

SANDIA REPORT

SAND2005-6917

Unlimited Release

Printed November, 2005

Understanding the Effects of Microarchitectural parameters on the Uniprocessor Performance of Sandia Scientific Applications

DANA HARDIN

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>



UNDERSTANDING THE EFFECTS OF MICROARCHITECTURAL PARAMETERS
ON THE UNIPROCESSOR PERFORMANCE OF
SANDIA SCIENTIFIC APPLICATIONS

BY

DANA HARDIN, 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

“Understanding the Effects of Microarchitectural Parameters of the Uniprocessor Performance of Sandia Scientific Applications,” a thesis prepared by Dana Janae Hardin 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. Steve Stochaj

Dr. Erik DeBenedictis

VITA

March 27, 1980	Born at Hobbs, New Mexico
1998	Graduated from Lovington High School, Lovington, New Mexico
1998-2000	Associate of Science, New Mexico Junior College Hobbs, New Mexico
2000-2003	Bachelor of Science, Electrical Engineering New Mexico State University Las Cruces, New Mexico
2003-2005	Graduate Assistant College of Engineering New Mexico State University

Field of Study

Major Field: Electrical Engineering (Computer Engineering)

ABSTRACT

UNDERSTANDING THE EFFECTS OF MICROARCHITECTURAL PARAMETERS ON THE UNIPROCESSOR PERFORMANCE OF SANDIA SCIENTIFIC APPLICATIONS

Master of Science in Electrical Engineering

New Mexico State University

Las Cruces, New Mexico, 2005

Dr. Jeanine Cook, Chair

Designing the best performing microprocessor for a class of applications involves researching the impact of each major design decision and exploring innovative methods to best solve the application specific challenges. The class of large scientific applications executed at Sandia National Laboratories present unique characteristics and challenges for microprocessor performance optimization. This thesis investigates bottlenecks limiting the performance of SNL's large scientific applications and proposes configurations and techniques to improve performance. Initially, the cache hierarchy was proposed to a major bottleneck to the overall performance of these applications, however we present evidence, through simulation, that even perfect cache behavior (no cache stalls) does not greatly improve performance. Moreover, simulations with a "super" microarchitecture, configured with nearly infinite resources, show only modest

performance gains. Since other types of benchmarks achieve IPC rates of hundreds, even tens of thousands of instructions per cycle with this “super” configuration, the performance reduction from instruction-level dependency stalling was the next potential bottleneck explored. Our simulation of the instruction-level dependency stalls in the SNL benchmark for the default Alpha configuration reveals that on average each instruction incurs more than 5.5 stall cycles waiting for the resolution of instruction-level dependencies. Classifying each stall by instruction type indicates that most stalling occurs from floating-point and load instructions. Possible techniques to reduce the stalling caused by these instructions are discussed. Finally, we present the best performing finite cache configurations for the SNL benchmark and performance results from a bypass caching technique designed to exclude large matrices and vectors from the cache hierarchy in order to prevent cache pollution.

Table of Contents

List of Tables.....	viii
List of Figures.....	ix
1 INTRODUCTION.....	1
1.1 The Cube Benchmark.....	1
1.2 Performance Evaluation.....	5
1.2.1 Analytical Modeling.....	6
1.2.2 Direct Measurement.....	6
1.2.3 Simulation.....	7
1.3 SimpleScalar Simulators.....	8
2 RELATED WORK.....	13
2.1 Performance Analysis.....	13
2.2 Performance Improvement Techniques.....	18
2.2.1 Itanium2 Processor.....	19
2.2.2 Selective Fill Data Cache.....	22
2.2.3 Load Redundancy.....	26
2.2.4 Split and Victim Caches.....	28
3 METHODOLOGY.....	32
3.1 Performance Analysis Tools and Metrics.....	32
3.2 Problem Size.....	35
3.2.1 Problem Size Results for the Alpha Configuration.....	37
3.2.2 Problem Size Results for the Itanium2 Processor.....	42
3.3 Performance and Bottleneck Analysis.....	45

3.3.1	Investigating the Cache Bottleneck.....	47
3.3.2	The Integer Issue Queue and Other Architectural Bottlenecks.....	51
3.3.3	Instruction-level Dependency Bottleneck.....	54
3.3.4	Reducing Instruction-level Dependency.....	61
3.4	Performance Results and Conclusions.....	65
4	IMPROVING CACHE PERFORMANCE.....	66
4.1	Cache Configuration Effects of Performance.....	69
4.2	Cache Bypassing.....	73
5	CONCLUSION.....	77
	REFERENCES.....	80
	APPENDIX A.....	82

List of Tables

Table 1.1: ALPHA 21264 configuration for sim-alpha.....	10
Table 3.1: Itanium2 configuration for sim-alpha.....	34
Table 3.2: Average performance statistics for sim-alpha.....	46
Table 3.3: “Super” configuration.....	52
Table 3.4: IPC achieved by sim-alpha and sim-outorder with “super” configuration.....	53
Table 4.1: Recommended cache configuration for sparse matrix multiplication.....	67
Table 4.2: IPC and cache miss rates for various cache configurations.....	72

List of Figures

Figure 1.1: CRS format example.....	3
Figure 1.2: VBR format example.....	4
Figure 1.3: ALPHA 21264 block diagram.....	10
Figure 1.4: Explanation of different cache configurations.....	11
Figure 2.1: Itanium2 block diagram.....	21
Figure 2.2: VPR benchmark results of Selective Fill Data Cache method.....	24
Figure 2.3: MCF benchmark results of SFDC method.....	24
Figure 2.4: Parser benchmark results of SFDC method.....	25
Figure 2.5: GZIP benchmark results of SFDC method.....	25
Figure 3.1: Visualization of 3D “cube” benchmark mesh.....	36
Figure 3.2: Sim-alpha performance (IPC) for varying width and depth.....	38
Figure 3.3: Sim-alpha performance (cache miss rates) varying width and depth.....	38
Figure 3.4: IPC and number of instructions for equal equations.....	39
Figure 3.5: Number of equations versus problem size.....	40
Figure 3.6: Performance of Itanium2-type configuration on sim-alpha.....	42
Figure 3.7: Performance of Itanium2 processor.....	43
Figure 3.8: Itanium2 performance for varying degrees of freedom (40x40).....	44
Figure 3.9: Itanium2 performance for varying degrees of freedom (300x1).....	45
Figure 3.10: Sim-alpha IPC for problem size 55x55x1 (Alpha configuration).....	48
Figure 3.11: Sim-alpha IPC for problem size 55x55x1 (infinite cache).....	48
Figure 3.12: Sim-alpha IPC for fma3d (Alpha configuration).....	50
Figure 3.13: Sim-alpha IPC for fma3d (infinite cache).....	50

Figure 3.14: Average number of total stall cycles for sim-alpha (“cube” 55x55x1).....	55
Figure 3.15: Instruction Mix for sim-alpha (“cube” 55x55x1).....	56
Figure 3.16: Reorder buffer stalls by instruction for sim-alpha (“cube” 55x55x1).....	57
Figure 3.17: Instruction Mix for sim-alpha (fma3d).....	58
Figure 3.18: Fma3d reorder buffer stalls by instruction.....	58
Figure 3.19: Instruction Mix for Itanium2 processor (“cube” 55x55x1).....	60
Figure 3.20: Total stalls collect from Itanium2 processor (“cube” 55x55x1).....	60
Figure 3.21: Loop unrolling technique applied to “cube” inner loop.....	62
Figure 3.22: Software pipelining technique applied to the “cube” inner loop.....	64
Figure 4.1: Bandwidth of sparse matrix M.....	68
Figure 4.2: Cache hit ratio for cache configurations varying block size from [27].....	69
Figure 4.3: Cache performance for caches with varying block sizes.....	70
Figure 4.4: Cache performance for caches with varying associativity.....	71
Figure 4.5: Cache performance for caches with varying sizes.....	71
Figure 4.6: Cache bypassing performance results.....	74

1 INTRODUCTION

Ideal computers would be designed to perform well on each and every type of program or application to be executed. However, the truth in life and computer architecture is that everything is a trade-off. Computers optimized for specific tasks may exhibit degraded performance for other workloads. In some arenas, such as personal computing, performance needs to be good for various types of applications. On the other hand, most scientific applications tend to execute mostly tight-looped, repetitive computations. Optimizing computers for these tight-looped computations would improve performance for those executing several of these applications daily.

To efficiently solve large scientific problems, a group of microprocessors each optimized for scientific computing is needed. The intent of this research is to understand the major performance constraints of scientific computing and to determine how to optimize the performance of these workloads at the uniprocessor level. Understanding and improving uniprocessor performance is an integral step in developing a prescription for better overall performance of scientific applications executed on supercomputers.

1.1 The Cube Benchmark

Benchmarks are programs specifically designed or chosen to measure and compare performance on different computers. Uni-processor benchmarks are divided into two classes (integer, floating-point). Integer benchmarks perform the majority of operations on 32-bit words, and similarly floating-point benchmarks concentrate on single-precision floating point decimal operations (also 32-bits). The non-profit

corporation SPEC, Standard Performance Evaluation, was established to approve and maintain a standardized set of relevant benchmarks that can be applied to the newest generation of high-performance computers [1]. SPEC has created a suite of floating point benchmarks that serve as a standard representative of the key characteristics of scientific workloads. Likewise, Sandia National Laboratories (SNL) also has developed a scientific benchmark, initially created to verify linear solver libraries and the Finite Element Interface (FEI). FEI is a platform allowing applications to interface with multiple solvers needed in different types of scientific computations.

According to SPEC standards, a potential benchmark must meet the following specified criteria to be included in their suite of benchmarks. First, the benchmarks must be commonly used and utilize a significant portion of the hardware resources. They must solve important and relevant technical problems and produce valid results to be published in a respectable publication. Finally, they require benchmarks to be maintainable and pertinent to computer designers and vendors [1].

The cube test benchmark developed by SNL employs the Trilinos library of solvers to perform finite element analysis (FEA). FEA is a common technique for modeling complex structures and calculating the response of the model/structure to different conditions by solving a set of simultaneous equations. The cube test method starts with the creation of a cube model (8-node hexahedral) with the user specified dimensions and degrees of freedom. The number of degrees of freedom represents the number of statistics (i.e. physical properties like temperature, distance, etc) collected at each node of the cube mesh and is responsible for the density of the mesh.

In the next step, the cube is divided into smaller elements connected at specified node points and is arranged into either a Compressed Row Storage (CRS) or Variable

Block Row (VBR) format depending on user preference. The CRS format arranges the nonzero elements of a matrix in an array of values that is contiguous in memory. The array is paired with two descriptive vectors; one that provides the column number of each nonzero element and another vector that stores the locations in value array that represent the first in each row [2]. An example of a matrix arranged in CRS format is shown in Figure 1.1 below:

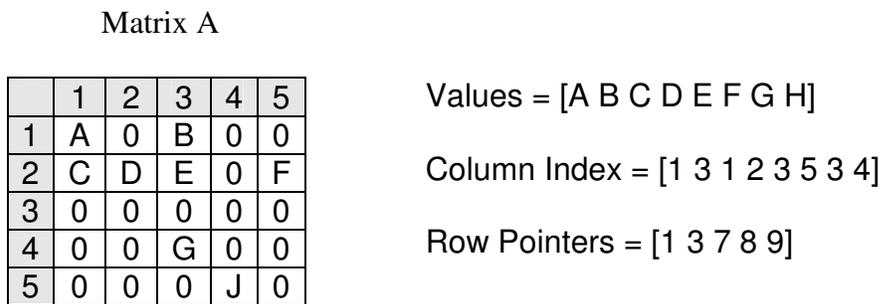


Figure 1.1: Example matrix A displayed in CRS format

CRS reduces the necessary memory storage locations from one location for each element of the cube (width * height * depth) to 2 times the number of nonzeros elements added to the width plus 1 ($2 * nnz + W + 1$). In the example above, the memory locations would be reduced from 25 to 22 with the CRS formatting. The benefits of CRS formatting are contiguous nonzero elements and less memory space, and the only drawback is the introduction of indirect memory accessing to the array.

VBR provides an efficient way to arrange sparse matrices according to clusters or blocks of nonzero data. The VBR format organizes matrices or cube meshes into six arrays including:

Row pointer array (rptr) - pointing to the first row number of each block row

Column pointer array (cptr) - pointing to the first column number of each block column

Value array (val) - containing the entries of the matrix

Index array (indx) - pointing to the beginning of each block entry stored in val

Block index array (bindx) – pointing to the block column indices of the nonzero blocks

Block pointer array (bptr) – pointing to the beginning of each block row in bindx and val

Matrix B

	1	2	3	4	5	6	7	8
1	A	C	E			I		
2	B	D	F			J		
3				G	H	K		
4						L	N	P
5						M	O	Q

rptr = [1 3 4 6]

cptr = [1 4 6 7 9]

val = [a b c d e f i j g h k l m n o p q]

indx = [1 7 9 11 12 14 18]

bindx = [1 3 2 3 3 4]

bptr = [1 3 5 7]

VBR Matrix B

	1	2	3	4
1	B1		B2	
2		B3	B4	
3			B5	B6

Figure 1.2: Example Matrix B displayed in VBR format

The VBR format, although used less frequently than CRS, helps the sparse matrix solvers to perform the kernel matrix operations more efficiently on the block entries [3].

Preconditioning is a technique by which the cube (system of equations) is transformed into an equivalent representation that converges more rapidly than the

original mesh. After preconditioning the “cube” with SNL’s main multigrid (ML) preconditioning package, the large system of linear equations is solved [4]. Sandia’s Aztec00 solver library provides several Krylov iterative solvers. Krylov iterative solving techniques work to minimize the residual (or error) with each iterative approximation to rapidly converge on a solution.

The “cube” test problem has been evaluated and meets the specified requirements of a benchmark according to SPEC standards. FEA is an important technique in modeling and solving large-scale scientific problems of large dimensions and degrees of freedom for Sandia National Labs and other scientific organizations. A benefit of the “cube” test problem is that it can be sized according to the amount of resources available, allowing the user to determine how rigorously it taxes the architecture. FEA is a commonly employed technique for solving and modeling various types of structures and environments such as thermal analysis, heat transfer, frequency analysis, fluid flow, motion simulation, and electromagnetic interactions. Because of the frequent use and the importance of FEA, the results of the “cube” test program as well as the functionality of all the preconditioners and solvers have been carefully validated and documented. Finally, a technique to optimize the performance of the “cube” test problem and the scientific workloads that it represents will be useful to organizations like Sandia that run large numbers of these applications on a daily basis.

1.2 Performance Evaluation Strategies

The performance of particular benchmarks on microprocessors can be evaluated through one or a combination of three primary methods – analytical modeling,

measurement, and simulation [5]. It is important to realize that each performance analysis method possesses its own strengths and shortcomings. Therefore, there is no one preferred evaluating technique; the choice is made based on the resources and time available and the accuracy needed.

1.2.1 Analytical Modeling

Analytical models replicate microprocessor and cache architectures in order to predict performance. These models accomplish this modeling through mathematical equations that describe hardware behavior. Analytical models aim to describe the hardware behavior using a mathematical equation. Describing such a complex system mathematically is not an easy task. While analytical modeling surpasses the other methods in simplicity of implementation and time of execution, these models have a hard time matching the accuracy of simulators or real measurements. Analytical models are fast and useful for reducing the design space to the best configuration of major structures within a processor, but are typically not very accurate at predicting overall performance.

1.2.2 Direct Measurement

Measurement methods are most commonly used to evaluate and compare the performance of different architectures and systems. Since the performance measurements are obtained from real, already manufactured machines, this technique is used to compare different architectures against each other. Many would argue that the results from measurement techniques are more accurate than other approximating techniques (simulation and modeling). However, variations in runtime factors such as

scheduling and workload requirements create difficulty in recreating the same experimental environment on different systems at different times in order to achieve an accurate comparison [6].

Direct measurement methods are useful for determining how architectures and caches actually perform with real workloads, but are limited to working systems and the performance monitoring capabilities of those systems. Direct measurement results are also limited by the number of performance counters available for a particular architecture. The performance counters on modern processors are able to track performance metrics program execution. Most modern architectures provide between 2 and 8 performance counters while some, like the Power4 architecture, provide up to 18 counters. The complexity of modern computer architectures forces designers to move beyond measurement to predict and test performance before the designs are fabricated.

1.2.3 Simulation

Simulators are software models of computer systems that provide a reconfigurable architecture and are used to predict the performance of microprocessors. Because simulators are only models that attempt to replicate the behavior of real hardware, the accuracy of simulation analysis depends on the ability of the model to mimic the functioning of the desired configuration. Simulation results must always be considered a non-perfect representation of the true performance. Nonetheless, modern simulators are meticulously validated against real hardware and remain an essential part of the process to understand and improve the performance of microprocessors.

The two main types of architectural simulators are trace-driven and execution-driven simulators. Trace-driven simulators execute a list of events or instructions that are

either collected from the execution of a workload on a native system or are a generated synthetic list called traces. Trace-driven simulators gain some efficiency from executing a list of instructions already decoded and ordered, but the simulators still take substantial time when executing lengthy traces generated by real applications. Research is being done on the reduction and sampling of traces to improve the effectiveness of trace-driven simulations, but the greatest drawback to trace-driven simulation remains the inability of a trace to capture any speculatively executed instructions [7, 8].

Execution-driven simulators model actual processor functionality, and are therefore the most accurate of microprocessor simulators. However, because modern processors are superscalar, speculative, and execute out-of-order, these simulators are increasingly complex and take an excessively large amount of execution time. Cycle-accurate (or timing class) describes a branch of execution-driven simulators that execute real programs or benchmarks and collect performance data at every processor clock cycle. Often, detailed cycle-accurate behavior is necessary for exploring design decisions or for understanding why programs perform better or worse during certain portions of their execution [9].

1.3 SimpleScalar Simulators

Developing and validating an execution-driven micro-architecture simulator is difficult and time-consuming. Few of these complex simulators are available to the research community; most academic researchers rely on a suite of simulators called SimpleScalar [9], which is freely available for non-profit research. The SimpleScalar suite was collaboratively developed by SimpleScalar LCC, the University of Michigan,

and the University of Texas with funding from the National Science Foundation and the Defense Advanced Research Projects Agency. SimpleScalar tools are widely used and respected in the research community. One third of all top computer architecture conference papers published in 2002 used SimpleScalar simulators [9].

The suite of SimpleScalar simulators includes several types of processor simulators of varying granularity and accuracy. Only two of the processor simulators provided by SimpleScalar (sim-outorder and sim-alpha) implement detailed, cycle-accurate modeling of out-of-order, speculative execution performed by the popular superscalar (multiple issue) processors. These simulators, though time costly, produce the cycle-by-cycle modeling necessary to understand the performance characteristics of Sandia's "cube" benchmark.

Sim-alpha is a validated model of the ALPHA 21264 processor with a simulation net error of 15% across 22 SPECCPU 2000 benchmarks. Figure 1.3 and Table 1.1 on the next page shows features of the ALPHA 21264 architecture implemented and configurable in sim-alpha [10]. Sim-alpha provides the benefit of modeling the dynamics of a manufactured microprocessor at a detailed, cycle-accurate level.

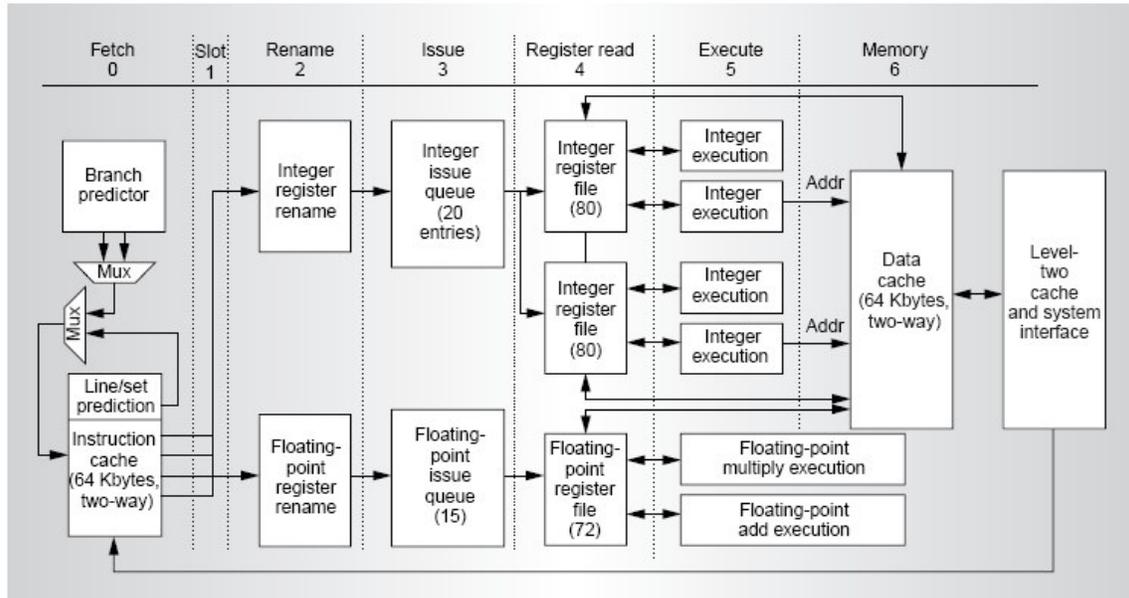


Figure 1.3: Block diagram of ALPHA 21264 architecture [10]

Feature	Default Configuration
Issue Width	4 instructions
Issue Queues	20-entry integer, 15-entry floating point
Reorder Buffer	80-entry buffer for tracking in flight instructions
Memory Management Unit	128-entry, fully associative data translation buffer 128-entry, fully associative instruction translation buffer
Functional Units	4 integer units that operate on specific class of instructions 2 pipelined floating point units, one for multiplication
Register File	80-entry integer, 31 architectural regs, 41 renaming, 8 PAL 72 floating regs, 31 architectural, 41 renaming
Instruction Cache (L1)	64KB, virtually addressed, 2-way set associative with set predictor
Data Cache (L1)	64KB virtually addressed, physically tagged dual-read-ported
L2 Cache	2MB virtually addressed, physically tagged, direct mapped
Branch Predictor	1024, 2-level local predictor 4096-entry global predictor with 2-bit saturating counters 4096-entry choice predictor choosing between above
Victim Buffer	8-entry victim data buffer
Load Queue	32-entry load queue
Store Queue	32-entry store queue
Address File	8-entry miss address file

Table 1.1: Features of ALPHA 21264 for configuration in sim-alpha

The sim-alpha configuration contains several types of caches and buffers. Figure 1.4 below shows the mapping of a main memory block into different types of cache configurations. Fully associative caches (a) and buffers allow any memory block to be placed in any cache location. In direct mapped caches (b), each memory block can be mapped to only one location in the cache (found by the memory block address MOD the number of cache blocks). Set-associative caches (c) contain sets to which memory blocks are mapped (found by the memory block address MOD the number of cache sets). Figure 1.3 shows how the Memory Block 12 maps into three types of cache configurations. The highlighted blocks represent locations to which the Memory Block 12 may be mapped. Memory Block 12 may be placed into any location of the fully associative cache. However, Memory Block 12 may only be placed into Block 4 of the direct mapped cache and only Block 0 or 1 (Set 0) of the 2-way Set-associative cache.

