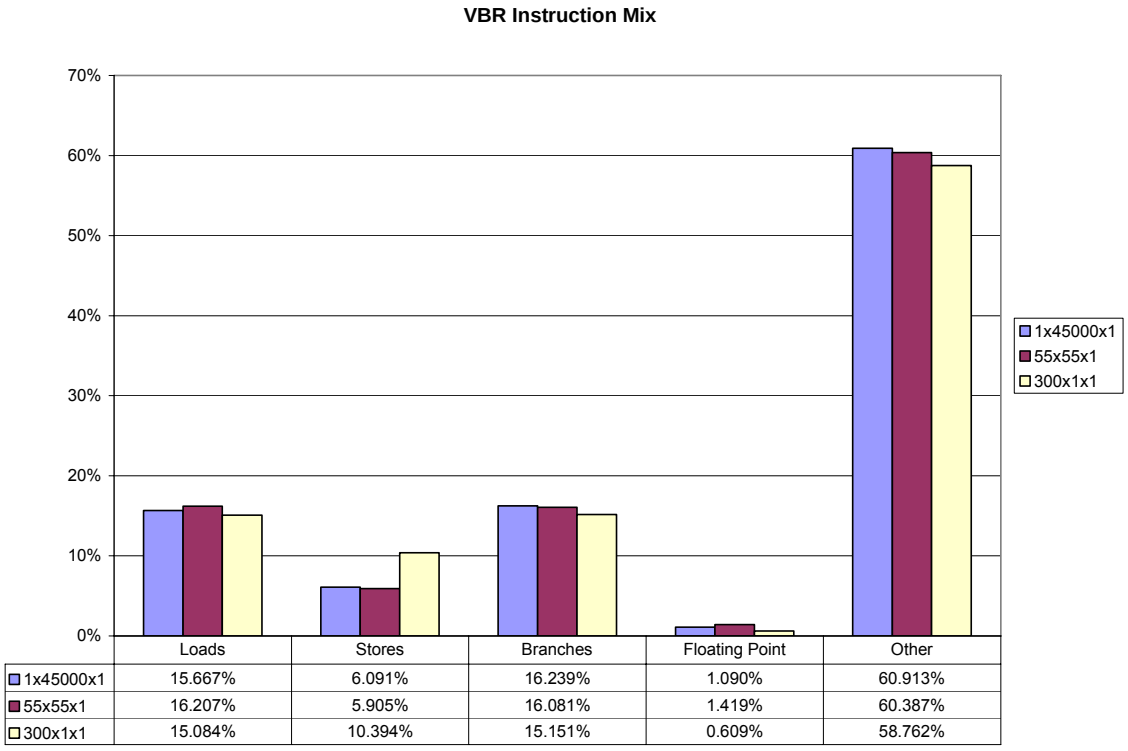





Crs- Width 40 x Depth 40 x Vary Dof
IPC
"./IPC data"
L2 Miss Rate
"./L2 Miss Rate data"
L3 Miss Rate
"./L3 Miss Rate data"
Instructions * 10^6


Crs- Width 300 x Vary Depth 1 x Vary Dof
IPC
"./IPC data"
L2 Miss Rate
"./L2 Miss Rate data"
L3 Miss Rate
"./L3 Miss Rate data"
Instructions * 10^6

VBR – Statistics/Graphs



	Loads	Stores	Branches	Floating Point	Other
1x45000x1	15.667%	6.091%	16.239%	1.090%	60.913%
55x55x1	16.207%	5.905%	16.081%	1.419%	60.387%
300x1x1	15.084%	10.394%	15.151%	0.609%	58.762%