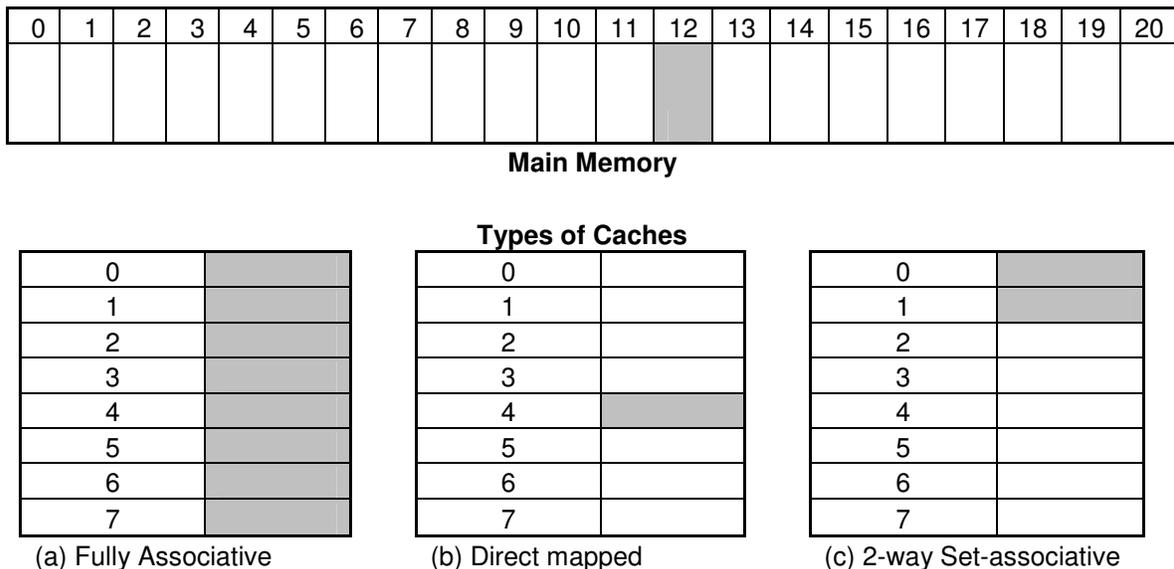


Figure 1.4: Mapping of blocks from main memory to different cache configurations

The remainder of this thesis is organized as follows: Chapter 2 examines other methods of optimizing the performance of workloads. Chapter 3 describes the approach to identifying the major performance constraints of the “cube” test problems and results of corresponding optimizations. Chapter 4 presents the technique and results of smart caching. Lastly, Chapter 5 summarizes the results and conclusions of this thesis.

2 RELATED WORK

As the complexity of microprocessor design increases, the processes of analyzing performance and pinpointing bottlenecks have evolved from just following logical hunches to detailed performance investigations. Performance evaluation techniques often differ from company to company and within the academic community as well. A standard, systematic approach to analyze and compare the performance of processor designs would be useful, but the vast difference existing between the various designs and implementations of microprocessors makes the task of developing such an approach a daunting one. Therefore, performance analysis and bottleneck identification have become an art form of choosing and combining reputable techniques and procedures that will result in accurate and informative results. The following chapter describes several approaches to workload performance analysis and presents techniques for improving the performance of microprocessors.

2.1 Performance Analysis

According to literature from PAID (Performance Analysis and Its Impact of Design) workshops held at the International Symposium on Computer Architecture, there are common elements of performance analysis that transcend genre. The workshops provide a platform to share advances in performance studies and establish some key methods to accurately analyze microprocessors. References from the PAID workshops identified CPI (cycles per instruction), path length (dynamic instruction count) and execution time as the primary components in determining the net execution cost (T) of a

particular program [11]. The net execution cost (T) is often used to measure architectural performance because it includes effects from the architectural organization (measured in CPI), the instruction set architecture (ISA) and compilation (effect seen in path length), and the clock speed (inverse of seconds per cycle).

$$T = (\text{path length}) \times \text{CPI} \times (\text{seconds per cycle})$$

Another concept arising from the PAID workshops, called separable components, describes the processes of separating components such as CPI into more expressive components. Separating CPI into two quantities that add up to the total CPI, such as infinite-cache CPI and FCE (finite-cache effects), helps to quantify the effect on CPI of the finite cache size decision [11]. This separation technique isolates and clearly demonstrates the impact of design choices on performance. The workshops also emphasize the importance of determining upper bounds on performance calculated with infinite queues, resources, and bandwidths. In addition to upper bounds, resource limit tests identify performance limits imposed by the actual finite resource sizes. These tests are performed by modeling or simulating all infinite resources except the test resource (actual size). By isolating components and resources, performance characteristic and bottlenecks become easier to identify.

The Scientific Computing Group at Los Alamos National Laboratory (LANL) characterizes workloads at the instruction level to demonstrate how aspects of micro-architectures will affect the performance [12]. These calculated parameters of the workload are used to estimate performance bottlenecks. On-chip performance counters, as compared to detailed simulations, provide a relatively fast and accurate method of measuring the instruction-level characteristics. The five instruction-level parameters (# of fp instructions, int instructions, memory instructions, L1 cache misses, and L2 cache

misses) are measured in terms of λ , a ratio of the total number of completed instructions to a subset of completed instructions (like completed floating point instructions, etc). In the first level of analysis, the L1 cache is assumed to be infinite and the growth rate, G , of queues (number of entries – number of exits) is tracked in terms of the average issue rates described above (λ_x), the ideal instruction issue rate of the microprocessor (β), and the hardware-defined dispatch rate from that queue (Δ_x).

$$G_x = \frac{\beta}{\lambda_x} - \Delta_x$$

Positive growth rates indicate bottlenecks arising without cache constraints, branching effects, or data dependencies and in this approach are considered to comprise the lower bounds for CPI (upper bounds for IPC). Multiple positive growth rates limit the number of in-flight instructions possible in the given architecture.

The second and third levels of this characterization and bottleneck detection technique both include cache limitations in the equation. The second level calculates the parameter Q , the maximum number of outstanding cache misses for a particular workload and architecture.

$$Q = \frac{(\text{outstanding memory instructions}) * \lambda_m (\text{completed inst.} / \text{completed memory inst.})}{\lambda_{L1} (\text{complete inst.} / \# \text{ L1 cache misses})}$$

The parameter Q indicates the value of extending the number of outstanding cache misses supported by the architecture. Finally, the third step observes the change in λ with increased cache size. Increasing the cache size should increase λ until the point at which a larger cache will not improve performance.

The instruction-level characterization method described above offers a quick and concise approach to identifying performance bottlenecks using performance counters and key parameters. The equations derived from these parameters clearly quantify and

highlight hardware constraints of the microprocessor. The method also provides a rough estimate for the best performing on-chip cache size. And, while the overall results of this approach are validated with results from empirical and statistical models, [12] noted that to improve the accuracy and usefulness of this method, the effects of branching and data dependency must be considered.

Real world applications and benchmarks are useful tools for performance studies especially for comparing the overall performance of different architectures. However, the complexity of these unique workloads makes it hard to pinpoint the cause(s) of performance bottlenecks. This difficulty prompted the creation of an adaptable synthetic benchmark specifically designed to help researchers and developers identify and quantify bottlenecks of specific architectures.

Synthetic benchmarks are artificial programs that are statistically representative of real world applications, but provide the added versatility of user-defined parameters for controlling the benchmark's behavior. For example, the adaptable synthetic benchmark, sqmat, represents the behavior of matrix multiplication and linear solvers with four user-defined parameters for isolating microprocessor bottlenecks [13]. The four variable parameters include the working-set size (N), the computational intensity (M), the level of indirection or noncontiguous memory access (I) and the irregularity of memory access (S). The sqmat benchmark operates on a number of matrices each of size $N \times N$. The number of matrices, L , is chosen to create an array big enough to exceed the size of the cache. The elements of the matrices are placed in memory according to the user-defined irregularity (S). Each matrix is accessed on the order of M^2 times with some degree of irregularity (I).

Varying the values of the parameters allows the user to better represent different types of workloads. High computation intensity (M) characterizes workloads solving dense matrices, whereas lower computational intensity and a reduced working-set (N) better represent dense matrix-vector or vector-vector operations. High indirection (I) and irregularity (S) are found in workloads using Finite Element solvers for dense matrices.

This synthetic benchmark was used to compare the performance of scientific workloads on four modern microprocessors - Itanium2, Opteron, Power3, and Power4. In this case, performance is measured in terms of the algorithmic peak performance (AP) based on the effective maximum FLOP (floating point operation per second) rate for each microprocessor. For workloads containing high computational intensity, defined as the ratio of floating point operations to load/store operations, Power3 and Itanium2 perform the best. However, the performance of Itanium2 decreases significantly with decreased computational intensity indicating a bottleneck between the registers and cache for floating point operations, perhaps caused by Itanium2's L1 cache which excludes floating point data.

Large working-set sizes test the effects of working with a data set that exceeds the register set. The results of this test indicate that Power3 handles register spills more efficiently. The test of indirection, chosen to mimic the compressed row format (CRS) of sparse matrices, checks for bottlenecks in memory bandwidth and instruction fetching. Indirection is introduced by compressed formatting method because a single access to matrix data requires gathering information from three compressed arrays (in CRS format) stored in three different memory locations instead of one access to the original, consolidated matrix. The Opteron, Power3 and Power4 each adequately handle the introduced indirection. The Itanium2, however, suffers a slowdown in performance of

between 1.5 to 5.4 times due to the introduced indirection, a slowdown unexplained by [11]. The final test of irregular memory access patterns examines the effect of cache misses on the architecture. Itanium2 handles cache misses effectively surpassing the Opteron in a close second.

Detailed results of this performance study are presented in [13], however the above results demonstrate the effectiveness of using synthetic benchmarks to isolate and identify the source of performance degradation. Synthetic benchmarks characterize only a narrow spectrum of workload behaviors as compared to real applications or benchmarks. Therefore, the results of the sqmat benchmark performance studies are useful for bottleneck detection, but are best used in conjunction with other forms of performance analysis.

2.2 Performance Improvement Techniques

Commonly explored formulas for improving performance of high-volume scientific workloads consist of increasing resources such as functional units (arithmetic logic units), registers and especially the cache. Intel's Itanium processor was developed using the above approach to better manage high-volume scientific workloads [14,15,16, 17]. However, the vast quantity of data and operations performed by these workloads often overwhelms on-chip caches, even large caches, preventing optimal performance. Techniques to help reduce memory-access latencies by reducing cache pollution and unnecessary accesses include selective fill [18], load redundancy [19], and split and victim caches [20,21] explained in detail in the following sections. Also, in 2002, Cray released a shared-memory multi-vector processor (X1) that implemented a technique

called scalar caching [22]. This remainder of this chapter will examine and report the performance improvements of the techniques and implementations.

2.2.1 Itanium2 Processor

The Itanium processor was developed by Intel to better meet the computing needs of the high-performance technical computing and large enterprise communities. The design concept by Intel for the Itanium supplied a generous amount of resources along with powerful compilers optimized for parallel execution. This approach termed EPIC (Explicitly Parallel Instruction Computing) was created to exploit and extract the inherent parallelism that exist in most scientific, looping workloads. Unlike most processors categorized as either CISC (Complex Instruction Set Computers) or RISC (Reduced Instruction Set Computers), Itanium 2 decodes words (or long instruction strings each containing several instructions) making it the first general use processor in the realm of VLIW (Very Long Instruction Word) processors. Scalability, ability to maintain performance in a multi-processor environment, is another feature necessary for scientific supercomputing. Itanium was developed to perform as the key component of large scale, supercomputing systems.

In 2004, Intel joined with manufacturer California Digital and the University of California at Lawrence Livermore National Laboratory (LLNL) to build a supercomputer composed of 4,000 Itanium2 processors [14]. Hal Graboske, Deputy Director of Science and Technology of LLNL says of the project:

“Thunder (the above mention supercomputer project) will serve a critical role supporting the Lab’s mission to drive unclassified science and technology for

multiple program areas. Intel Itanium 2 processors address capacity and capability issues facing national security and science programs, with a long-term goal to develop a viable path to petaFLOP's-scale computing. [14]"

Organizations like NASA and companies including Wells Fargo Bank have also chosen the Itanium2 to fulfill their diverse computing needs.

The extensive resource enhancements implemented on the Itanium processor include three-levels of generous on-chip cache, an enlarged register file, more functional units and among others a bus system developed for efficient multiple processor communications. The three (physically indexed and tagged, non-blocking) caches include split level one data and instruction caches each 16KB, 4-way set-associative, and double ported with 64-byte lines. The L1 instruction cache is fully pipelined supplying six instructions per cycle. The L1 data cache supports two simultaneous loads and stores and also does not cache floating-point data, only integer. The 256KB, 8-way set-associative L2 cache is unified and stores instructions and all types of data memory. Finally, the on-chip L3 cache is customer specified with sizes ranging from 1.5 to 9MB. The single ported L3 cache is 12-way set-associative, fully pipelined and has a maximum transfer rate of 32GB per cycle [15].

Other significant expansions include the register file which has 128 registers, 64-bit general registers for integer storage, 128 floating-point registers each 82-bits wide, 64 one-bit predicate registers, and 8 branch registers also 64-bits. The Itanium2 processor provides 21 execution units - 6 multimedia units, 6 integer units, 2 floating-point units, 3 branch units, and 4 load/store units. These robust structures exist to assist in the 64-bit, 1 GHz processing of the six to eight parallel instructions/operations per cycle [15].

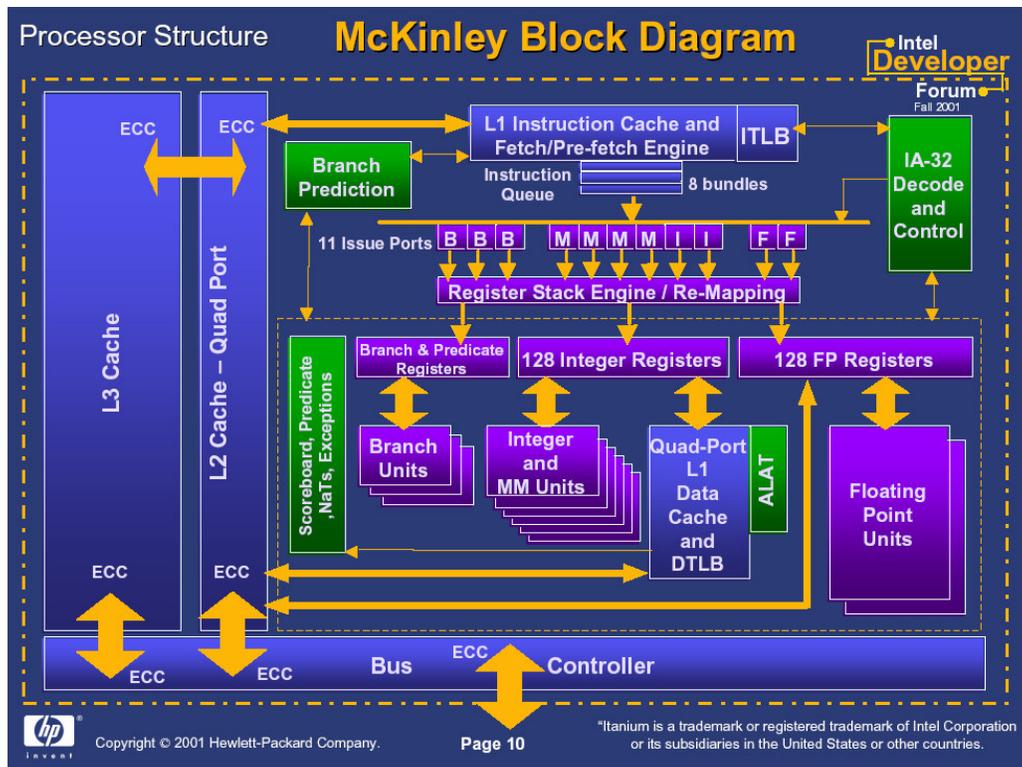


Figure 2.1: Itanium2 block diagram [16]

Performance studies on the Itanium2 processor with 3M of L3 cache show that an Altix system with 64 (Itanium2) processors compared against the same configuration of Hewlett Packard 750MHz PA-8700 processors and 1 GHz, UltraSPARC III Cu processors performs 3.94 and 1.95 times faster, respectively, on SPEC 2000 floating-point benchmarks [17]. On the uniprocessor level, however, the Itanium 2 performs only 0.57 times faster than the 1.15GHz Alpha 21364 and 0.23 times better than the 1.7 GHz IBM POWER4 processor also on SPECfp 2000 [18]. These statistics speak volumes about the scalability of the Itanium2 performance, but this processor built to perform on floating-point workloads does not exhibit the expected performance improvement on the uni-processor level. This observation will be confirmed and discussed in later chapters.

2.2.2 Selective Fill Data Cache

Many researchers believe that the key to overcoming the performance bottleneck observed in most scientific applications can be accomplished by improving the performance of the cache. Even substantial on-chip caches, like in Intel's Itanium2, can become polluted by the massive amounts of data processed in these workloads. The idea of a selective fill cache operates on the notion that minimizing the cache pollution enables the cache to benefit from the expected temporal locality exhibited in scientific processing. The Selective Fill Data Cache (SFDC) method dynamically filters data with low temporal locality and prevents it from caching [19].

The Selective Fill Data Cache is implemented in hardware by three modifications and additions. First, the existing data cache is concatenated with an additional "used" bit per block, similar to a dirty bit. The "used" bit for a particular block is cleared upon a new block entering that position in the cache and set if the block is accessed again while in the cache.

Next, an additional and separate direct-mapped cache with the same number of sets as the data cache functions as the Cache Fill Policy Table (CFPT). The CFPT holds the tags of the recently evicted blocks that were not reused while in the cache. Furthermore, the table also utilizes a two bit saturating counter to count how many times a particular block is evicted without reuse for the purpose of profiling access patterns. Entries in the CFPT are replaced when a new block (not in the table) is evicted from the data cache. All block tags residing in the CFPT are not allowed back into the cache [16].

Thirdly, a bypass buffer is added to hold blocks that must bypass the data cache. The small cache buffer, like a victim buffer, sits between the data cache and the next

level of memory (L2 cache) and is intended to hold bypassed blocks close at hand for a short time to reduce the risk of mispredicting blocks that should have been cached.

Experimentation shows that buffers bigger than $1/16^{\text{th}}$ the size of the data cache exhibit diminishing returns.

The SimpleScalar 3.0 tool set is used to test the effectiveness of the Selective Fill Data Cache technique among 4 SPEC2000 integer benchmarks - 175.vpr, 181.mcf, 197.parser, and 164.gzip. Four configurations of L1 Data Cache (8K and 16K direct-mapped, 8K and 16 2-way set-associative) are simulated with no modifications, using SFDC, and with a victim cache equivalent to the bypass buffer. The SFDC method shows slight improvement over the other methods for the mcf benchmark, the benchmark that exhibited the highest miss rates. The parser benchmark results display that the SFDC method actually degraded the performance of the 8K direct-mapped cache compared to the un-modified data cache. Also, the simple victim cache produces lower miss rates than either the un-modified or the SFDC across the board for the parser, vpr, and gzip benchmarks [19]. The complete results for the SFDC are shown below in Figure1. While the concept of the Selective Fill Data Cache is interesting and promising, this result shows that simple alternatives are currently more advantageous and SFDC requires further investigation and improved implementation.

VPR Benchmark Results

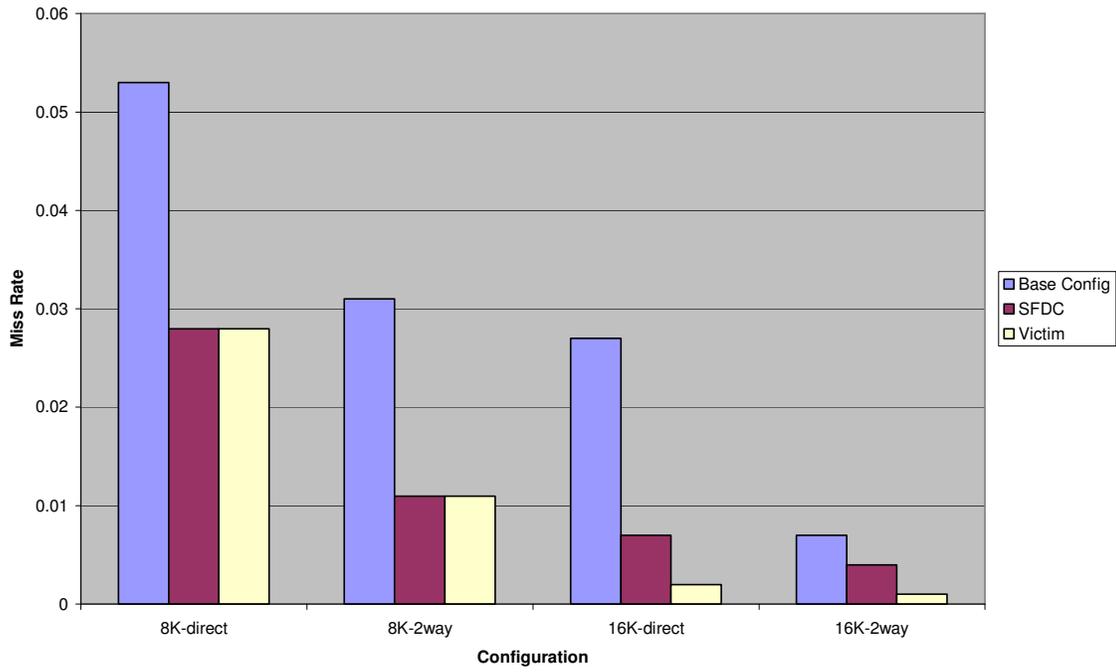


Figure 2.2: VPR benchmark results of Selective Fill Data Cache method and victim cache miss rates [19]

MCF Benchmark Results

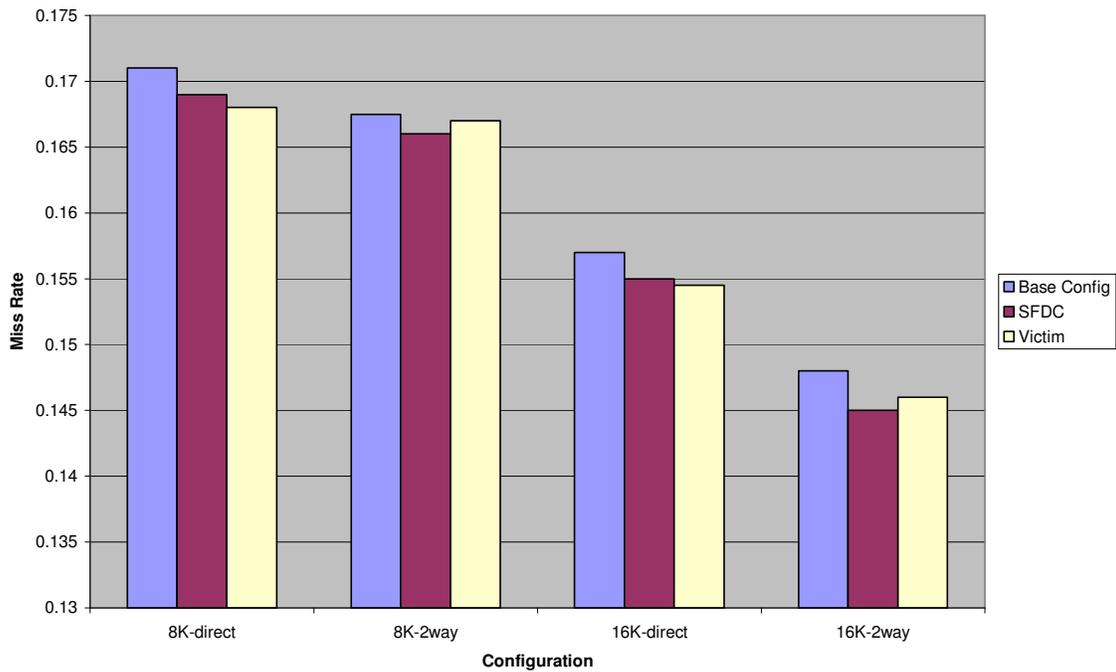


Figure 2.3: MCF benchmark results of SFDC method and victim cache miss rates [19]

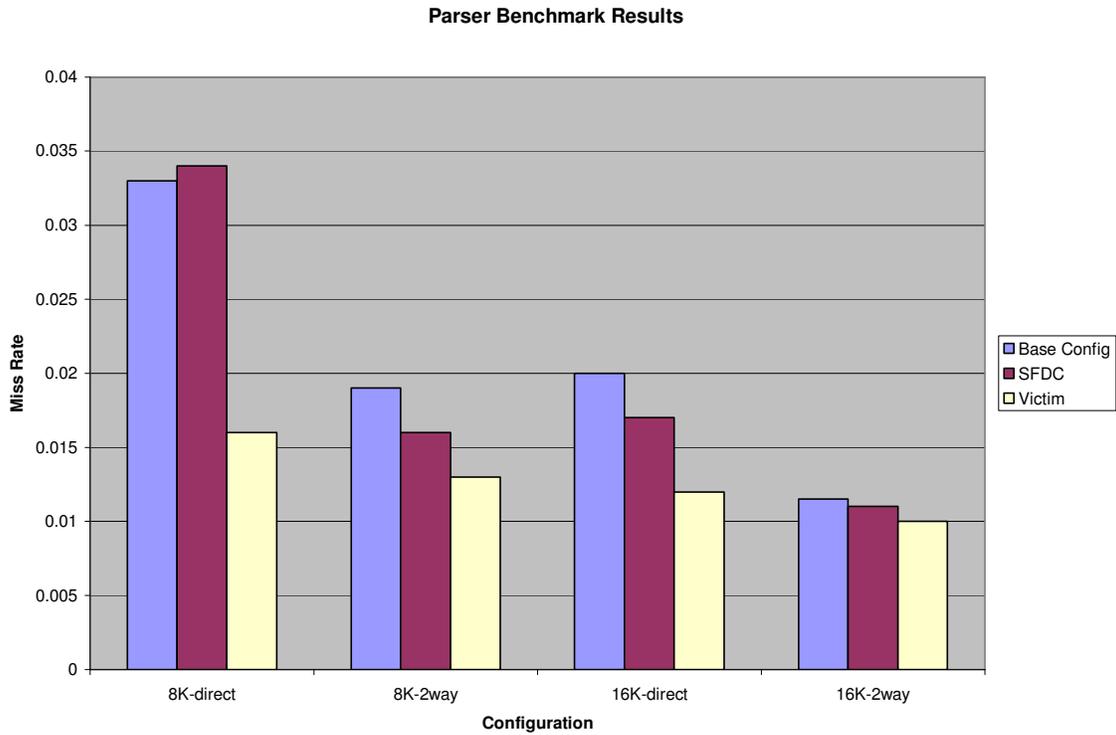


Figure 2.4: Parser benchmark results of SFDC method and victim cache miss rates [19]

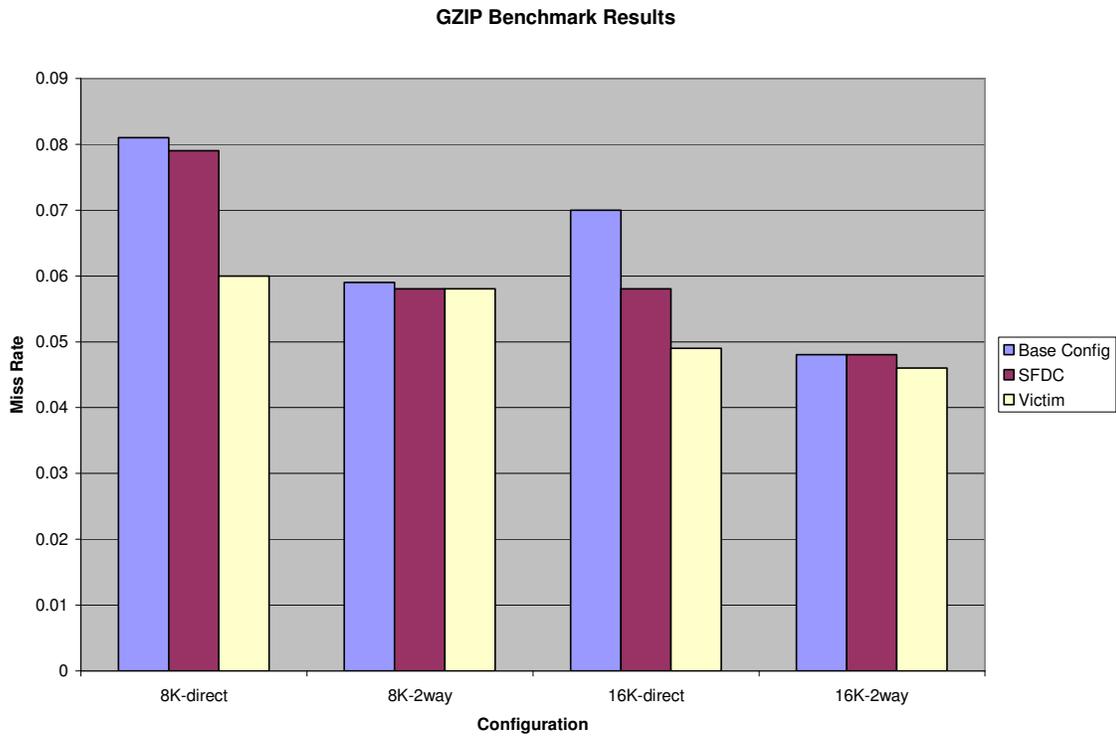


Figure 2.5: GZIP benchmark results of SFDC method and victim cache miss rates [19]

2.2.3 Load Redundancy

Another alternative for improving cache memory performance reduces the number of redundant loads. Load instructions often take more time to execute than any other instruction because they must incur the access penalty of the cache, and the occasional memory latency, in order to complete. While avoiding loads altogether is impossible and impractical, reducing redundant loads reduces execution time while preserving the correct functioning of the program.

[20] defines load redundancy as the act of reloading a value into a register that already contains that same value. The amount of load redundancy existing in a benchmark or program can be calculated by stepping through the executing program with the GNU debugger, counting redundant loads. Of the SPEC95 benchmarks compiled with optimization, lisp and ijpeg exhibited the highest (22.81% of loads) and lowest redundancy (14.18%), respectively.

A load within the malloc function produced the most redundancy in some benchmarks. In this case, the redundancy that could be eliminated with smarter programming was not resolved by the compiler. Many times redundancy arises from subroutines that contain complicated controls and are not easily optimized by compilers.

The maximum performance achieved by eliminating all redundant loads and the expected performance increase by implementing a load redundancy predictor are calculated across four SPEC95 benchmarks. By modifying SimpleScalar to detect and bypass redundant loads, the SPEC95 benchmarks demonstrate the expected performance increase within each program. Each of the 4 SPEC95 benchmarks tested contained a load redundancy percentage between 14.19% (jpeg) and 22.81% (lisp). The IPC of the modified and normal SimpleScalar execution of the benchmarks showed that bypassing

all redundant loads increased IPC from 2% to 10% across the benchmarks. This metric serves as the upper bound for increasing performance with this method [20].

A more realistic implementation of eliminating load redundancy is simulated by adding a load predictor (a modification of a branch predictor) that compares a decoded load instruction to the instruction addresses stored in a prediction table. The prediction table holds recently executed load instructions. Therefore, if the current load matches a load in the table, the load is bypassed. Various table prediction sizes are tested, and those with over 128 sets (each with 2 entries) showed diminishing returns. Results from this implementation show increased IPC from 1% to 8% when compared against normal simulation of the benchmarks [20]. These results may be slightly optimistic considering the simulated results charged no penalty to resolve missed predicted loads.

This method of improving performance is a simple hardware implementation that resolves load redundancies missed by gcc compiler optimizations. While the actual performance gains are modest, the simplicity of this hardware method makes it attractive. For those working with highly redundant programs, this technique may actually be easier than modifying a compiler to prevent the redundancies. In conclusion, the load redundancy results reveal that a slight performance increase can be achieved by reducing redundant loads, but it also proves that load redundancy is not the leading cause of the performance degradation in cache and memory latencies.

2.2.4 Split and Victim Caches

Researches have strived to design cache memory able to exploit the natural access patterns that exist in most workloads. Temporal locality describes the tendency of programs to re-reference a location in the cache that has recently been referenced. Spatial locality explains the high probability of accessing data nearby data just recently accessed. Cache design choices like block size and associativity (number of blocks in set) each exploit different locality. Larger blocks of data brought into the cache on a miss help caches take advantage of spatial locality by bringing in more nearby data on each access. Caches with more blocks per set best serve temporally local programs by allowing more recently accessed data blocks to remain in each set. However, because access tendencies change considerably between different programs and even in different phases of a single program, a single cache and configuration often never achieves optimal performance.

A common technique to improve cache performance and to better accommodate changing access patterns involves splitting the first level of data cache into two separate caches. Each separate cache is configured to best manage the data stored in that cache. Several split cache types and schemes have been simulated to test the effectiveness of each arrangement [21]. The first example split cache configuration (DUAL) consists of two independent cache organizations. One cache with bigger block size for spatially local data and another with one word blocks for temporally local data. In the DUAL scheme, data is classified as either spatial, temporal, or neither at runtime using a hardware history table. Spatial and temporal data are cached in their respective caches while data exhibiting neither is bypassed to the next level of memory. The Split

Temporal Spatial Data Cache (STS) [21] method uses the same configuration as above, but classifies data during compilation and attaches counters to data to check locality and correct data mislabel during compilation.

Another approach to splitting the data caches comes from the trend that data arrays and structures tend to be more spatially local, whereas scalar data exhibits more temporal locality. For this reason, the Scalar-Array Data Cache [21] caches array and scalar data separately. In this case, the compiler classifies data signaling the controller to put the data into the corresponding cache. Another method, the Cacheable Non-Allocatable Model (CNA) [21], classifies data as cacheable or non-cacheable depending on a predication counter that holds the cache hit history of data access instructions. Instructions commonly causing cache misses will therefore not be cached again. A modified CNA cache scheme, the Memory Address Table (MAT) [21], stores the reuse information of each kilobyte block in a direct-mapped hardware table used to determine whether or not data should be cached.

Finally, data cache memory can be divided into one large main cache and a smaller cache (called a Victim cache) acting as a buffer before data is stored to the next level of cache or main memory [21]. Similar to the Victim cache, a Filter cache is a small cache buffer that holds the most frequently referenced data blocks as indicated by a counter associated with each block. The ABC (Allocation by Conflict) [21] victim caches check an extra conflict bit to decide whether to evict a block on a conflict miss. Every cache hit to a block resets the conflict bit, and recently hit blocks (conflict bit equal to zero) remain in the main cache on a miss while the blocks that cause the miss will be deposited into the small buffer cache.

To compare the relative performance of the cache enhancement techniques described above, one configuration from each approach is simulated with identical conditions and benchmarks. A modification of the DUAL cache configuration, called the Nontemporal Streaming Cache (NTS) [21], dynamically routes data between the two split level one data caches. NTS allocates one cache for data that is strictly temporal and another cache to hold data that exhibits both temporal and spatial locality tendencies. NTS was chosen to represent caches that separate data according to locality because it is widely used and documented. In [21], the Victim cache scheme used for comparison consists of a large, direct-mapped main cache (16Kbytes) with a fully associative cache buffer (2-Kbytes). Lastly, the MAT cache represents those configurations that cache scalar data in a separate cache from array or structure data.

When comparing each split cache configuration to an equivalent single cache, the Victim cache shows the greatest speedup for direct mapped caches among integer (speedup of 7.68 over single cache) and floating point (speedup of 3.67) SPEC benchmarks. NTS performed best in four way set-associative caches with a speedup of 10.29, while the Victim cache still demonstrated greater speedup (3.78) among the SPEC floating point benchmarks. Although the MAT scheme showed improvement, especially with increased block sizes, it did not improve cache performance as much as the NTS or the Victim cache setup [21].

One study researched the effects of combining the techniques of the MAT cache, Victim cache, and stream buffer [22]. The stream buffer, in this case a 10 line, fully associative cache, serves as a place to store prefetched data brought in on a missed cache block. This integrated solution implements a 4-Kbyte direct-mapped Scalar cache, a 4-Kbyte direct-mapped cache, 8 line fully-associative Victim cache, and the stream buffer

mentioned above. This configuration achieved a 55 % improvement over a single cache configuration of the same size executed on SPEC floating point benchmarks [21].

Moreover, the Cray X1 (a shared-memory multi-vector processor) is one example of a working system that implements the concept of a separate cache for scalar data. The Cray X1 system has a strictly scalar, level one data cache of size 16 KB. The 2 MB non-scalar cache, called the E-cache, contains all the vector data references and also all the references that miss the scalar cache [23]. Performance studies demonstrate the Cray X1 “achieved high raw performance relative to the Power (IBM) systems for the computationally intensive applications”, but other systems still completed with faster runtimes [24]. Though the X1’s performance is certainly attributed to many architectural components and design attributes, the results prove that split cache configurations are an achievable, workable solution to be explored.

3 Methodology for Performance Analysis

3.1 Performance Analysis Tools and Metrics

The previous chapter describes several methods and formulas for measuring the performance of microprocessors and for identifying the bottlenecks limiting their performance. Most of these methods look first at Cycles Per Instruction (or IPC) to gauge overall performance. Calculating or simulating the maximum IPC provides an upper bound for comparison and identification of bottlenecks. The theoretical upper bound of IPC comes from the maximum number of instructions the architecture can issue/commit per cycle. However, the more commonly used measure of maximum IPC comes from the IPC calculated or simulated with no cache misses or access penalties (assuming infinite cache), no branching effects, and no data dependency stalls. Beyond IPC, the utilization and performance metrics of major components and queues become important for specific bottleneck identification.

The “cube” test benchmark previously introduced serves as the primary benchmark for performance evaluation for this research. This benchmark provides the opportunity to observe the performance of a fully functioning finite-element analysis tool without the hassle of providing actual input data. The test problem solves a system of equations associated with a mesh of arbitrarily chosen data points. Upon the recommendation of SNL, the mesh will always be organized according to the CRS format type and ML will be used to precondition each mesh before solving. The size, shape and density of the mesh are user-defined parameters allowing the user to determine the physical dimensions of the cube mesh and in turn how intensely the benchmark taxes the architecture of the microprocessor. Performance results from the “cube” benchmark are

compared to results from SPEC2000 benchmark fmad3, characterized by similar functioning. The benchmark fma3d is a finite element method, written in Fortran, designed to simulate in the inelastic transient dynamic response of three-dimension objects subjected to suddenly applied loads [1]. Results from the fma3d benchmark will serve as a comparison for the results of the “cube” benchmark, reinforcing the performance results for FEA and scientific workloads.

Performance data is collected from the sim-alpha simulator, also described in a previous chapter, along with verification by means of another cycle-accurate SimpleScalar simulator, sim-outorder. Another important method of verification includes comparing simulator results with similar performance metrics of current microprocessors. For this reason, the configuration file describing sim-alpha’s architecture is modified to best resemble the Itanium2 microprocessor, allowing performance data to be compared. Although the Itanium2 has a VLIW architecture, major structures in the architecture such as registers, caches, and queue sizes can be imitated for a general comparison of performance. The architecture features of the Itanium2 and their representation in the sim-alpha configuration are shown in Table 3.1.

Itanium Configuration of sim-alpha		Itanium 2 Architecture
Architecture	SuperScalar	VLIW
Processing	Out-of-order	In-order
Fetch Width	6	6
Commit Width	6	6
Physical Integer Registers	128	128
Physical Floating- point Registers	128	128
Functional Units	12	11
Cache Configuration	2 Levels	3 Levels
DL1	256 KB 8-way set- associative	16 KB, 4-way set-associative (no floating point access)
L2	1.5 MB, 12-way set-associative	256 KB, 8-way set-associative
L3	none	3 MB, 12-way set-associative

Table 3.1: Itanium 2 configuration on sim-alpha versus standard Itanium 2 architecture

The Itanium2 microprocessor was chosen for this comparison for a variety of reasons. The Itanium2 was developed largely for the purpose of improving performance of scientific workloads much like the “cube” benchmark and others. Also, the Itanium2 architecture provides a generous number of performance counters allowing for the collection of several performance statistics simultaneously. And, because cycle-accurate simulators, like sim-alpha, take several days, even weeks to complete a simulation, the Itanium2 provides a platform on which several problems sizes and configurations can be efficiently tested and analyzed (in real time). Thus, the Itanium2 provides a better coverage of the “cube” benchmark analysis and is useful in identifying information about specific areas where more detailed information, from simulators, is necessary. Performance counters, if available, offer the quickest and perhaps the most accurate means of gathering initial performance information for a particular application or benchmark. Simulators, conversely, contribute more detailed performance data with the

added capability of modifying architectural features to observe their effect on performance. When used in conjunction, these two performance evaluation tools are powerful resources.

The organization of this chapter is as follows: Section 3.2 describes the process of choosing the most appropriate problem sizes for complete performance analysis and comparison. Section 3.3 presents the performance and bottleneck analysis methodology and results. Section 3.4 summarizes the results of this method, the conclusions arising from these results, and our plans for further research.

3.2 Problem Size

The capability of simulating a great variety of problem sizes, shapes and densities provides for excellent performance coverage and mounds of performance data. The only limitation to uni-processor problem size selection is processor memory. The performance analysis of the “cube” benchmark is performed on a Beowulf cluster of eight machines. Each of the eight machines utilizes 2GHz, dual AMD processors with 2GB RAM. To avoid over running the 2GB of RAM and thereby suspending operation of the “cube” benchmark, problems sizes which require over 1M resulting equations will be excluded. The three user-defined parameters that determine problem size include the width (W), depth (D), and the number of the degrees of freedom (DofPerNode). The number of equations resulting from problem sizes with parameters mentioned above is found using the following equations [4]:

NNodes = the number of nodes in a mesh

NEqns = the total number of equations for a particular problem size

$$\text{NNodes} = (W+1) * (W+1) * (D+1)$$

$$\text{NEqns} = \text{NNodes} * \text{DofPerNode}$$

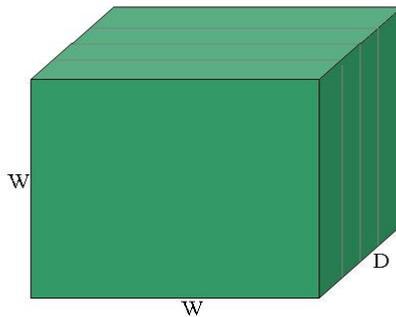


Figure 3.1: Visualization of 3 dimensional “cube” benchmark mesh from [4]

Each mesh created by the “cube” benchmark exists in three dimensions. The width and depth are the only dimensions of the 3D object that can be specified by the user with the assumption that the height is always equal to the user-defined width. The other user-defined parameter, DofPerNode, describes the number of parameters (in FEA, the number of physical properties or attributes) monitored at each node. Increasing the number of degrees of freedom per node makes for a denser mesh, more equations and higher computational intensity.

3.2.1 Problem size results for Alpha configuration

While it is impossible to simulate every potential size, shape, and density of a mesh, gaining insight about the performance trends among different configurations is important. Results from Itanium2, along with comparable simulations from sim-alpha (Alpha and Itanium2 configuration), are used to measure performance among different problem sizes. To better understand the effects of varying W, D and DofPerNode on performance, the number of equations for a mesh is held constant. Two problem sizes, each with a constant number of equations, are evaluated with for varying W, D, and Dof. The two problem sizes, equations equaling 175,616 and 274,625, are chosen to overwhelm (operate on a mesh with more nodes/values than can be stored in the cache) the DL1 cache and L2 cache respectively. The following Figure 3.2 shows the results of varying the W, D, and DofPerNode on IPC as a measure of overall performance for the Alpha configuration.

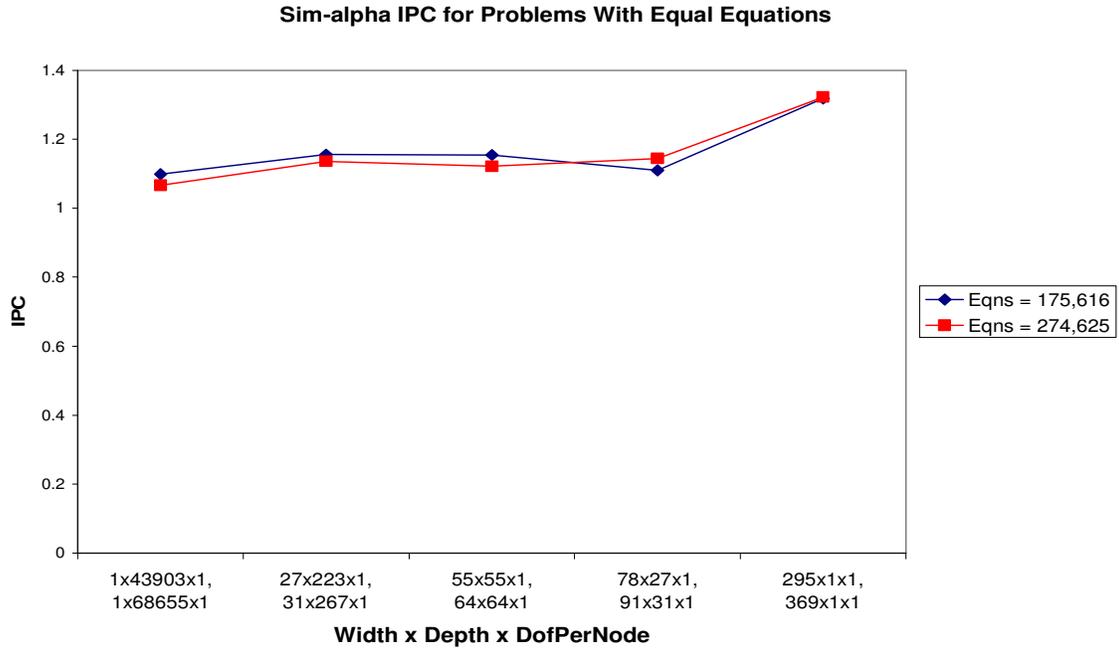


Figure 3.2: Sim-alpha performance (in term of highest IPC) for varying width and depth with equal equations

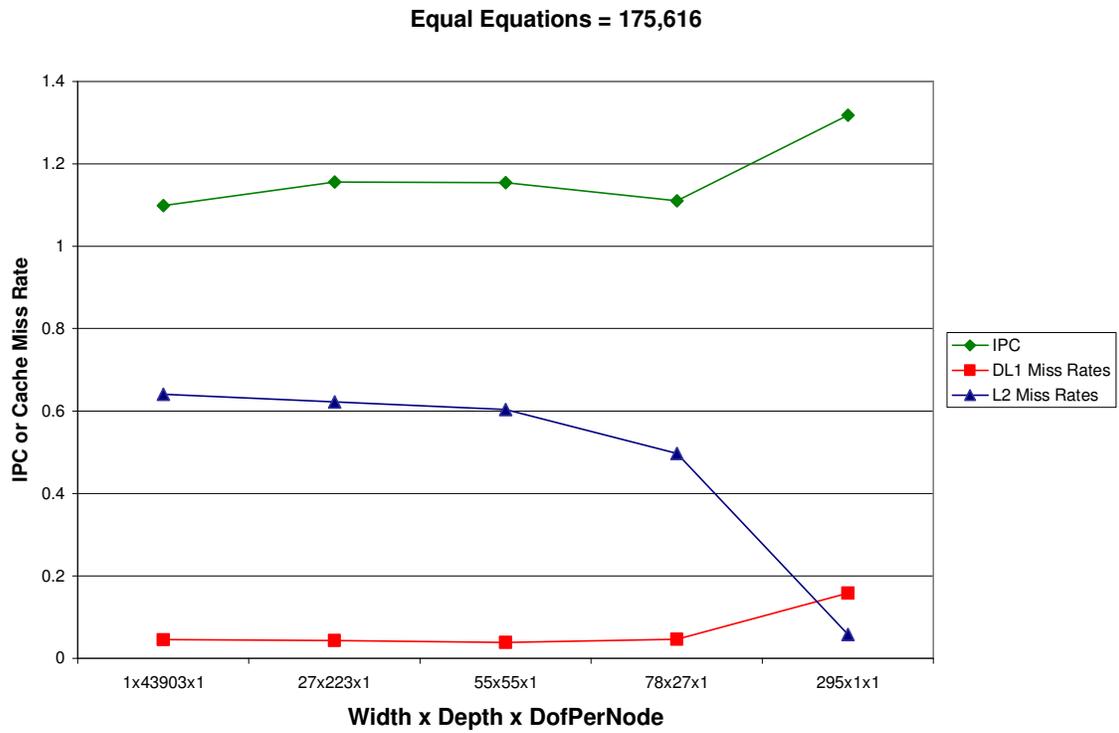


Figure 3.3: Sim-alpha performance (in terms of highest IPC and lowest cache miss rates) varying width and depth with equal equations