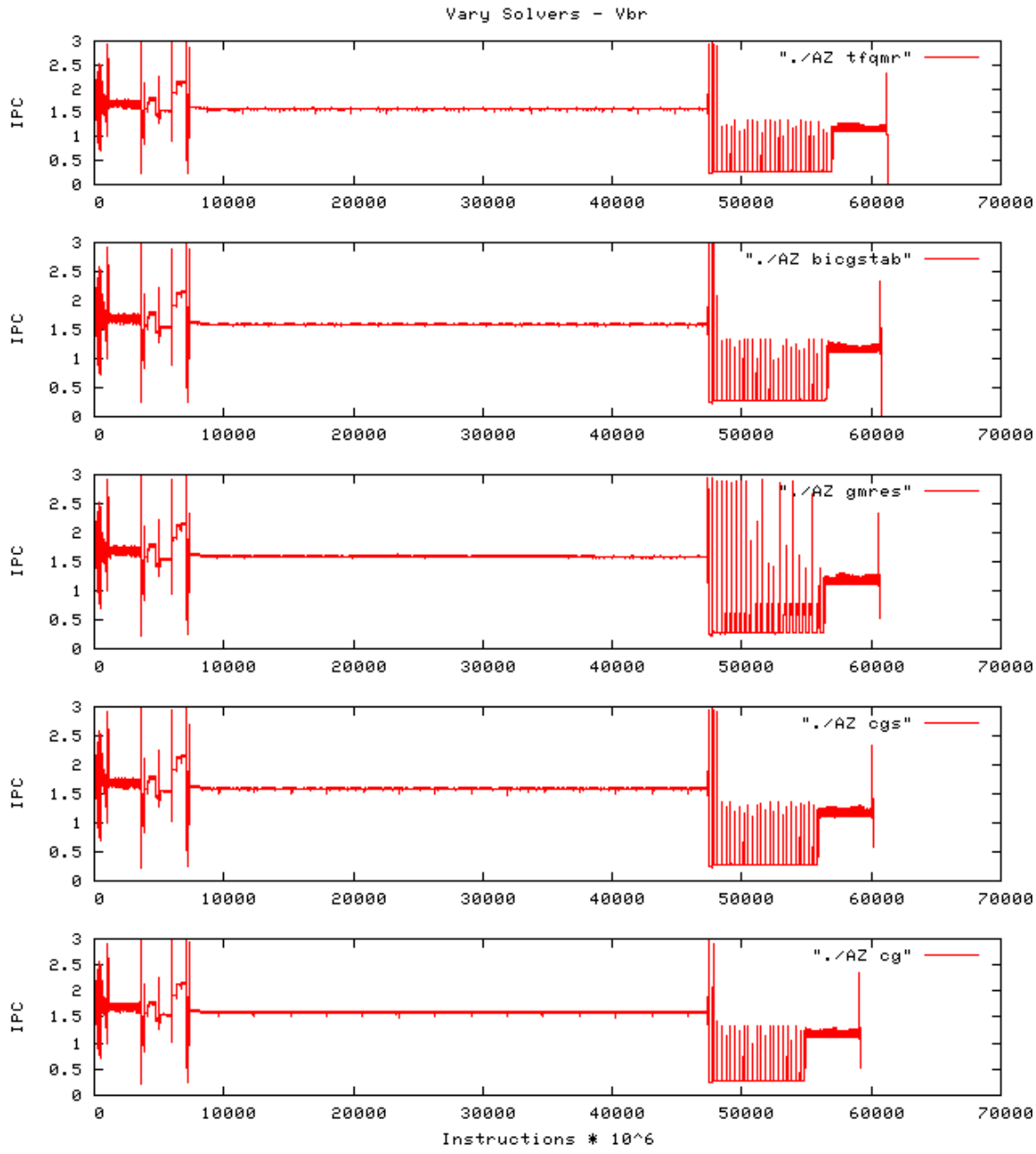
Cache Statistics			
Vbr	**w01_d45000_dof1**	**w55_d55_dof1**	**w300_d01_dof1**
L1I miss rate	4.15%	4.21%	1.89%
L1I prefetch miss rate	20.28%	21.14%	21.06%
L1D miss rate	7.60%	8.98%	10.58%
L2 miss rate	3.19%	4.81%	4.60%
L2D miss rate	3.47%	5.23%	4.63%
L2I miss rate	0.15%	0.18%	1.93%
L3 miss rate	96.79%	89.38%	**33.34%**
L3D miss rate	97.34%	89.88%	**31.59%**
Cycles/L2 data miss	395.28	303.00	183.99
Cycles/L3 data miss	398.62	330.55	543.37