IPC of Different Problem Shapes

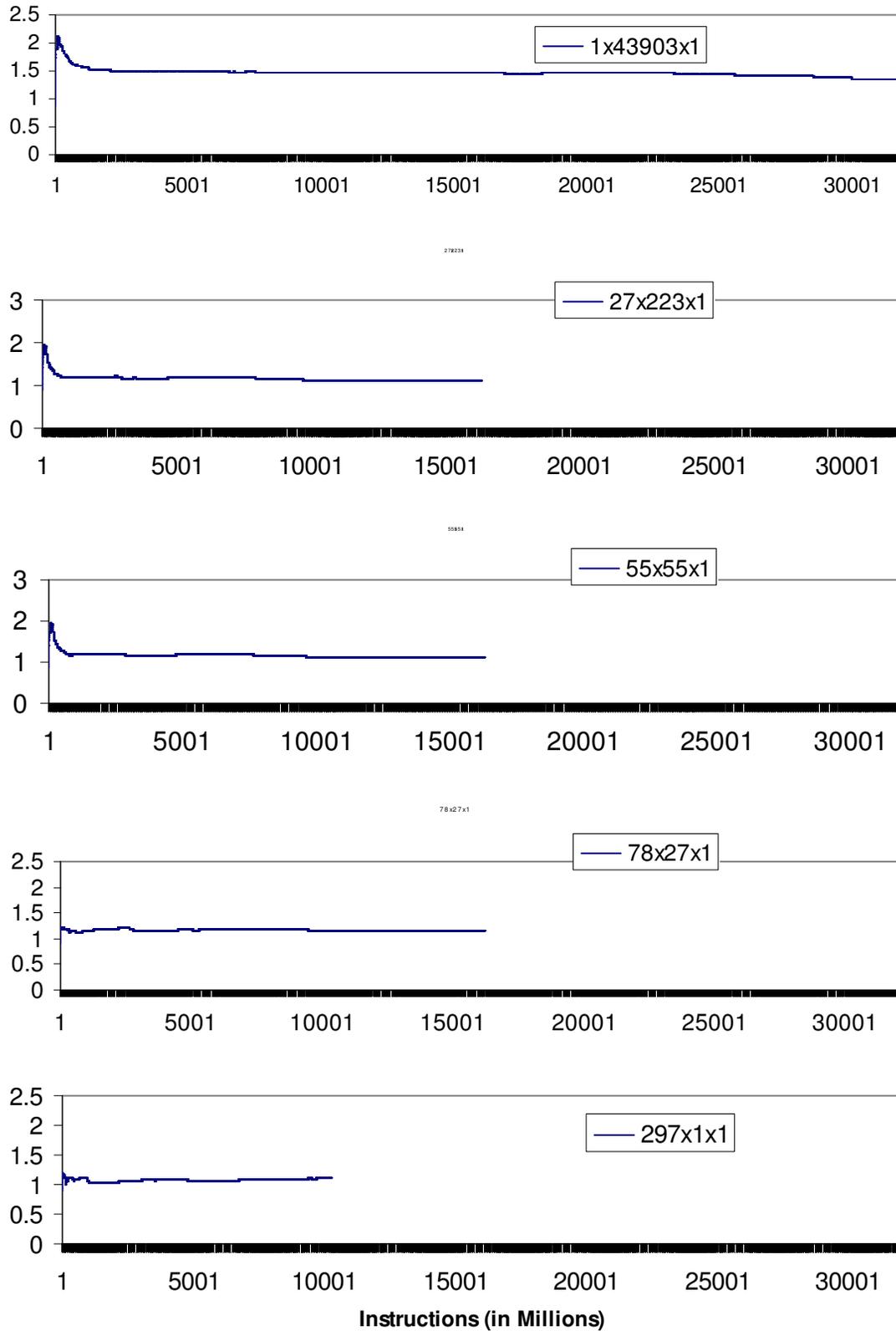


Figure 3.4: IPC and # of instructions for equations = 175,616

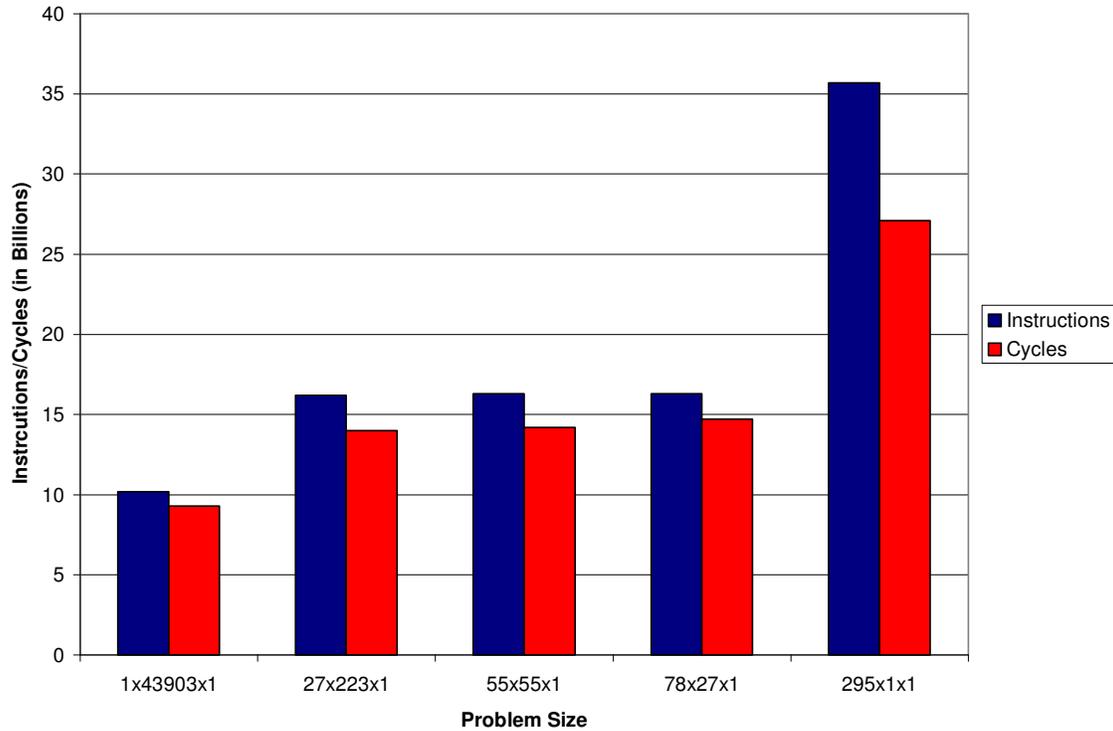


Figure 3.5: Number of instructions and cycles executed by each problem size, equations = 175,616

Figure 3.3 reveals the trends of overall performance and caches miss rates among different shapes of meshes with a constant number of equations. Figure 3.2 displays the highest IPC achieved by the problems with the largest width and smallest depth. The improvement in IPC produced by problems with larger widths than depths appears to be caused by a significant decrease in L2 cache miss rates. Even though the decrease in the L2 miss rate is very drastic, the resulting increase in IPC is only 0.22 instructions per cycle. Also, Figures 3.4 and 3.5 reveal that although the problem 295x1x1 achieves the highest IPC, it also requires the most instructions to complete. Therefore, the problem 1x43903x1 will actually execute more quickly than the other problems on this architecture. Thus, the Figures demonstrate that the problems with larger depths complete faster and the problems with larger widths better utilize the architecture by completing more instructions per cycle.

The results of the problem size simulations indicate little change in IPC among problems with different dimensions. In addition, the maximum difference in IPC among the two problem sizes (equations equal 175,616 and 274,625) is less than 0.034 instructions per cycle as seen in Figure 3.2. However, the number of cycles executed by different problem sizes and shapes does vary greatly. Even so, our research is focused on improving the microarchitectural performance at the uniprocessor level, thus our concentration will be on improving IPC. Because the two problem sizes exhibit such similar performance behavior in terms of IPC, the performance of these problems can be represented by one simulation of either of the problem sizes and any of the shapes simulated above. Therefore, all subsequent simulations will be performed on the problem with equations equal to 175,616 and $W=55$, $D=55$, and $Dof=1$. Validation for our choice of $Dof=1$ will be presented below in the Itanium2 results section.

3.2.2 Problem size results for the Itanium2 processor

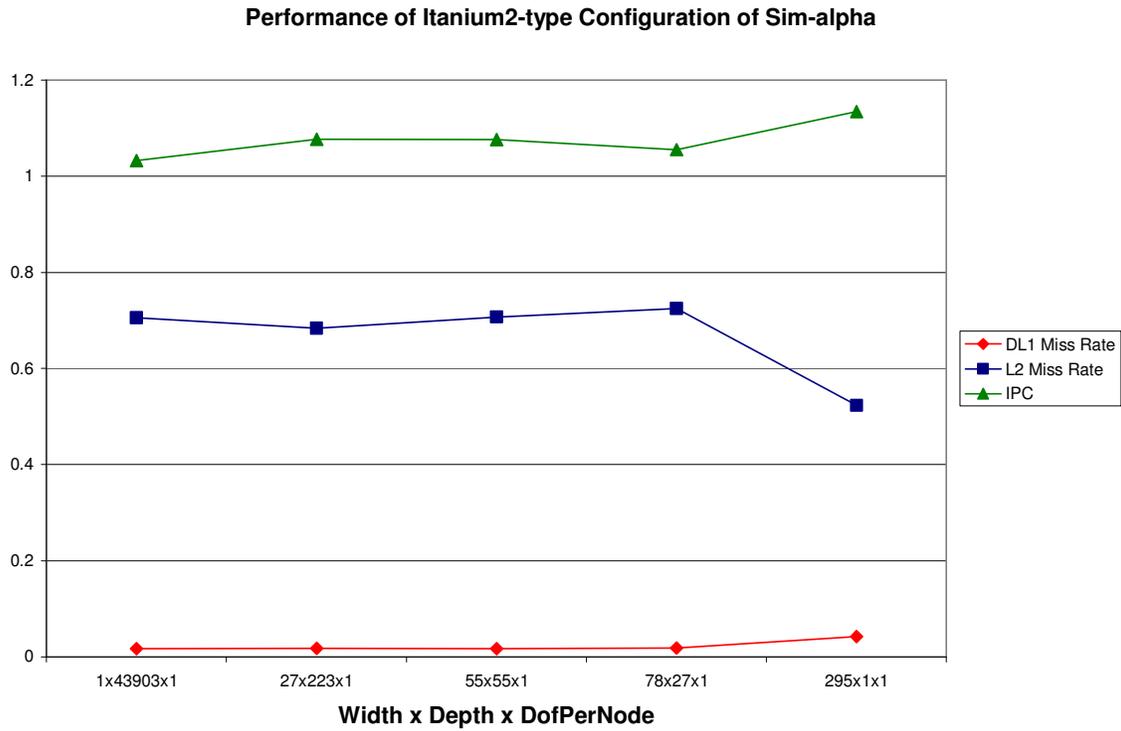
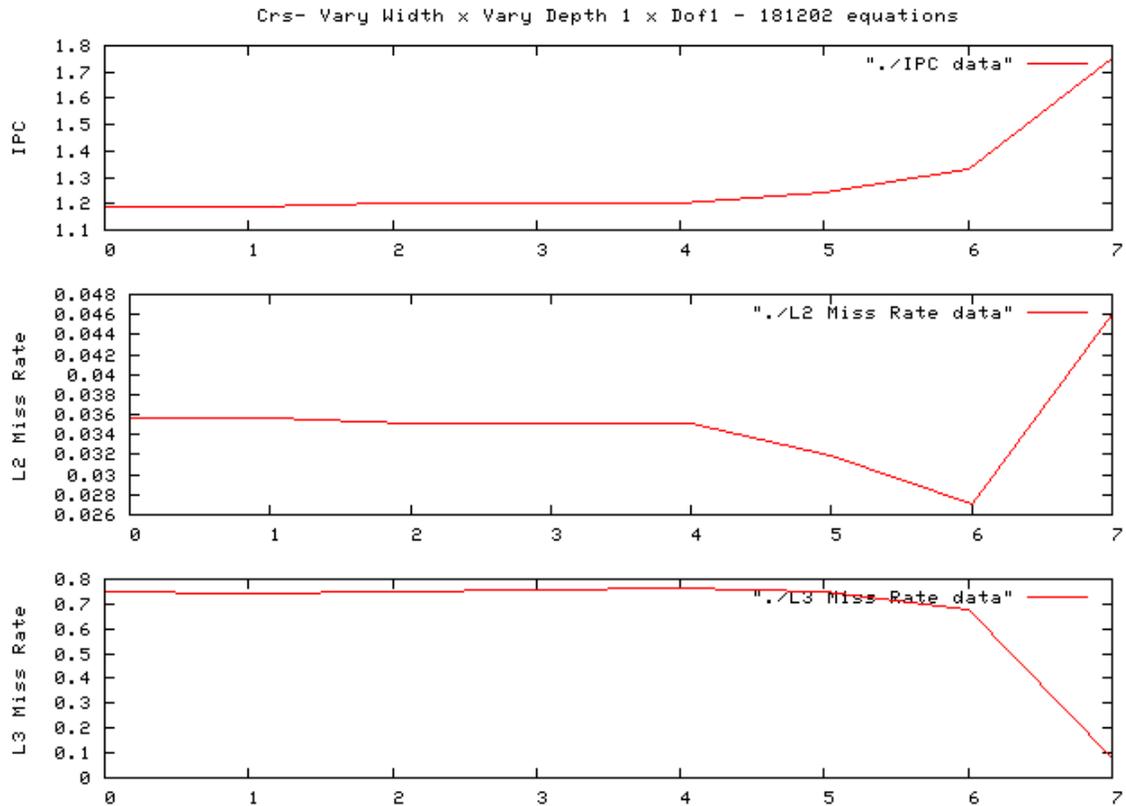


Figure 3.6: IPC and cache miss rates for Itanium2-type configuration of sim-alpha



Problem	Width	Depth	Dof
0	1	300	1
1	5	173	1
2	10	128	1
3	50	59	1
4	75	48	1
5	100	41	1
6	150	31	1
7	300	24	1

Figure 3.7: IPC and cache miss rates for Itanium2, varying width and depth with equations = 181,202

Figures 3.6 and 3.7 are used to validate the results shown in Figure 3.3 (Alpha configuration) showing performance trends among problems of different shapes. Figure 3.6 displays the performance of the sim-alpha simulator configured to mimic the Itanium2 processor. Figure 3.7 presents the actual results from the “cube” benchmark executed on the Itanium2 processor. Both Figures 3.6 and 3.7 show the same trends as the Alpha configuration results shown in Figure 3.3. The problems with the greatest width, smallest depth achieve the best performance in terms of IPC, but overall again the performance variation among problems of different shapes is less than 0.6 instructions per cycle.

In addition, the results of an Itanium2 problem size study varying the degrees of freedom provides the evidence leading to our decision to concentrate on a problem size with Dof=1, shown in Figures 3.8 and 3.9. This study is performed on the Itanium2 processor because problems can be evaluated on the Itanium2 in a fraction of the time it takes to simulate the same problem on the sim-alpha simulator. Using the Itanium2 processor, many problem sizes, shapes and varying degrees of freedom can be simulated in a matter of days, instead of weeks with a simulator. Also, the similarity in results

collected from the Itanium2 processor and the sim-alpha simulator provides validation of these results and confidence in using data from each method. Figures 3.8 and 3.9 below indicate some variation in performance caused by changing the degrees of freedom of the “cube” benchmark, but the y-axis scales reveal that this variation is very small. Therefore, our subsequent studies will be performed on problem with one degree of freedom.

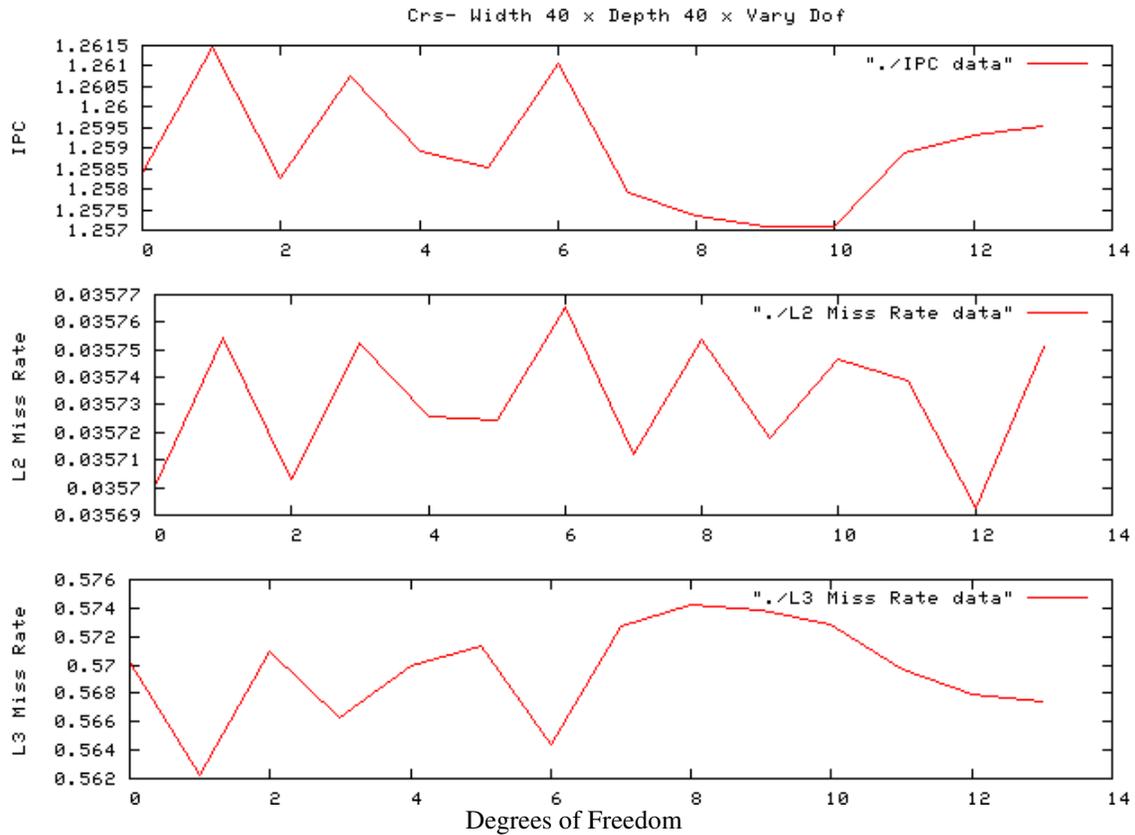


Figure 3.8: Itanium2 IPC and cache miss rates of CRS matrix W=40, D=40 with varying degrees of freedom

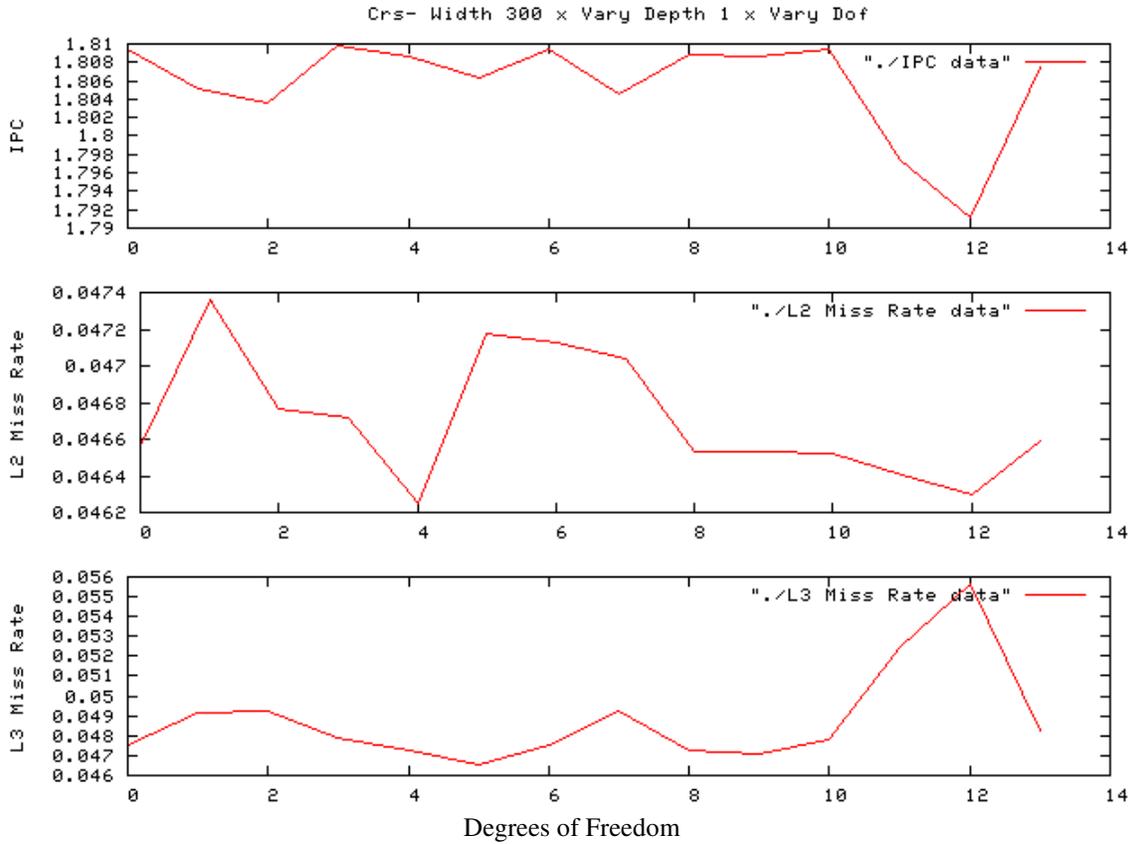


Figure 3.9: Itanium2 IPC and cache miss rates of CRS matrix W=300, D=1 with varying degrees of freedom

3.3 Performance and Bottleneck Analysis

The next step in analyzing the performance of the “cube” test problem is to identify the bottlenecks preventing maximum performance of the workload. A good place to start the investigation of these potential bottlenecks is to observe the behavior of each major structure and queue simulated under default conditions in sim-alpha. The table below shows the average performance and utilization of each major structure and queue in the sim-alpha architecture calculated for two different problem sizes (number of equations) each with 5 variations of W, D and DofPerNode (10 total simulations).

Architectural Structure	Average Statistics	Statistic Definition
L1 Data Cache	0.07079	Miss Rate
L2 Cache	0.47072	Miss Rate
Data Translation Look-aside Buffer (TLB)	0.00013	Miss Rate
Branch Predictor	0.01073	Miss Predict Rate
Integer Issue Queue	10.12167	Average Number of Instructions in Queue
Floating-point Issue Queue	1.32417	Average Number of Instructions in Queue
Load Queue	4.81939	Average Number of Instructions in Queue
Store Queue	1.93474	Average Number of Instructions in Queue
Function Unit Utilization	0.36786	Percent Utilization of Functional Units
Instructions Per Cycle	1.16264	Instructions Completed Per Cycle

Table 3.2: Average performance statistics for sim-alpha simulations

As shown in Table 3.2, the average IPC is far below the maximum IPC of 4, the theoretical maximum for this superscalar architecture with a 4 instruction fetch width. Table 3.2 also indicates 4 areas of potential architectural bottlenecks - the DL1 cache, the L2 cache, the Integer Issue queue, and high instruction-level dependencies (indicated by low functional unit utilization). Due to the high disparity between cache and memory latencies, the impact of a 47 percent L2 cache miss rate can greatly affect the overall performance. Even a 7 percent DL1 cache miss rate can impact performance because a miss to DL1 must incur the DL1 miss penalty plus the L2 hit time, assuming the cache block resides in the L2 cache. Therefore, both the DL1 and L2 caches are both potential bottlenecks. The DTLB and branch predictor rates both indicate about 99 percent accuracy or better and show no indications of hindering performance. The Integer Issue Queue is the only queue with average entries above half of the total capacity. While an average of half capacity is not alarming, since the IIQ is the most frequented of all the queues, it is also considered a potential bottleneck. Finally, the function unit utilization proves that the functional units themselves are not limiting performance, but the low

utilization suggests a potential instruction-level bottleneck. Programs that are bound by dependencies (i.e. programs containing a high percentage of instructions that depend on previous instructions to complete) often exhibit low functional unit utilization because waiting instructions can not be issued to the functional unit until the dependencies have been resolved. Therefore, the low performance of the “cube” benchmark may be caused by these instruction-level dependencies.

3.1.1 Investigating the Cache Bottleneck

Because so much research has been devoted to improving the cache performance in microprocessor, the potential cache bottleneck was the natural place to begin our bottleneck research. As described in the previous chapter, there are many proposed techniques for improving cache performance among scientific workloads represented by the cube benchmark. However, before implementing and testing the effectiveness of such methods, it is important to assess the actual impact of caching on the overall performance of the cube test problem. The maximum performance of the benchmark without cache delays is simulated by replacing the caches with a perfect cache model. In sim-alpha, the configuration file provides the option of choosing a perfect L2 cache. While sim-alpha does not simulate a truly perfect L1 data cache, nearly perfect cache performance (miss rates less than 1 percent of accesses) is achieved by making the DL1 over 64MB. The gain in IPC without cache interference is shown in Figure 3.11 below:

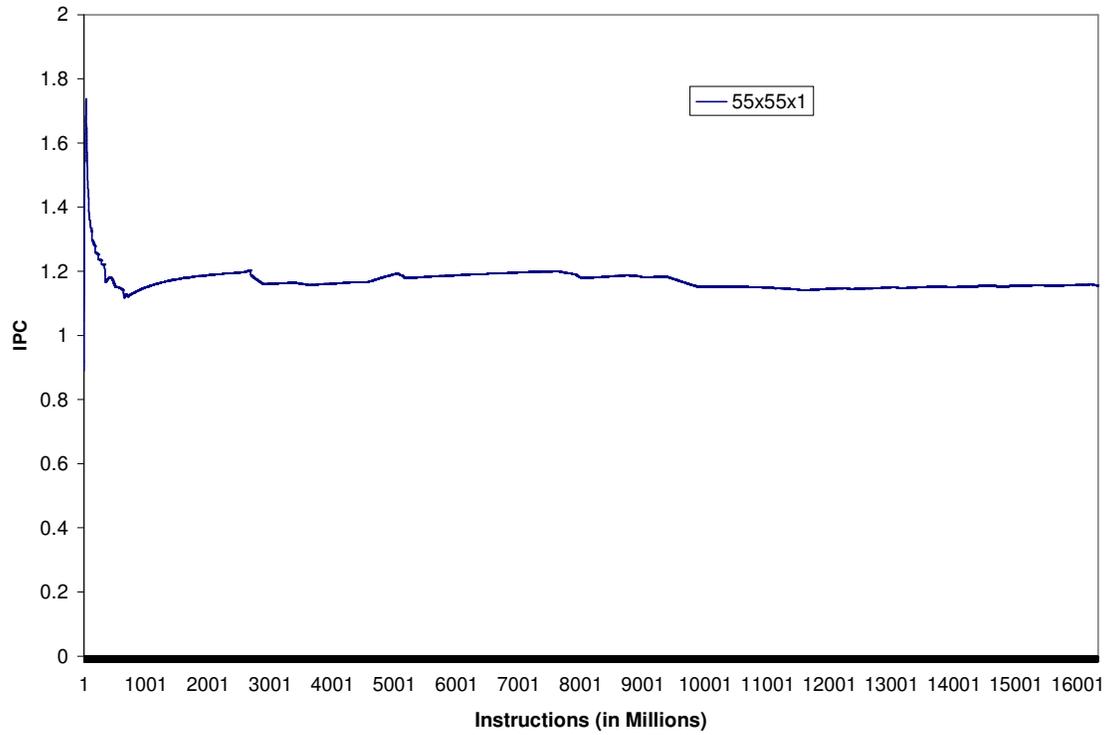


Figure 3.10: Sim-alpha IPC for Problem Size 55x55x1 with default Alpha cache configuration

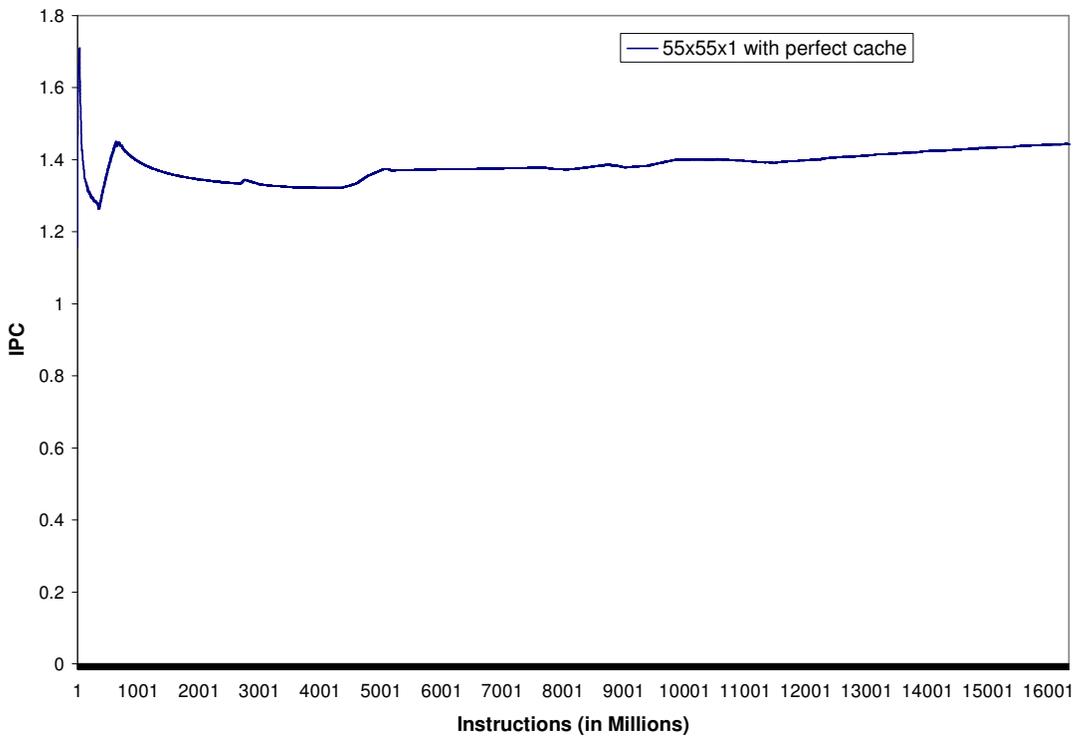


Figure 3.11: Sim-alpha IPC for Problem Size 55x55x1 with 65M DL1, infinite L2 cache configuration

Figures 3.10 and 3.11 demonstrate the overall performance improvement achieved by simulating nearly perfect cache behavior for the entire cache hierarchy. This ‘close to perfect’ cache behavior is characterized by incurring no L2 cache misses and only a 0.66 percent DL1 miss rate. The average IPC rate of the processor simulated with “infinite” cache only increases by 0.2885 instructions per cycle from the default cache configuration. The average IPC of the default configuration is 1.1548 instructions per cycle compared to 1.4433 instructions per cycle for the perfect cache configuration. The small performance gain accomplished by eliminating almost all cache misses suggests that cache behavior is not the most significant bottleneck existing for the “cube” benchmark. This conclusion is surprising due to the extensive research devoted to improving cache behavior for scientific workloads. Also, the implications of this result help shift our efforts and priorities of performance analysis from caching to other areas of potential bottlenecks.

The following Figures 3.12 and 3.13 display the performance in IPC of the SPEC00 benchmark fma3d for the default Alpha configuration and a simulated perfect cache. The fma3d benchmark displays similar performance behavior to the “cube” benchmark. The performance of both FEA benchmarks improves only slightly with “perfect” cache performance. The fma3d performance increases from 1.561 instructions per cycle to 1.8446 instructions per cycle no cache misses. While the overall performance of fma3d is minimally better than the “cube” benchmark, the 0.2836 IPC increase from the perfect cache configuration on the fma3d benchmark almost exactly matches the IPC increase from the same configuration on the “cube” benchmark.

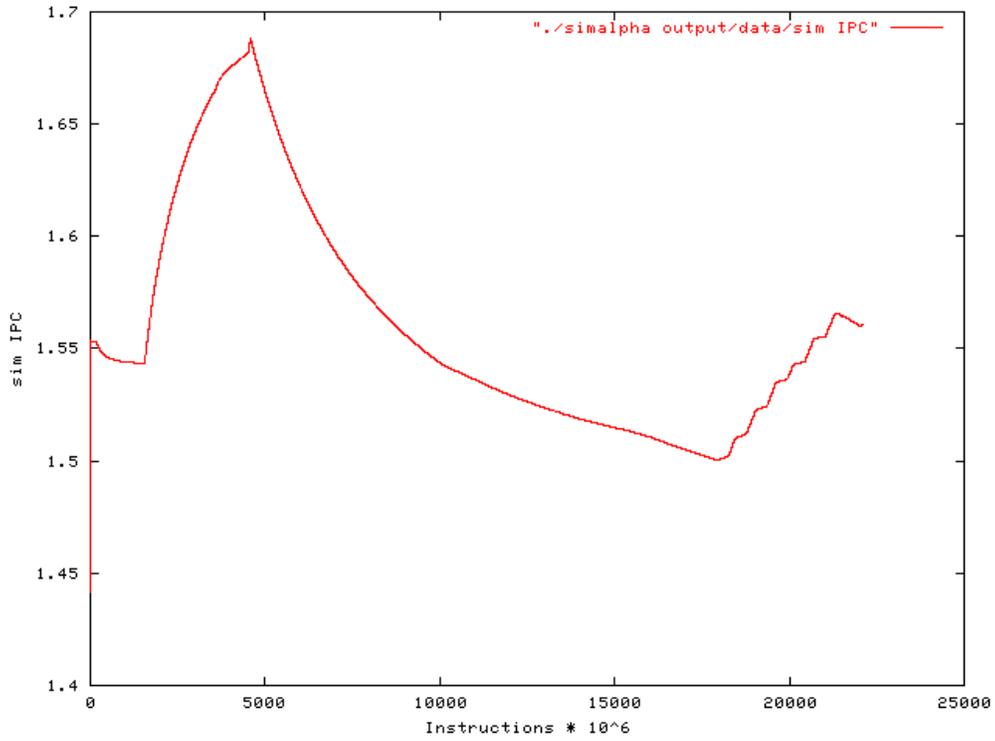


Figure 3.12: Sim-alpha IPC for SPEC00 fma3d with default Alpha cache configuration

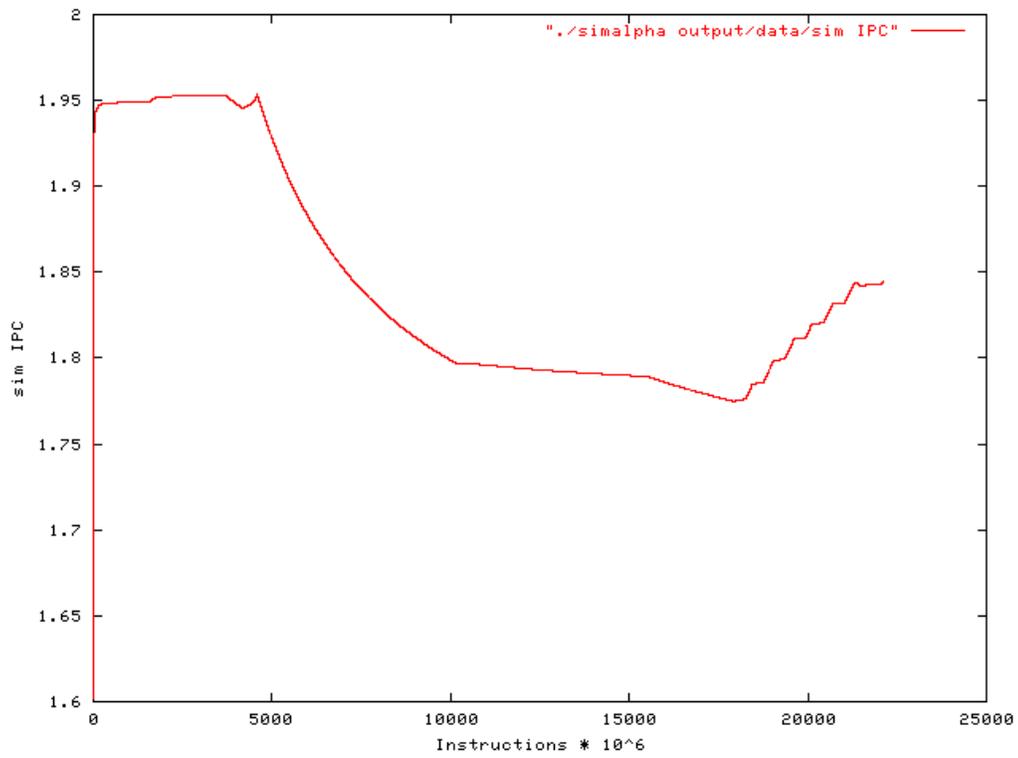


Figure 3.13: Sim-alpha IPC for SPEC00 fma3d with perfect cache configuration

3.1.2 The Integer Issue Queue and Other Architectural Bottlenecks

Similarly, to estimate the effects of the finite Integer Issue Queue, simulations are performed with the IIQ queue increased from 20-entries to 40-entries. However, this change in configuration shows no improvement on overall performance or no change in the average queue occupancy for both the “cube” and fma3d benchmarks. Finally, a ‘super’ architecture is simulated on sim-alpha and sim-outorder to estimate the maximum possible IPC of the “cube” benchmark, shown in Figure 3.3 below. This “super” configuration aims to eliminate the effects of architectural bottlenecks by providing more resources than need by the application. Although the “super” configurations are unrealistic compared to even the largest microprocessors currently available, the simulations help gauge the upper bound on performance achievable for a particular program. The following table describes the “super” configurations for sim-alpha and sim-outorder.

Configuration	Sim-outorder	Sim-alpha
Fetch Queue Size	32	8
Fetch Speed	1	1
Branch Prediction Config	2 level	Alpha
Branch Prediction Size	32 KB	32 KB
Decode Width	32	12
Issue Width	32	12
Commit Width	32	22
Reorder Buffer Size	256	160
Load/Store Queue Size	32	64
L1 Data Cache	512 KB	64 MB
L2 Data Cache	4 M	Perfect
L1 Instruction Cache	256 KB	64 MB
Instruction TLB	512 KB	8 KB
Data TLB	1 M	8 KB
Int ALU's	16	4
Int Mult/Div	8	4
Memory ports	4	2
Floating Point ALU's	16	2
Floating Point Mult/Div	8	2

Table 3.3: “Super” configuration for sim-alpha and sim-outorder

“Super” configurations are implemented in both sim-alpha and sim-outorder to demonstrate two estimates of peak performance. Because sim-alpha simulates a more complicated, realistic architecture, there are limits to its super configuration. For example, sim-alpha executes the clustering of instructions issued to the function units as performed in the ALPHA 21264. This clustering schedules the issuing of instructions to specific functions units in order to minimize producer/consumer delays. Adding functions units to the sim-alpha simulator would require a major modification of this clustering technique. The sim-alpha configuration described in Table 3.2 is the best configuration executable on our version of the sim-alpha simulator. Table 3.2 also shows that sim-outorder allows more freedom in modifying resources due to its simple architecture. A configuration similar to our sim-outorder “super” configuration has been

used in the research of other benchmarks achieving IPC values up to 10, 000 instruction per cycle for the SPEC95 ijpeg benchmark [25].

Configuration	IPC
<i>sim-alpha</i>	
Base	1.1548
Super	1.609
<i>sim-outorder</i>	
Base	1.54
Super	2.08
Super with bigger branch predictor	3.09
Super with even bigger branch predictor no cache misses No TBL misses	4.05
Super with even bigger branch predictor no cache misses no TBL misses double integer and floating point functional units	4.4
Super with even bigger branch predictor no cache misses no TBL misses double integer and floating point functional units 32 memory ports	4.73
Super with even bigger branch predictor no cache misses no TBL misses quadruple integer and double floating point functional units 32 memory ports 1 cycle functional unit latencies	4.98

Table 3.4: IPC achieved with sim-alpha and sim-outorder ‘super’ configurations for CRS matrix W=55, D=55, Dof=1

The sim-alpha “super” configuration showed modest performance gains due to limited configuration flexibility. The best sim-outorder configuration achieves an IPC rate of almost 5 instructions per cycle. The over 2x speedup of this configuration with almost no architectural constraints estimates the maximum performance achieved by eliminating the bottlenecks. The SPEC00 benchmark executing finite-element analysis,

fma3d, also achieves limited improvement with the “super” configuration, improving only 1.8 instructions per cycle from its base IPC of 1.561.

The “super” configuration does enable both the “cube” benchmark and fma3d to achieve better overall performance, but performance is still far below ideal. After quantifying the effects of potential architectural bottlenecks, it is important to understand the bottlenecks within the benchmark itself. As noted before in Table 3.2, the functional units appear to be under-utilized indicating a possible instruction-level dependency bottleneck. Quantifying the effect of this instruction-level bottleneck is an essential part in understanding the performance of the “cube” benchmark.

3.1.3 Instruction-level Dependency Bottleneck

The degree of instruction-level dependency existing in the cube benchmark is calculated by measuring the number of cycles each instruction spends waiting on a previous instruction to complete. Out-of-order execution, used in sim-alpha, sim-outorder, and the Itanium2 processor, allows instructions to be executed by the functional units out of program order. This type of execution works to eliminate wasted cycles by keeping the functional units busy and executing independent instructions (instructions that don't depend on the execution of previous instructions) during idle cycles. However, to preserve correct program execution, instructions must be committed (saved) in program order. Therefore, in programs with high levels of instruction-level dependency, instructions may spend several cycles in the reorder buffer waiting for a previous instruction commit.

During simulation, tracking the number of cycles each instruction spends waiting to be issued to the functional units and to commit in the reorder buffer quantifies the existence of instruction-level dependency during each phase of the cube benchmark, shown in Figure 3.14. The cycles are also grouped by instruction-type to identify which type of instructions spend the most time waiting at the top of the reorder buffer to commit. These instruction types are responsible for holding up subsequent instructions and are the main cause of this instruction-level bottleneck. Figure 3.15 displays those instructions causing the most waiting time in the reorder buffer.

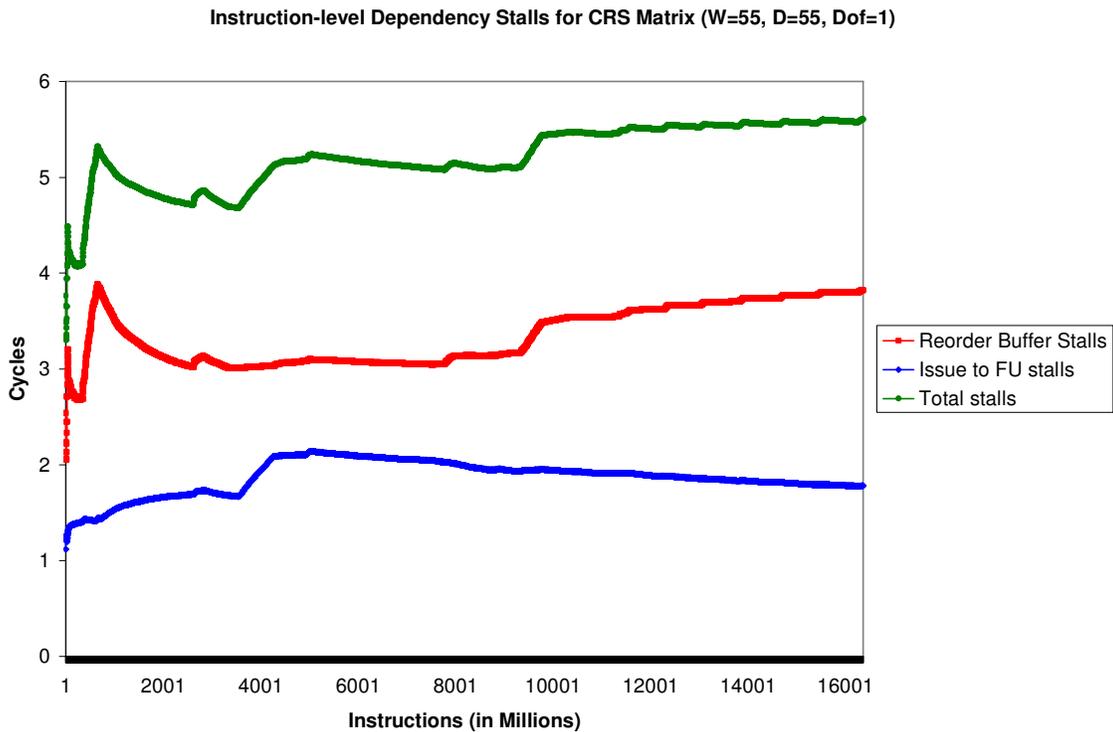


Figure 3.14: Average number of cycles each instruction waits to issue and commit for sim-alpha

The graph above shows the overall average wait for each instruction to issue to the functional units and to commit from the reorder buffer is approximately 5.58 cycles. The average graph peaks during an initial and also jumps up and continues to increase during the solve phase starting around 10 billion instructions. The average wait for each

instruction to issue to the functional units is approximately 1.78 cycles. The average wait for each instruction in the reorder buffer is 3.8 cycles. Because of the low functional unit utilization, the average wait to issue can be attributed to instruction-level dependencies. Only instructions with all operands available can issue to the functional units. Therefore, if an instruction is depending on the previous instruction for an operand, the instruction will wait to be issued until the previous instruction has produced the needed operand. This is another example of an instruction-level dependency stall.