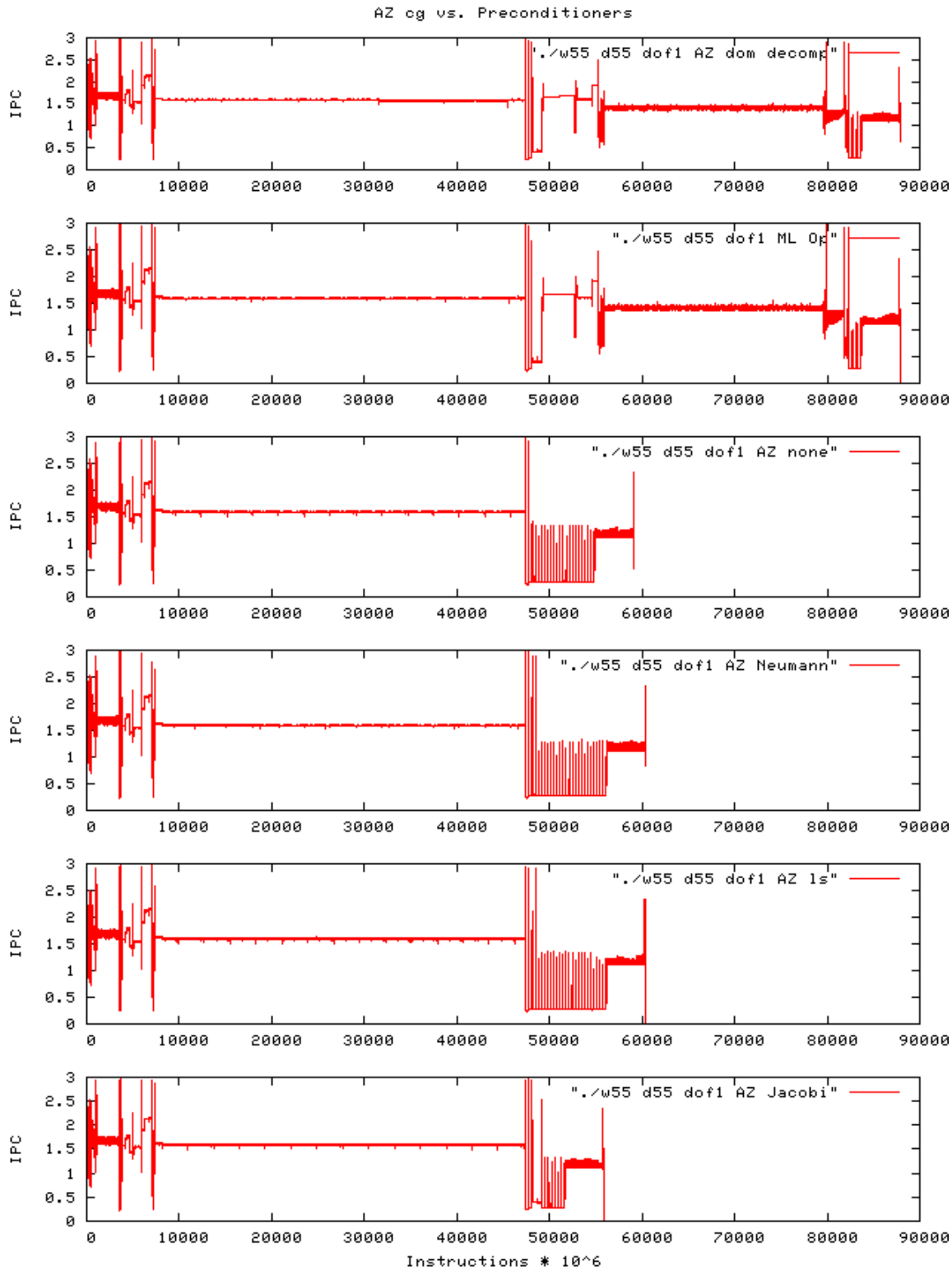
Vbr	w01_d45000_dof1		w55_d55_dof1		w300_d01_dof1	
Counter Name	Count	Percent	Count	Percent	Count	Percent
BACK_END_BUBBLE_ALL	9882892235	100.00%	39673467563	100.00%	25399976359	100.00%
BE_FLUSH_BUBBLE_ALL	1295178321	13.11%	3693153357	9.31%	2223023294	8.75%
BE_L1D_FPU_BUBBLE_ALL	136588649	1.38%	593116712	1.49%	888702138	3.50%
BE_EXE_BUBBLE_ALL	7774087202	78.66%	33295543901	83.92%	20783644887	81.83%
BE_RSE_BUBBLE_ALL	209631049	2.12%	690115968	1.74%	397920517	1.57%
BACK_END_BUBBLE_FE	451072683	4.56%	1356195974	3.42%	841363555	3.31%