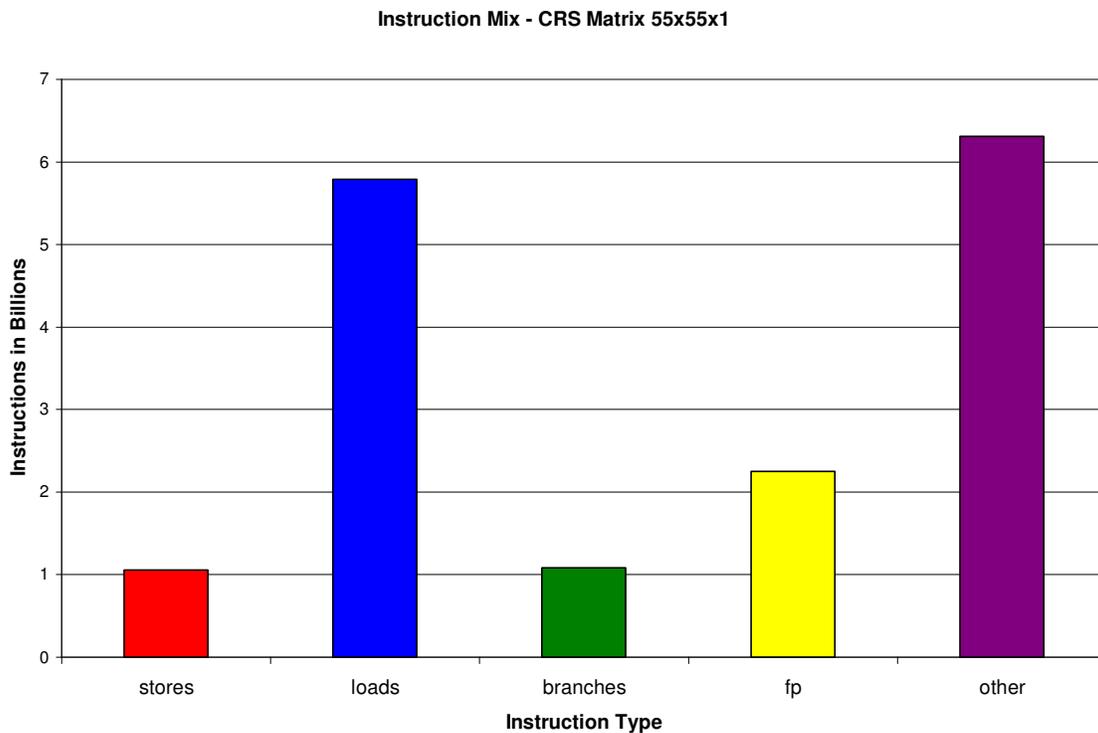


Figure 3.15: Number of each instruction type executed in the “cube” benchmark (Instruction Mix)

Percentage of Stall Cycles Caused by Each Instruction Type (CRS Matrix 55x55x1)

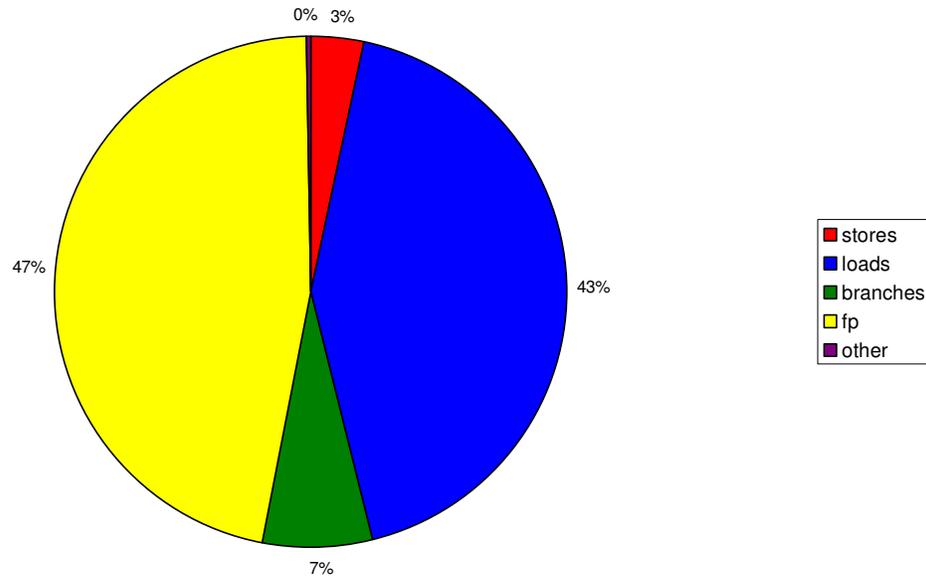


Figure 3.16: Reorder buffer stalls caused by instruction type in sim-alpha (55x55x1)

Figure 3.15 categorizes the number of each instruction that is executed by the “cube” benchmark. The major of the instructions executed fall into the load or other category. However, Figure 3.16 indicates that the most of the stalling in reorder buffer is caused by floating point instructions and load instructions. These results are consistent with the operations performed in the main kernel of the solve phase performed by the cube benchmark. This kernel, found in the Trillinos Aztec solver library (azgmres.c), performs sparse matrix multiplication. Consider the simple sparse matrix multiplication, $y = A * x$. This operation requires the loading of each value of the matrix A (the cube mesh in the cube benchmark) and the vector x (values are reused). Then, each element of A is multiplied the appropriate x value and saved in the vector y. Each matrix multiply is a floating point operation which account for most of the stall cycles in the reorder buffer.

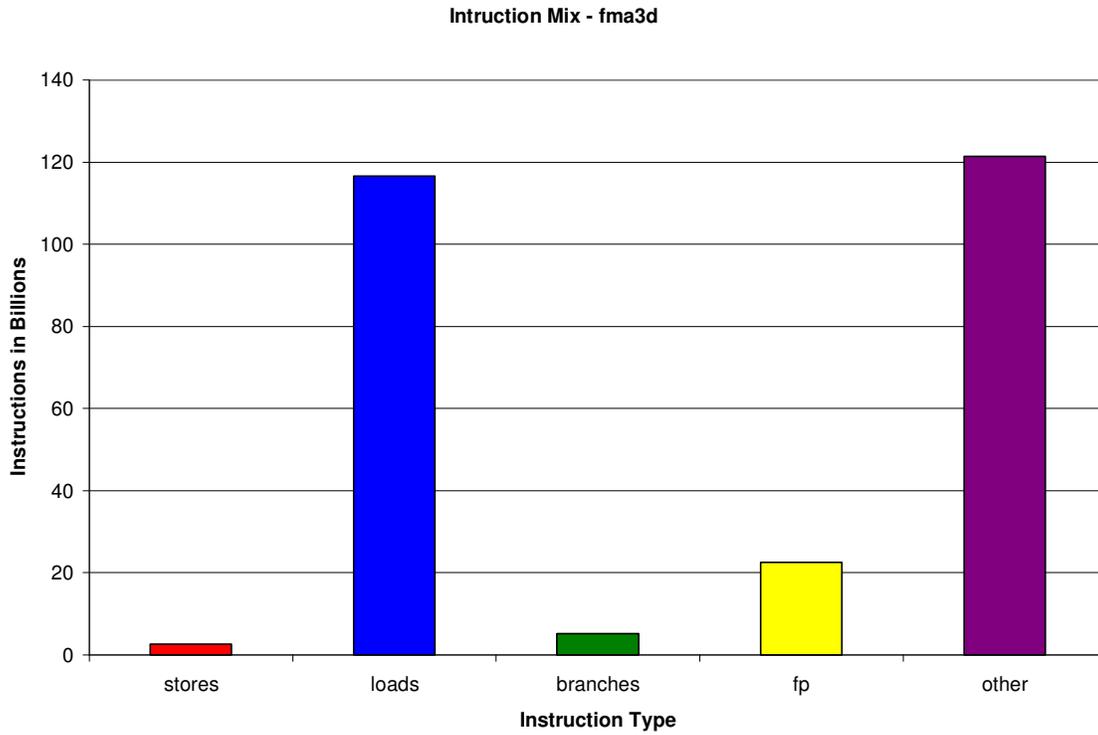


Figure 3.17: Number of each instruction type executed in the fma3d benchmark (Instruction Mix)

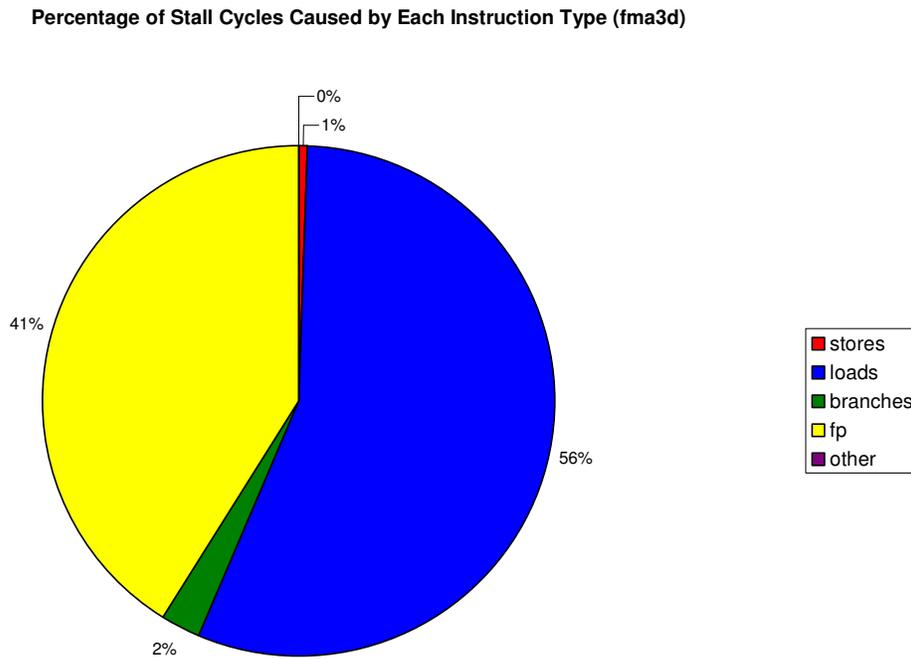


Figure 3.18: Fma3d reorder buffer stalls caused by instruction type

Figure 3.17, categorizing the instructions causing stall cycles in the SPEC00 benchmark fma3d, shows results very similar to the “cube” benchmark. Floating point and load instruction still cause the major of the stall in the reorder buffer, but in the fma3d benchmark the loads account for more stall than the floating point instructions (Figure 3.18). These results identifying floating point and load instructions as the major contributors to instruction-level stalls for the “cube” and fma3d benchmarks are verified by statistics gathered by the Itanium2 performance counters.

Figure 3.19 presents the instruction mix for the “cube” benchmark collected by the Itanium2 processor. The instruction mix is similar to the “cube” instruction mix from sim-alpha, however the more floating point operations are performed by the sim-alpha simulator and more branches are executed by the Itanium2 processor. These differences occur due to the differences in the hardware, compilers and instruction set architectures (ISA). The ISA describes the aspects of the computer architecture visible to a programmer, including the instruction and data types, addressing modes, memory architecture, exception handling and others. Figure 3.20 identifies the major contributors to stalling in the Itanium2 processor. And even with the differences in the two processor types and ISAs, the figure confirms that loads and floating point operations are still the major operations causing stall in the execution of the “cube” benchmark.

CRS Instruction Mix

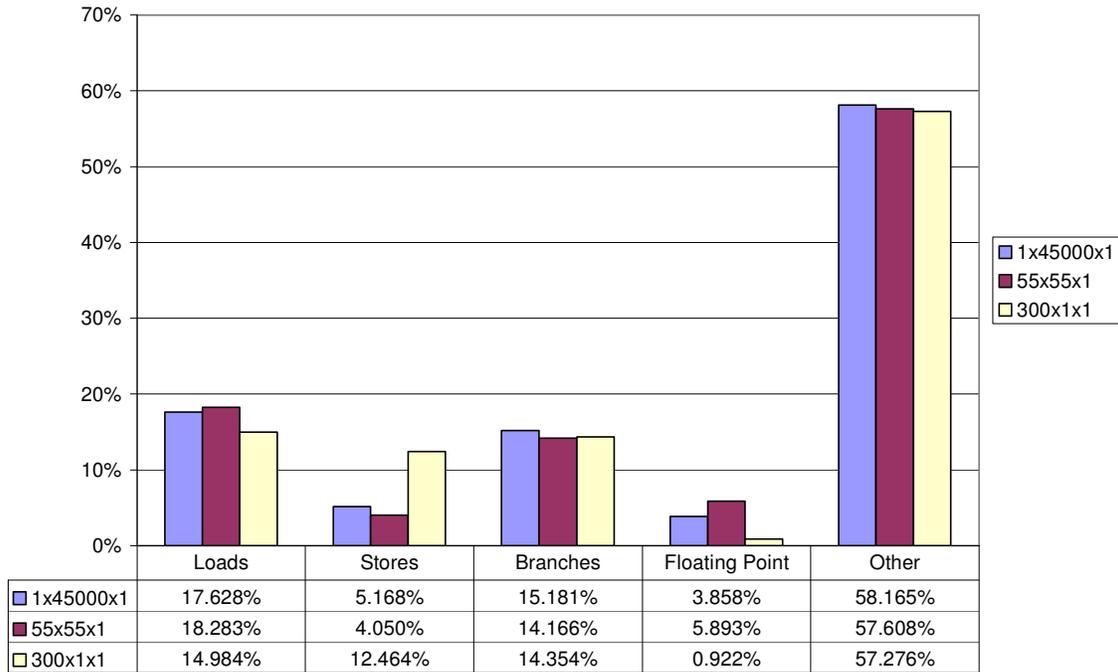


Figure 3.19: Number of each instruction type executed in the “cube” benchmark on the Itanium2 processor for 3 equal equation problems (Instruction Mix)

Stall Percentage CRS

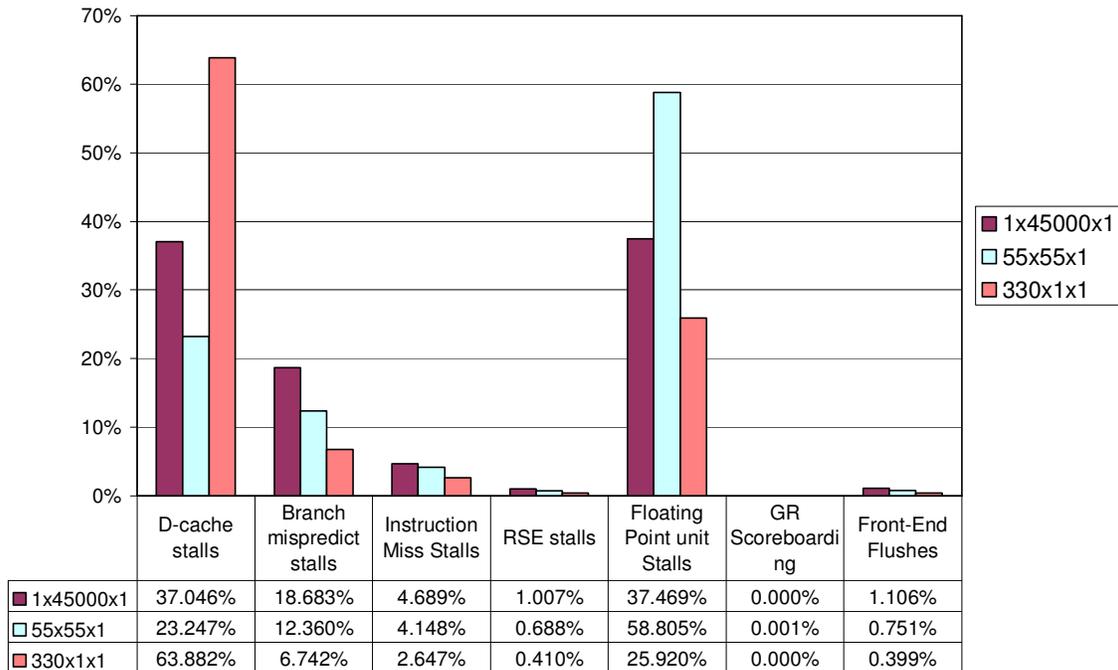


Figure 3.20: “Cube” benchmark dependency stalls collect from Itanium2 processor for 3 equal equation problems

3.1.4 Reducing Instruction-level Dependency

Instruction-level dependency chains cause delays in the execution and processing of instructions and therefore stifle performance. Many times, this instruction-level dependency arises in scientific computing due to the tight loop computation method used to solve scientific problems. Recurrences are computations or microprocessor instructions whose value for one iteration depends directly previous iterations. For example, the equation

$$\text{sum} = \text{sum} + x$$

demonstrates recurrence. Loop unrolling is a technique used to overcome the dependency chains caused by recurrences and other dependencies. Loop unrolling consists of unrolling (moving outside the loop) several iterations of a loop to make available more independent instructions. Consider the loop unrolling technique as applied to the inner loop of matrix multiplication used in each of the solvers available in the “cube” benchmark.

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

In this loop, NumMyRows equals the number of rows in the cube. NumEntries is the number of nonzero entries per row of the cube. And, the core multiplication involves an element from the cube (RowValues[j]) and the corresponding element from the multiplication vector (xp[RowIndices[j]]). The following example shows the technique of loop unrolling applied to the inner matrix multiplication loop of the “cube” benchmark.

Core Loop

```
load RowValues[j]
load RowIndices[j]
mult RV[j]      RI[j]
add  sum        (RV[j] + RI[j])
```

Example Latencies of Operations	
Instructions	Clock cycles
FP op to FP op	3
FP op to Store op	2
Load op to FP op	1
Load op to Store op	0

Normal execution

(4 loop iterations with stalls)

```
1 load RV[j]
2 load RI[j]
3 stall
4 mult RV[j]      RI[j]
5 stall
6 stall
7 add  sum        (RV[j] + RI[j])
8 load RV[j+1]
9 load RI[j+1]
10 stall
11 mult RV[j+1]   RI[j+1]
12 stall
13 stall
14 add  sum        (RV[j+1] + RI[j+1])
15 load RV[j+2]
16 load RI[j+2]
17 stall
18 mult RV[j+2]   RI[j+2]
19 stall
20 stall
21 add  sum        (RV[j+2] + RI[j+2])
22 load RV[j+3]
23 load RI[j+3]
24 stall
25 mult RV[j+3]   RI[j+3]
26 stall
27 stall
28 add  sum        (RV[j+3] + RI[j+3])
```

Loop Unrolling Technique

(4 loop iterations with stalls)

```
1 load RV[j]
2 load RI[j]
3 load RV[j+1]
4 load RI[j+1]
5 load RV[j+2]
6 load RI[j+2]
7 load RV[j+3]
8 load RI[j+3]
9 mult RV[j]      RI[j]
10 mult RV[j+1]   RI[j+1]
11 mult RV[j+2]   RI[j+2]
12 mult RV[j+3]   RI[j+3]
13 add  sum        (RV[j] + RI[j])
14 add  sum        (RV[j+1] + RI[j+1])
15 add  sum        (RV[j+2] + RI[j+2])
16 add  sum        (RV[j+3] + RI[j+3])
```

Figure 3.21: Loop unrolling technique applied to “cube” inner loop

Figure 3.21 demonstrates how the loop unrolling technique in this case reduces the execution by 12 cycles. Compilers play a major role in decreasing dependency chains through scheduling and loop unrolling. Generic compilers perform some loop optimizations, but workload-specific loop optimizations could help to eliminate stalling caused by instruction-level dependencies. Workload-specific optimizations would allow the compiler to perform the exact amount of loop unrolling need to best improve the performance of that particular workload.

Another compiler technique used to reduce stalls caused by instruction-level dependencies is called software pipelining or symbolic loop unrolling. Software pipelining overcomes dependencies by including instructions from different iterations in the main loop, making them independent of other instructions in the loop. Figure 3.22 below shows the software pipeline technique applied to the inner loop matrix multiplication of the “cube” benchmark. The software pipelining technique includes a section of startup code that must be executed before the pipelined inner loop and a section of cleanup code to be complete after the inner loop. The inner loop includes four independent instructions that can be executed simultaneously with any dependency stalls.

In this case, the software pipelining technique reduces the execution cycles from the 63 cycles of the original loop to the 36 cycles of the software pipelined loop, 1.7 times faster. This technique of software pipelining is already performed in some compiler optimizations, however our executables were not compiled with an optimization including this technique. Future work will include recompiling with a compiler optimization employing software pipelining and simulating the performance enhancement resulting. Also, we plan to use these compiler optimizations in conjunction with cache enhancement techniques in an attempt to maximize performance.

Core Loop

```
load1  RowValues[j]
load2  RowIndices[j]
mult   RV[j]    RI[j]
add    sum     (RV[j] + RI[j])
```

Example Latencies of Operations

Instructions	Clock cycles
FP op to FP op	3
FP op to Store op	2
Load op to FP op	1
Load op to Store op	0

Software Pipelining
(9iterations)

startup code:

```
load  RowValues[j]
load  RowIndices[j]
load  RowValues[j+1]
mult  RV[j]    RI[j]
load  RowIndices[j+1]
load  RowValues[j+2]
```

inner loop:

```
for(i=0; i<NumMyRows_,
i++)
    for(j=0; j<NumEntries, j++)
        load  RV[j+3]
        load  RI[j+2]
        add   sum     (RV[j]+RI[j])
        mult  RV[j+1] RI[j+1]
```

cleanup code:

```
load  RI[j+8]
mult  RV[j+7] RI[j+7]
mult  RV[j+8] RI[j+8]
add   sum     (RV[j+6]+RI[j+6])
add   sum     (RV[j+7]+RI[j+7])
add   sum     (RV[j+8]+RI[j+8])
```

j	j+1	j+2	j+3	j+4	j+5	j+6	j+7	j+8
load1								
load2								
mult								
add								

Figure 3.22: Software pipelining technique applied to the “cube” inner loop

3.4 Performance Results and Conclusions

The results from our analysis of the “cube” benchmark and the potential bottlenecks hindering its performance proved successful in eliminating the caches and other architectural structures as the major bottlenecks to the “cube” performance. However, the instruction-level dependency existing in the “cube” benchmark does prevail as the most significant bottleneck to the microprocessor performance of this scientific workload. In future work, instruction-level modifications or compiler optimizations such as workload-specific loop unrolling techniques are potential solutions to overcoming this instruction-level dependency bottleneck.

Moreover, the cache behavior in terms of miss rates was not considered a major bottleneck to performance, however determining the best performing cache configuration for the “cube” benchmark and other similar scientific workloads is an important consideration for microprocessor designers. The perfect cache simulated in this chapter is not a viable option for microprocessor design; therefore the next chapter will explore the best cache configuration for workloads represent by the “cube” test problem.

4 IMPROVING CACHE PERFORMANCE

Although the results from the previous chapter indicated that cache behavior was not the most significant performance bottleneck plaguing the finite element analysis of sparse matrices, choosing the best cache configuration for these scientific computations remains an important design decision. Previous simulations demonstrated the maximum performance achievable with a simulated ‘infinite’ cache. And because “infinite” cache does not exist, comparing the performance of realistic cache configurations and their effect on overall performance must be determined.

Most cache performance studies pertain to general purpose programs; however two studies described in the following paragraphs investigate cache performance of sparse matrix computations with an emphasis on 3-D finite element analysis. In [26], cache performance in terms of cache hit rates are examined for the parameters of cache size, associativity, block size, write policy, one-cycle write operation, and the number of read and write ports. Caches sizes are varied from 512 B to 128 KB. The associativity is varied from direct-mapped to 4-way set-associative, and block sizes range from 8 to 128 bytes. Simulations of the sparse matrix multiplication performed with iterative linear solvers, on preconditioned matrices stored according to a compact storage format, are evaluated on the Thor simulator. Thor is an event-driven functional simulator developed at Stanford and based on the CSIM simulator out of the University of Colorado at Boulder. The simulation results are based on the average hit rates for 8 input mesh sizes, including 2 real finite element problems acquired from Lockheed Palo Alto Research Laboratory. The summary of these simulations suggest the following cache configuration for sparse matrix computations as shown in Table 4.1:

Organization	Direct-mapped
Size	1-8 KB
Write Policy	Write-back, 1 cycle
Block Size	128 B
Read Ports	1
Write Ports	1

Table 4.1: Recommended Cache Configuration for Sparse Matrix Computations [26]

Another study of the effects of sparse matrix multiplication on caches by [27] analytically examines the cache access behavior of these workloads. The analysis assumes multiplication is performed on the sparse matrix (M) arranged in storage-by-row format, its description index vectors (I and D), the multiplication vector (X), and the resulting vector (Y). The vector I stores the non-zero column positions and D stores the non-zero row positions.

Sparse Matrix Multiplication Format: $Y = M * X$

This research focuses primarily on the cache access behavior of the matrix M and vectors I and X because vectors Y and D exhibit high spatial and temporal locality and account for a small percentage of cache misses. The column vector, I, is accessed many more times than the row vector, D, because matrix multiplication is performed by row. Row multiplication requires only one access to the row vector, D, and many accesses to the column vector, I. Matrix M and vector I demonstrate no temporal locality because each element is only accessed one time. Because of their size compared to the other vectors, both M and I may cause cache pollution hindering the exploitation of the locality found in the other vectors. Vector X shows the most potential for the exploitation of additional

temporal locality depending on the size of the vector, the impact of interference from the matrix M , and the parameters of the cache configuration.

Detailed analytical equations and simulations help quantify the effects of cache parameters and organization on the performance of caches during the computation of sparse matrix multiplication. These analytic calculations prove that when the bandwidth of matrix M is smaller than the cache size, the cache hit rate improves due to greater exploitation of temporal and spatial localities. The matrix bandwidth, W_B , is defined as the width in columns of the non-zero diagonal terms of the sparse matrix, M .

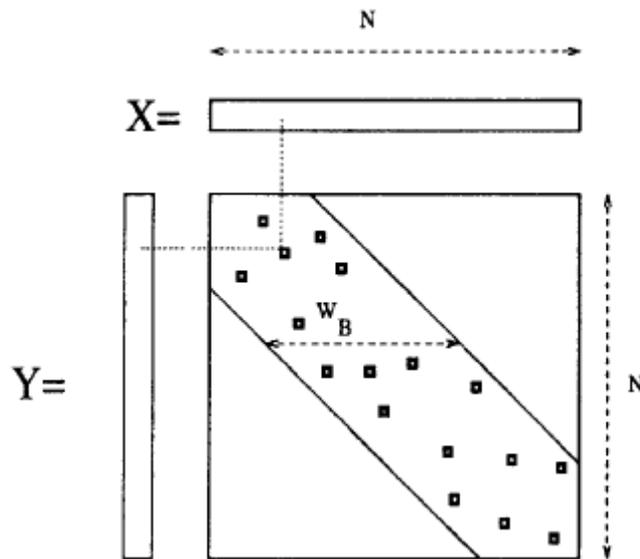


Figure 4.1: Bandwidth, W_B , of sparse matrix M from [27]

Also, increasing the line (block) size of the cache improves the overall cache hit rate for line sizes up to 64 bytes (maximum size for this study). However, the vector X shows a maximum hit rate at a line size of 8 bytes and then decreases along the parabolic curve for larger line sizes. The matrices with higher densities (less average distance between non-zero elements) show greater hit rates for each cache configuration considered benefiting from the temporal and spatial localities of X .

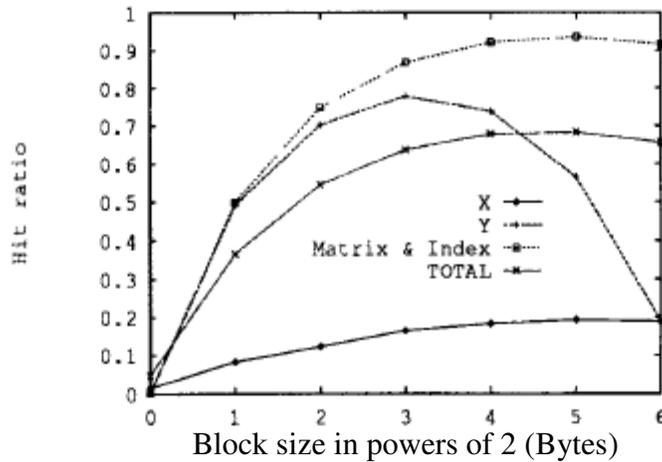


Figure 4.2: Cache hit ratio for cache configurations with block sizes from 0 to 64B taken from [27]

Finally, this research suggests that the blocking-by-diagonal formatting technique (rather than storage-by-row) provides better reuse of the vector X and a better parallelization platform for multi-processor computing. Blocking-by-diagonal involves splitting the original banded sparse matrix on the diagonal resulting in smaller submatrices each with smaller bandwidths. The smaller bandwidths of these submatrices decrease the burden on each local cache and improve local cache hit rates.

4.1 Cache Configuration Effects on Performance

In order to compare the cache behavior of the cube3 benchmark to the sparse matrix computation results summarized above, the DL1 and L2 cache miss rates for various cache configurations are presented. The miss rates for this study are collected from a CRS, pre-conditioned mesh with dimensions of width 55, height 55 and depth 55 and one degree of freedom per node. With this constant problem size, the DL1 and L2 cache sizes, associativities, and block sizes are varied.

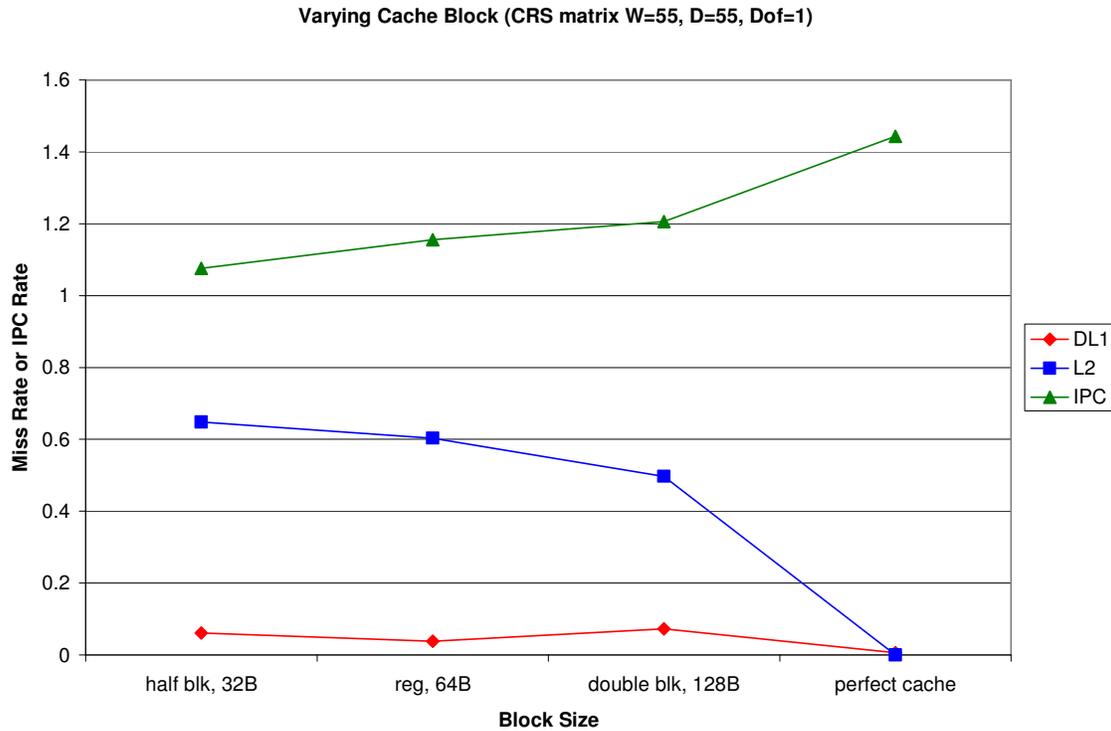


Figure 4.3: Cache miss rates and IPC rates for caches with varying cache block size

Figure 4.3 shows the cache miss rates and the IPC rates for the default sim-alpha cache sizes with varying block sizes. The block size, 128B, appears to be the best performing block size configuration as indicated by the highest IPC rate and the lowest L2 cache miss rate. The DL1 cache miss rate for 128B block size is slightly higher than the other block sizes, but not significantly. The 64B block size may be more appropriate for the smaller DL1 cache, but overall the best performance is achieved by the configuration with 128B block size.

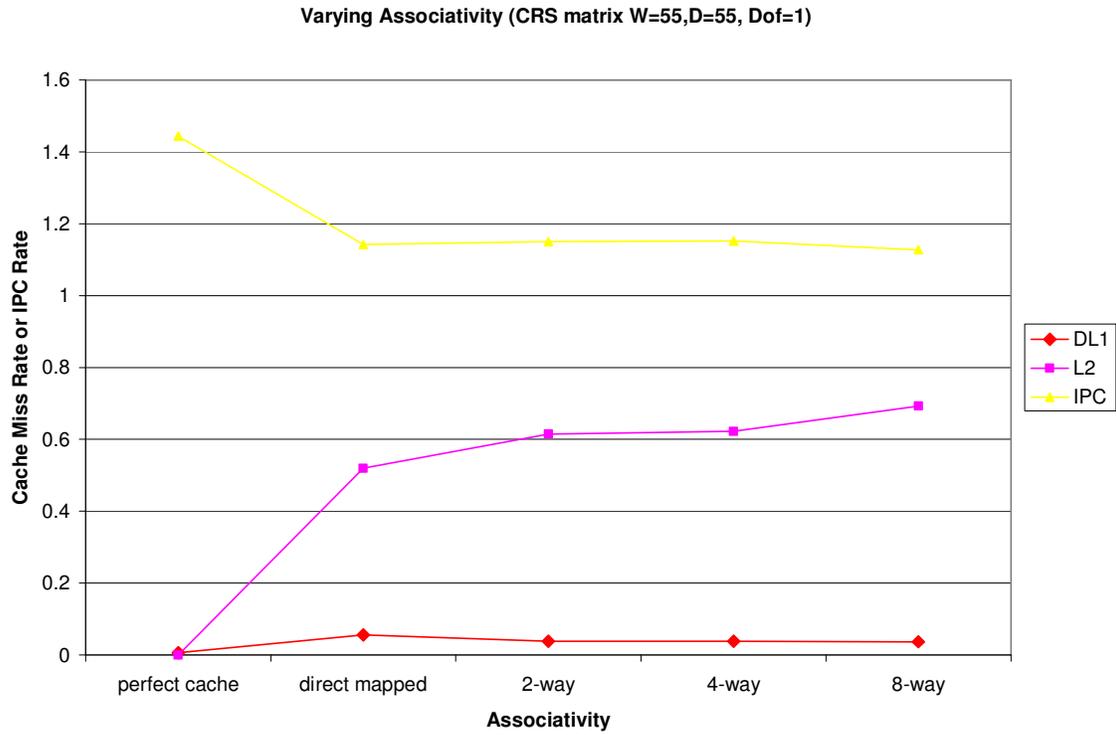


Figure 4.4: Cache miss rates and IPC rates for caches with varying associativity

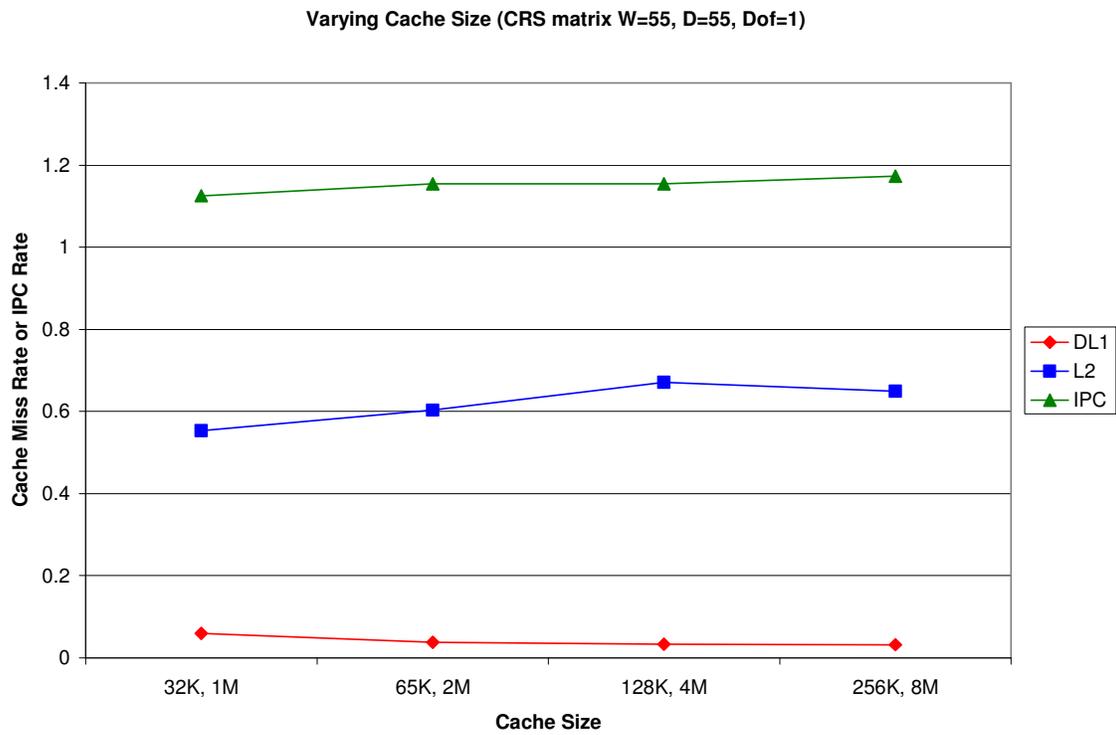


Figure 4.5: Cache miss rates and IPC rates for varying cache sizes

Associativity seems to have little impact on performance in terms of IPC as demonstrated in Figure 4.4, but the L2 cache incurs the least misses with the direct-mapped configuration and the DL1 with a 4-way set-associative cache. A 2-way set-associative DL1 and direct mapped L2 cache are the cache configurations used in the Alpha 21264 microprocessor. The DL1 cache size of 256KB and L2 cache size of 8MB together demonstrate the best performance in terms of IPC and the DL1 miss rate of all the four cache sizes simulated, according to Figure 4.5. The results of the cache configuration performance study are summarized in Table 4.2, with the best rate for each structure bolded and the best overall configuration option highlighted.

Cache	DL1	L2	IPC
Block Size			
half blk, 32B	0.0619	0.6481	1.0765
reg, 64B	0.0382	0.6034	1.1548
double blk, 128B	0.0731	0.4972	1.2054
perfect cache	0.0066	0	1.4433
Associativity			
direct mapped	0.0554	0.519	1.1421
2-way	0.0382	0.614	1.1502
4-way	0.0378	0.622	1.1521
8-way	0.0373	0.6926	1.1274
Cache Size			
32K, 1M	0.0599	0.5531	1.1249
65K, 2M	0.0382	0.6034	1.1548
128K, 4M	0.0334	0.6705	1.1547
256K, 8M	0.032	0.6492	1.1728

Table 4.2: Simulated cache miss rates and IPC rates for various cache configurations

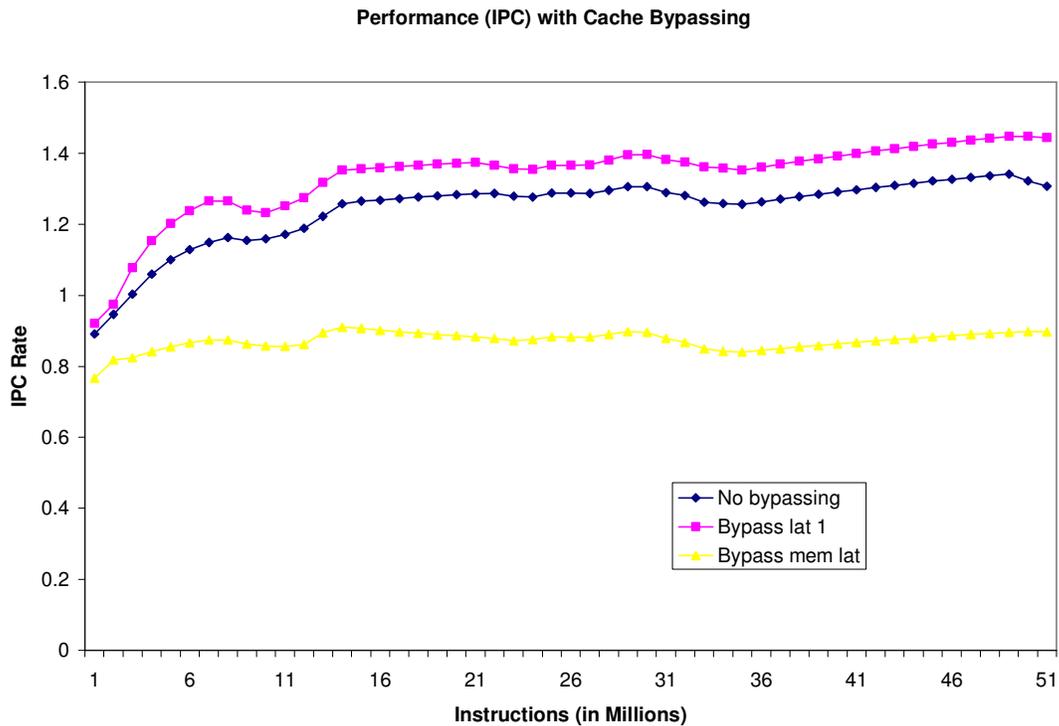
4.2 Cache Bypassing

Another method of cache optimization described in Chapter 2 involves strategically excluding large matrices and arrays (that are only accessed one time during the solve phase of finite-element analysis and sparse matrix multiplication) from the cache. All instructions involving these array elements would operate directly to and from main memory. This bypassing of the cache prevents cache pollution caused by the elements of large arrays that are never reused in the execution of the program. Some methods described in Chapter 2 dynamically determine which memory accesses to cache or not, and others create a separate cache exclusively for these array elements.

To quantify the effectiveness of this bypassing cache optimization, the memory addresses of the matrix describing the problem of the cube3 benchmark are statically marked for exclusion from both levels of cache. To determine the exact memory location of the matrix in cube3 test problem, an address print statement is placed after the storage optimization, just before the solve phase in the `Epetra_CrsMatrix.cpp` file of the Trilinos solver package.

```
double * tmp = All_Values_;
for (i=0; i<NumMyRows_; i++) {
    int NumEntries = NumEntriesPerRow_[i];
    for (j=0; j<NumEntries; j++){
        tmp[j] = Values_[i][j];
        /* INSERTED PRINT STATEMENT */
        printf("%p\n",&Values_[i][j]);
    }
    if (Values_[i] !=0)
        delete [] Values_[i];
    Values_[i] = tmp;
    tmp += NumEntries;
}
```

These addresses are hard-coded into the simulator at the point of the cache access, in files `writeback.c` and `commit.c` of `sim-alpha`. When one of these addresses is requested by an instruction, the bypass tells the simulator to go directly to memory for the data and prevents the data from being stored in the cache. The bypass counter tracks the number of memory accesses bypassed by the caches. The final counter result should equal the number of matrix elements bypassed times the number of iterations of the solver because we are bypassing the matrix elements that are only accessed one time per iteration.



Figures 4.6: Cache bypassing performance results measured by IPC

Figure 4.6 shows the effect of the cache bypassing on IPC for a small $8 \times 8 \times 3$ “cube” problem. The first line plots the IPC of the original $8 \times 8 \times 3$ problem with no cache bypassing. The next line shows the effect of the bypassing with an unrealistically low memory latency of one cycle. This low memory latency is plotted to show the effects of

the bypassing. The figure shows that the bypassing does affect the performance in terms of IPC, in this case positively because each access to the bypassed addresses costs only one cycle (instead of the 12 cycles actually needed to access memory). The third line implements the actual memory latency of 12 cycles, and the graph indicates the decrease in IPC caused by this change. While this line represents the most realistic implementation of the cache bypassing technique, other factors may be causing IPC to decrease in this case.

First, the problem size is relatively small and the matrix we are bypassing is actually a manageable size to the Alpha cache configuration. Therefore, the execution with no bypassing benefits from cache matrix element from one iteration to the next. However, when scaled to bigger problems, the matrix elements would be forced to reside in memory with or without cache bypassing, and the bypassing would not decrease performance, but only allow for the reuse of other, smallest elements in the cache. Also, in this initial test of the cache bypassing technique, the addresses are excluded from the cache in the entire execution of the “cube” benchmark. However, a more accurate implementation would only exclude the addresses during the solve phase of execution because initial matrix formation and conditioning may benefit from caching. Also, we obtained the addresses from the “cube” benchmark after the initial phases, just before the solve phase, so to exclude the addresses before the solve phase may cause erroneous addresses to be excluded. The second phase of the cache bypassing implementation that includes (a) scaling the method to larger problem sizes and (b) bypassing addresses only during the solve phase was postponed in light of the perfect cache results indicating that cache behavior was not the most significant performance bottleneck.

Furthermore, the approach of statically defining the memory addresses of the large matrix to exclude from the cache is a simple way to evaluate the effectiveness of this technique of cache optimization. However, static definitions are not a practical implementation for a variety of workloads and processor platforms. Future work on this technique includes adding attributes to the source code of the cube3 benchmark to dynamically signal which memory accesses should be placed in the cache and which should be stored strictly in main memory. This dynamic implementation provides a more universal solution for optimizing finite-element and sparse matrix multiplication solvers.

5 CONCLUSION

Designing the best performing microprocessor for a class of applications involves researching the impact of each major design decision and exploring innovative methods to best solve the application specific challenges. The class of large scientific applications executed at Sandia National Laboratories present unique characteristics and challenges for microprocessor performance optimization. This thesis investigates the microarchitectural bottlenecks limiting the performance of SNL's large scientific applications and proposes configurations and techniques to improve performance.

The cache configuration has traditionally been considered one of the major bottlenecks to scientific workload performance, and therefore much research has been devoted to cache improvement techniques. Our simulations of the "cube" test problem, (W=55, D=55, Dof=1) used to represent scientific applications used at SNL, suggest the following cache hierarchy for the best overall benchmark performance:

- DL1 - 256KB, 2-way set-associative, 64B block size
- L2 – 8MB, direct-mapped, 128B block size

However, our simulations also indicate that cache behavior is not the main bottleneck limiting performance.

A technique of cache bypassing that prevents matrix cache blocks and other one-time use cache blocks from entering the cache hierarchy was simulated to determine the performance gains of reducing polluting the cache. The cache bypassing technique implemented for this research did not improve performance as expected. Because the cache hierarchy was not the most significant performance barrier, improvements to the

technique were not explored. However, the cache bypass technique will be revisited in future work.

After ruling out the cache as the major performance bottleneck, other micro-architectural bottlenecks were explored. Using configurations for the SimpleScalar simulators, sim-alpha and sim-outorder, to simulate perfect cache behavior, we observed a performance increase of only 0.28 instructions per cycle over the IPC of the same problem with the default Alpha cache configuration. Simulating a perfect cache serves to eliminate all stalling caused by cache misses. Since removing all cache stalls does not significantly improve the overall performance of the benchmark, the cache configuration does not appear to be the main factor limiting performance.

A “super” microprocessor configuration with immense resources was simulated to remove all architectural restraints on microprocessor performance. The 3.44 IPC achieved with the best configuration shows that additional microprocessor resources do help performance, but not as much as expected. Since other types of benchmarks achieve IPC rates of hundreds, even tens of thousands of instructions per cycle with the “super” configuration, the performance reduction from instruction-level dependency stalling was next potential bottleneck explored.

Our simulation of instruction-level dependency stalls in the “cube” benchmark for the default Alpha configuration reveals that on average an instruction waits 1.78 cycles to be issued to a functional unit and waits 3.8 cycles in the reorder buffer before being committed. This average 5.58 cycle stall per instruction due to dependencies constitutes a significant performance bottleneck. The average stall cycle increases to over 7 cycles per instruction for the “cube” benchmark simulated with the “super” architectural configuration. Classifying each stall by instruction type indicates that most stalling

occurs from floating-point and load instructions. The result seems reasonable since one of the main kernels in the “cube” benchmark to perform sparse matrix multiplication involves these instructions. Workload-specific loop unrolling and other compiler techniques are possible solutions to reducing the stalling caused by instruction-level dependencies. Our future work includes researching and implementing these techniques to reduce instruction-level dependencies and improve overall performance.