BACK_END_BUBBLE_ALL	9882892235	100.00%	39673467563	100.00%	25399976359	100.00%	100.00%
BE_EXE_BUBBLE_GRALL	5563880615	56.30%	20031626715	50.49%	14404876672	56.71%	56.71%
BE_EXE_BUBBLE_FRALL	2214576925	22.41%	13251213935	33.40%	6395484604	25.18%	25.18%
BE_EXE_BUBBLE_GRGR	1414	0.00%	106328	0.00%	2062	0.00%	0.00%
BE_EXE_BUBBLE_ARCR_PR_CANCEL_BANK	2712180	0.03%	5603616	0.01%	4162439	0.02%	0.02%

Stalls						
Vbr	w01_d45000_dof1		w55_d55_dof1		w300_d01_dof1	
	Count	%	Count	%	Count	%
D-cache stalls	5700956342	41.180%	20624172455	37.496%	15291064411	45.747%
Branch mispredict stalls	4798357186	34.660%	17543891379	31.896%	8996505779	26.915%
Instruction Miss Stalls	584180606	4.220%	1719446663	3.126%	1110345988	3.322%
RSE stalls	209631049	1.514%	690115968	1.255%	397920517	1.190%
Floating Point unit Stalls	2351654066	16.987%	13843866003	25.169%	7281674405	21.785%
GR Scoreboarding	1414	0.000%	106328	0.000%	2062	0.000%
Front-End Flushes	199144808	1.438%	582475321	1.059%	348087778	1.041%
Total	13843925471		55004074117		33425600940	

	Counter	w01_d45000_dof1		w55_d55_dof1		w300_d01_dof1	
Dcache	grall-grgr	5563879201	97.596%	20031520387	97.126%	14404874610	94.205%
	BE_L1D_FPU_BUBBLE_L1D	137077141	2.404%	592652068	2.874%	886189801	5.795%
Branch	BE_FLUSH_BUBBLE_BRU	1283386628	26.746%	3700376108	21.092%	2222523236	24.704%
	FE_BUBBLE_BUBBLE	3216402404	67.031%	12947456014	73.800%	6235439269	69.310%
	FE_BUBBLE_BRANCH	298568154	6.222%	896059257	5.108%	538543274	5.986%
FP Units	BE_EXE_BUBBLE_FRALL	2214576925	94.171%	13251213935	95.719%	6395484604	87.830%
	BE_L1D_FPU_BUBBLE_L1D	137077141	5.829%	592652068	4.281%	886189801	12.170%


Vary Solvers - Vbr
"./AZ tfqmr"
"./AZ bicgstab"
"./AZ gmres"
"./AZ cgs"
"./AZ cg"
IPC
Instructions * 10^6
0
0.5
1
1.5
2
2.5
3
10000
20000
30000
40000
50000
60000
70000


AZ cg vs. Preconditioners
IPC
"./w55 d55 dof1 AZ dom decomp"
"./w55 d55 dof1 ML Op"
"./w55 d55 dof1 AZ none"
"./w55 d55 dof1 AZ Neumann"
"./w55 d55 dof1 AZ ls"
"./w55 d55 dof1 AZ Jacobi"
0
0.5
1
1.5
2
2.5
3
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
Instructions * 10^6

REFERENCES

[1] "Supercomputer Top 500," [online document], 2004 Nov. 1, [accessed 2004 Feb 26], Available HTTP: http://www.top500.org/lists/plists.php?Y=2004&M=11

[2] J. Pfeiffer, "The Memory Hierarchy," [online document], 2005 Apr. 1, [accessed 2005 May 10], Available HTTP: http://www.cs.nmsu.edu/~pfeiffer/classes/473/notes/memhierarchy.html

[3] G. Griem, et al. , "Identifying Performance Bottlenecks on Modern Microarchitectures using an Adaptable Probe", in Proceedings Parallel and Distributed Processing Symposium "04, 2004, pp. 255.

[4] Cameron McNairy and Don Soltis, "Itanium 2 Processor Microarchitecture," IEEE Micro, vol. 23, issue 2, pp 44-55, 2003

[5] "Itanium 2 Reference Manual for Software Development and Optimization," [online document], 2004 May 1, [accessed 2004 June 10], Available HTTP: http://www.intel.com/design/itanium2/manuals/251110.htm

[6] "HP PERFMON," [online document], 2004 Jan. 1, [accessed 2004 Feb 10], Available HTTP: http://www.hpl.hp.com/research/linux/perfmon/

[7] "Intel Vtune," [online document], 2005 Jan. 1, [accessed 2005 Mar 1], Available HTTP: http://www.intel.com/software/products/vtune/

[8] M. A. Heroux, et al., "An overview of Trilinos," Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[9] M. A. Heroux, M. Sala, and D. Day, "Trilinos 4.0 tutorial," Technical Report SAND2005-6331, Sandia National Laboratories, 2004.

[10] M. A. Heroux, "Trilinos Overview," Proceedings of the Eighth Copper Mountain Conference on Iterative Methods, Copper Mountain, CO., March 28 – April 2, 2004

[11] M. A. Heroux and J. M. Willenbring, "Trilinos Users Guide," Technical Report SAND2003-2952, Sandia National Laboratories, 2003.

[12] G. P. Nikishkov, "Introduction to the Finite Element Method," [online document], 2004 Jan. 1, [accessed 2005 Mar 15], Available HTTP: http://www.u-aizu.ac.jp/~niki/feminstr/introfem/introfem.html

[13] Y. Saad, Iterative Methods for Sparse Linear Systems, [online document], 1st edition, 1996, [accessed 2005 Feb 10], Available HTTP: http://www-users.cs.umn.edu/~saad/books.html

[14] O. Ural, Finite Element Method: Basic Concepts and Applications, New York, Intext Educational Publishers, 1973

[15] D. J. Dawe, Matrix and Finite Element Displacement Analysis of Structures, New York, Oxford University Press, 1984

[16] N. M. Baran, Finite Element Analysis on Microcomputers, New York, McGraw-Hill Book Company, 1988

[17] M. A. Heroux "AztecOO users guide," [online document], Technical Report SAND2004-3796, 2004 July 1, [accessed 2005 Mar 30], Available HTTP: http://software.sandia.gov/Trilinos/packages/aztecoo/AztecOOUserGuide.pdf

[18] "Sparse Matrix Storage schemes," [online document], 2003 Jan. 1, [accessed 2005 March 17], Available HTTP: http://www.sun.com/products-n-solutions/hardware/docs/html/817-0086-10/prog-sparse-support.html

[19] V. E. Taylor, "Sparse Matrix Computations: Implications for Cache Designs," in *Proceedings of the 1992 ACM/IEEE conference on Supercomputing,* December 1992.

[20] O. Temam, W. Jalby, "Characterizing the Behavior of Sparse Algorithms on Caches," *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, December 1992.

[21] L.M. Napolitano Jr., "A Computer Architecture for Dynamic Finite Element Analysis," in *Proceedings of the 13th annual international symposium on Computer Architecture,* 1986.

[22] M. W. Berry, "Scientific Workload Characterization by Loop-Based Analysis," ACM SIGMETRICS Performance Evaluation Review, vol. 19, issue 3, 1992, pp. 17-29

[23] M. Christon, "A Vectorized 3-D Finite Element Model for Transient Simulation of Two-Phase Heat Transport with Phase Transformation and a Moving Interface," in Proceedings Supercomputing '90, 1990, pp. 436-445.

[24] V. E. Taylor and A. Ranade, “Three-Dimensional Finite-Element Analyses: Implications for Computer Architectures,” in Proceedings Supercomputer ‘91, 1991, pp. 786-795.

[25] R. Vuduc, et al. , “Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply,” in Proceedings Supercomputing ‘02, 2002, pp. 26

[26] A. Purkayastha, et al. , “Performance Characteristics of Dual-Processor HPC Cluster Nodes Based on 64-bit Commodity Processors,” [online document], 2004 Jan. 1, [accessed 2005 May 10], Available HTTP: http://www.tacc.utexas.edu/publications/performancehpcclusternodes.pdf

[27] D. Bradley, et al., “Supercomputer Workload Decomposition and Analysis,” In Proceedings Supercomputing ‘91, 1991, pp. 458-467.

[28] Alan Williams, “Cube3 Description,” unpublished article, 2004

[29] Severe Jarp, “A Methodology for using Itanium 2 Performance Counters for Bottleneck Analysis,” [online document], 2002 Jan. 1, [accessed 2004 Mar 20], Available HTTP: http://www.gelato.org/pdf/Performance_counters_final.pdf

[30] Severe Jarp, “Searching for optimal performance on PF/Linux,” [online document], 2002 Jan. 1, [accessed 2005 May 20], Available HTTP: http://www.gelato.org/pdf/Compiler_options_final.pdf