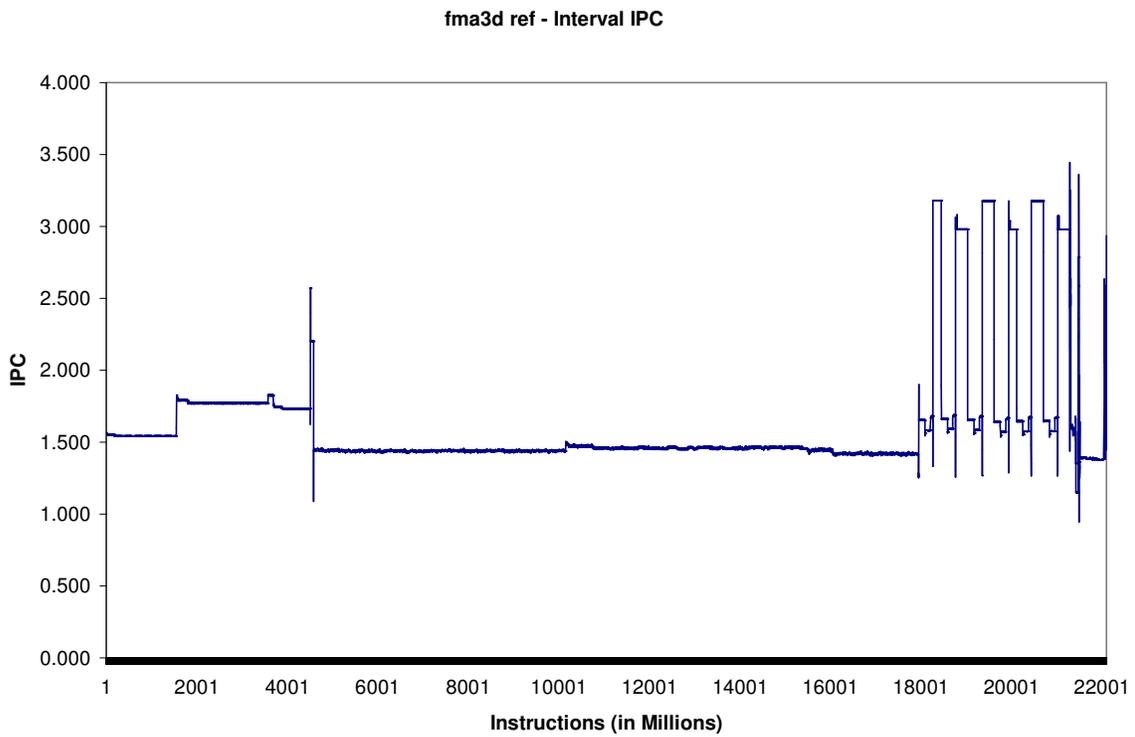
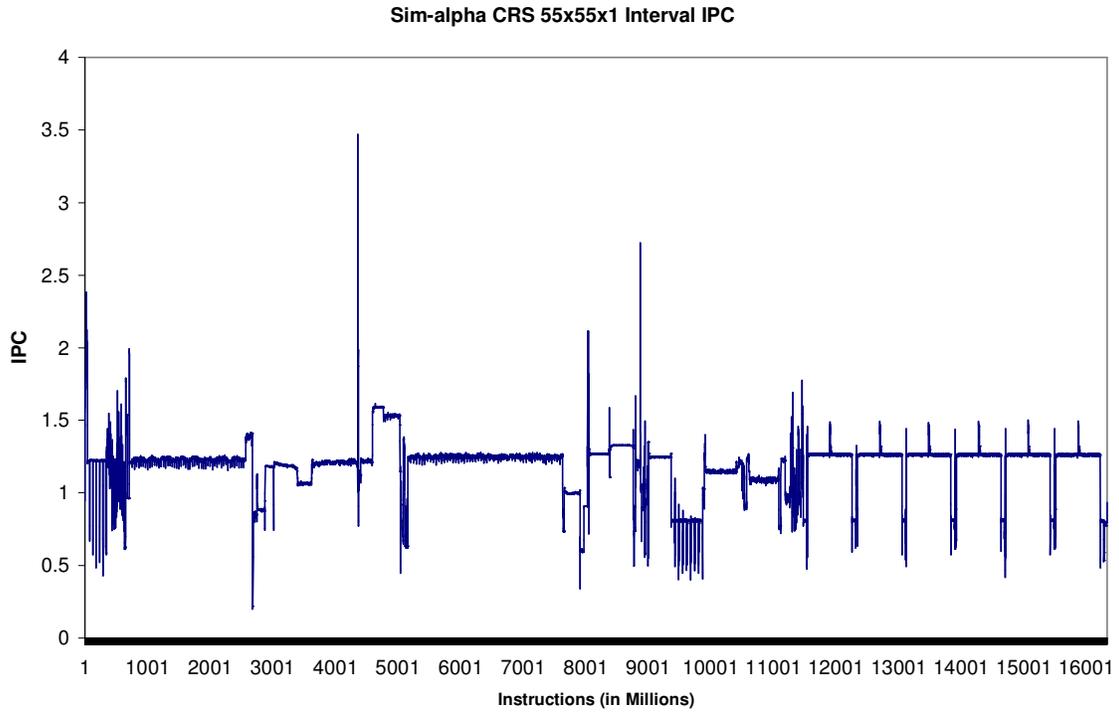
REFERENCES

- [1] "Standard performance evaluation corporation (SPEC)," <http://www.spec.org>.
- [2] "Compressed Row Storage," http://netlib2.cs.utk.edu/linalg/html_templates/node91.html.
- [3] "Sun™ S3L 4.0 Software Programming Guide," <http://www.sun.com/products-n-solutions/hardware/docs/html/817-0086-10/prog-sparse-support.html>.
- [4] The "cube" test problem document
- [5] L. Hu and I. Gorton, "Performance Evaluation for Parallel Systems: A Survey," *University of NSW, Sydney 2052 Australia*, pp. 1-56, Oct. 1997.
- [6] R. Jain, "The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling," *Wiley-Interscience*, New York, NY, April 1991.
- [7] J. Fu, and J. Patel, "Trace Driven Simulation using Sampled Traces," *Proc. 27th Ann. Hawaii Int'l Conf. on System Sciences*, pp. 211-220, 1994.
- [8] R. Uhlig and R. Mudge, "Trace-Driven Memory Simulation: A Survey," *ACM Computing Surveys*, vol. 29, no. 2, pp. 129-169, June 1997.
- [9] "SimpleScalar Simulators," <http://www.simplescalar.com>.
- [10] R Desikan, D. Burger, S. Keckler, and T. Austin, "Sim-alpha: a Validated, Execution-Driven Alpha 21264 Simulator," *Tech Report TR-01-23*, Dept. of Computer Science, University of Texas.
- [11] P. Bose and T. Conte, "Performance Analysis and Its Impact on Design," pp. 41-49, *IEEE Computer*, 1998.
- [12] Y. Luo and K.W. Cameron, "Instruction-level Characterization of Scientific Computing Applications Using Hardware Performance Counters," Scientific Computing Group, Los Alamos National Laboratory, 1998.
- [13] G. Griem, L. Oliner, J. Shalf, and K. Yelick, "Identifying Performance Bottlenecks on Modern Microarchitectures using an Adaptable Probe," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, pp. 255, IEEE Computer Society, 2004.
- [14] "Sustainable Performance Scaling for High-end Enterprise and Technical Computing, Intel Itanium 2 Processor," White Paper, Intel Corporation, 2004.

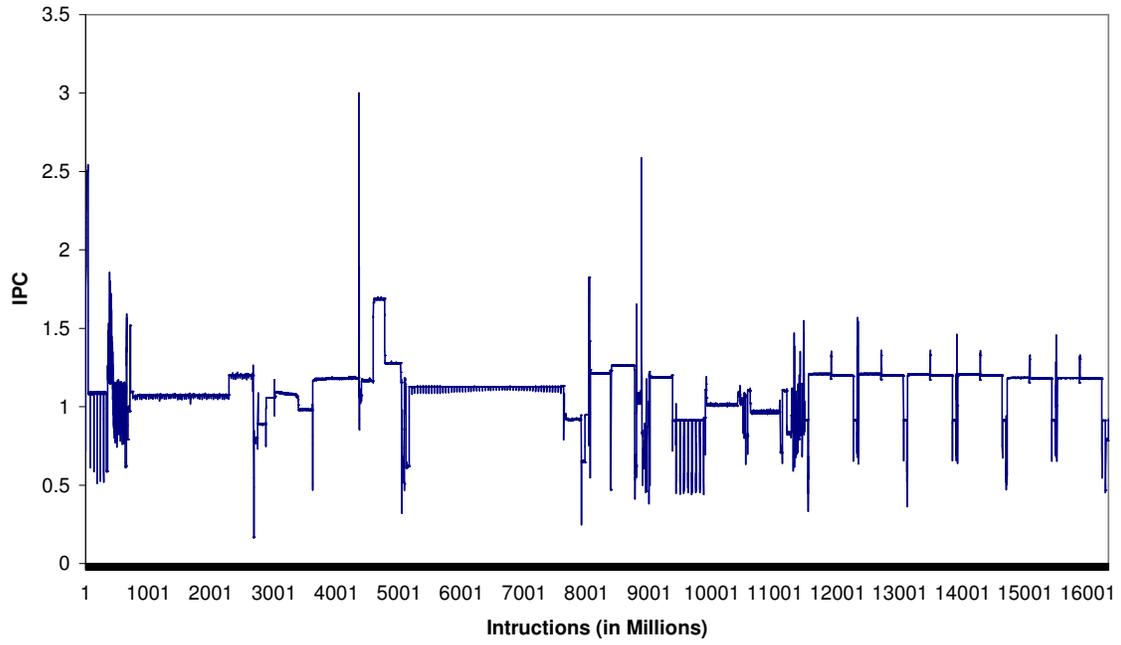
- [15] "Intel Itanium 2 Processor," Hardware Developer's Manual, July 2002.
- [16] Block diagram source
- [17] "SGI Altix 3000 with New Itanium 2 Processor Leads Competition on HPC Benchmarks," Silicon Graphics, Inc, June 24, 2003.
- [18] "SGI® Altix™ 3000," Intel® Itanium® 2-based (Madison) Benchmark Results June 2003.
- [19] A. Dharia, P. Rodriguez, and R. Verret, "Selective Fill Data Cache," Rice University, 2003.
- [20] S. Chaki and D. Mazzone, "Investigating Load Redundancy," Carnegie Mellon University, 1999.
- [21] J. Sahuquillo and A. Pont, "Splitting the Data Cache: A Survey," pp. 30-35, *IEEE Concurrency*, 2000.
- [22] A. Naz, M. Rezaei, K. Kavi, and P. Sweany, "Improving Data Cache Performance with Integrated Use of Split Caches, VictimCache and Stream Buffers," Department of Computer Science and Engineering, University of North Texas, 2004.
- [23] "Cray X1 System,"
<http://www.nas.nasa.gov/Users/Documentation/X1/hardware.html#cache>.
- [24] L. Oliker, R. Biswas, J. Borrill, A. Canning, J. Carter, M.J. Djomehri, H. Shan, and D. Skinner, "A Performance Evaluation of the Cray X1 for Scientific Applications," http://crd.lbl.gov/~oliker/papers/vecpar_2004.pdf.
- [25] D. Albonesi, R. Balasubramonian, S. Dropsho, S. Dwarkadas, E. Friedman, M. Huang, V. Kursun, G. Magklis, M. Scott, G. Semeraro, P. Bose, A. Buyuktosunogiu, P. Cook, and S. Schuster, "Dynamically Tuning Processor Resources with Adaptive Processing," pp. 49-58, *IEEE Computer*, 2003.
- [26] V.E. Taylor, "Sparse Matrix Computations: Implications for Cache Design," in *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pp. 598-607, 1992.
- [27] O. Temam and W. Jalby, "Characterizing the Behavior of Sparse Algorithms on Caches," in *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pp. 578-587, 1992.

APPENDIX A

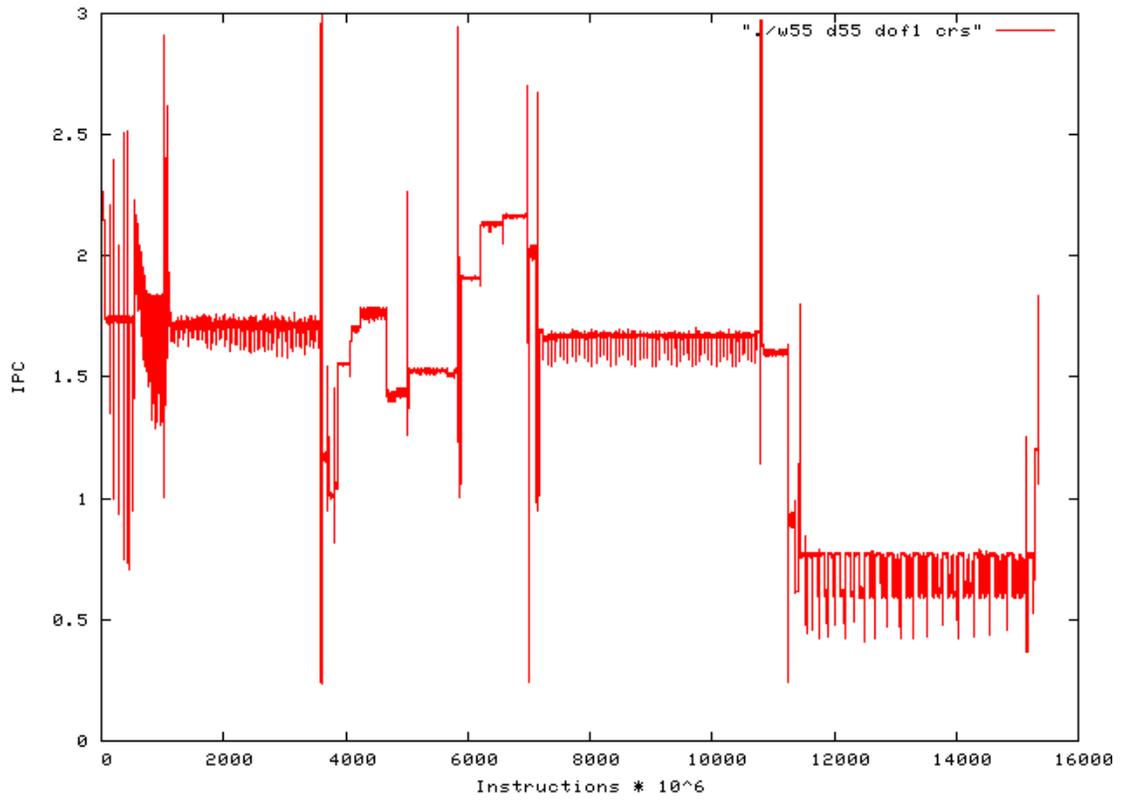
1 CRS 55x55x1 – Interval IPC Graphs



CRS 55x55x1 - Interval IPC
Sim-alpha Itanium Configuration

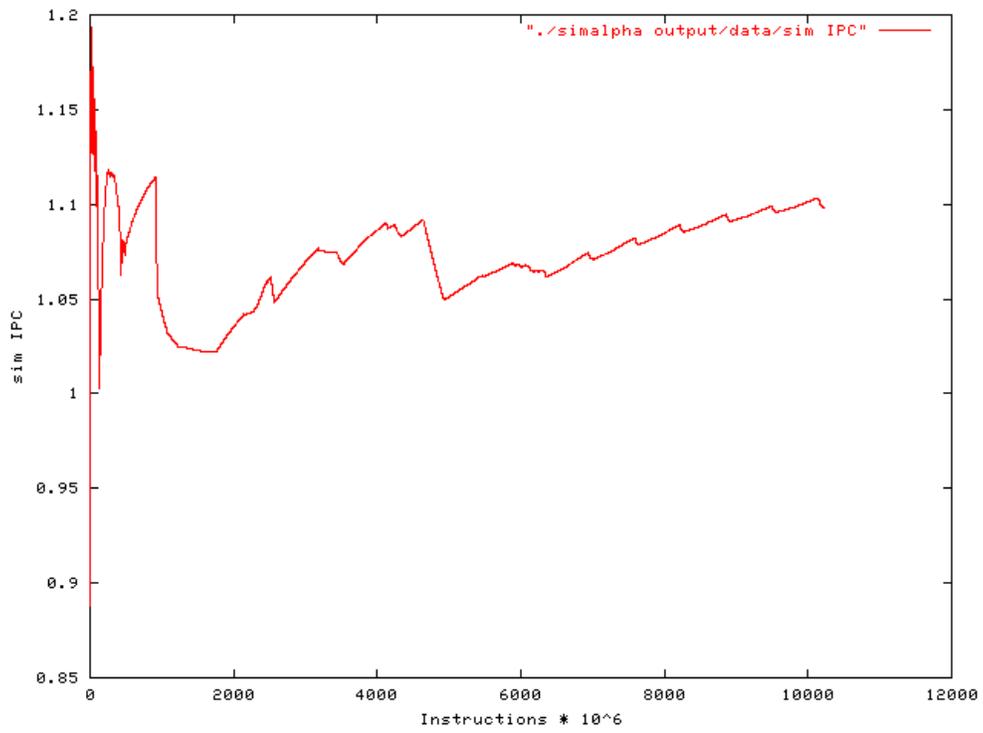


CRS 55x55x1 - Interval IPC
(Itanium2 Processor)

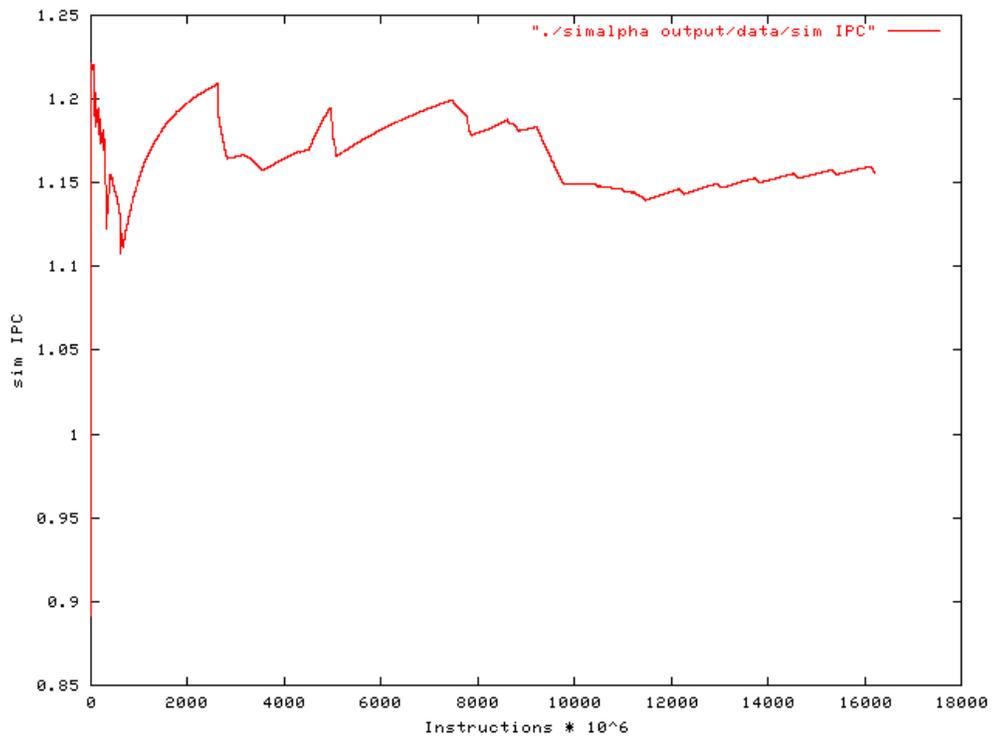


2 CRS Degrees of Freedom = 1

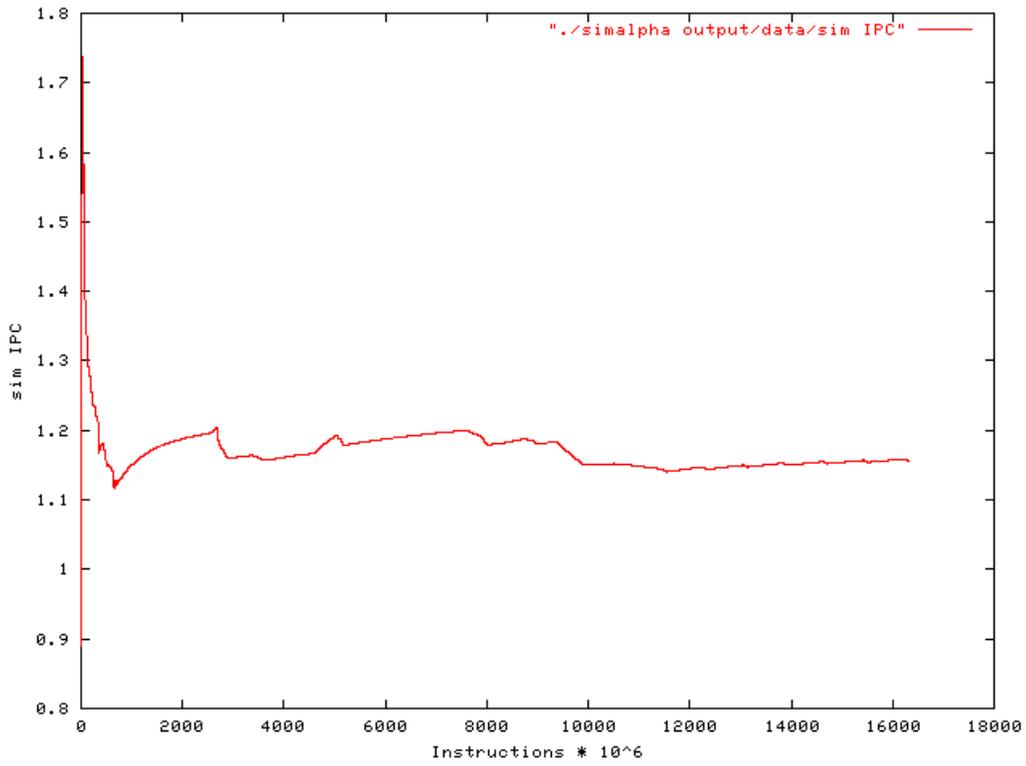
CRS 1x43903x1



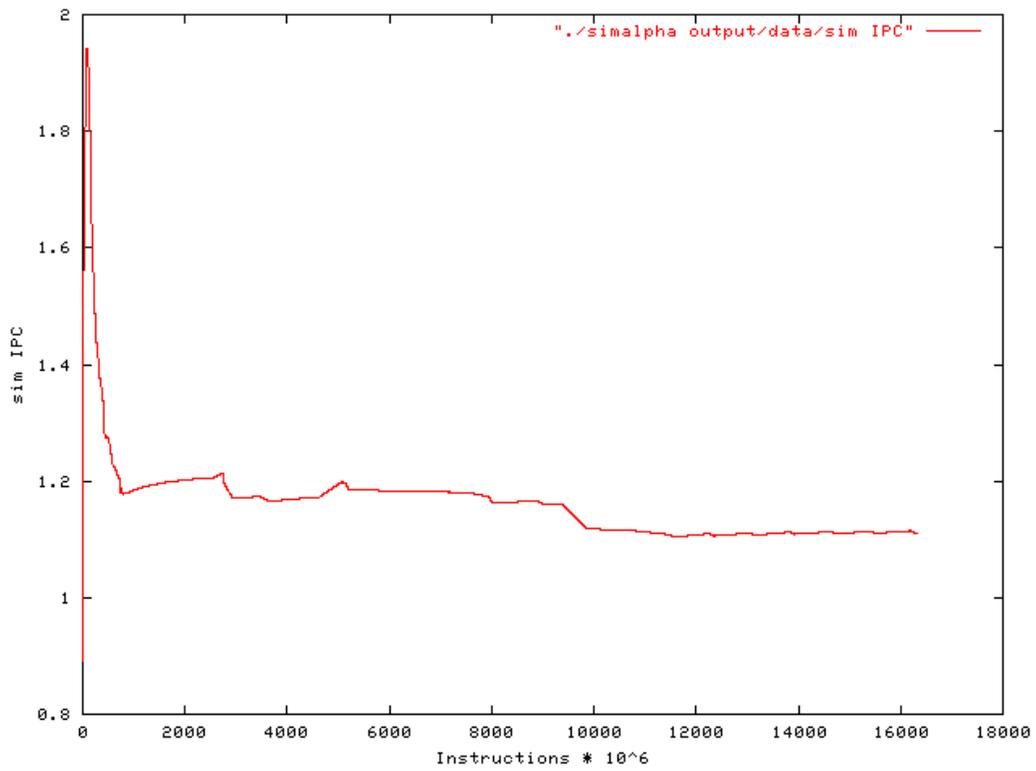
CRS 27x223x1



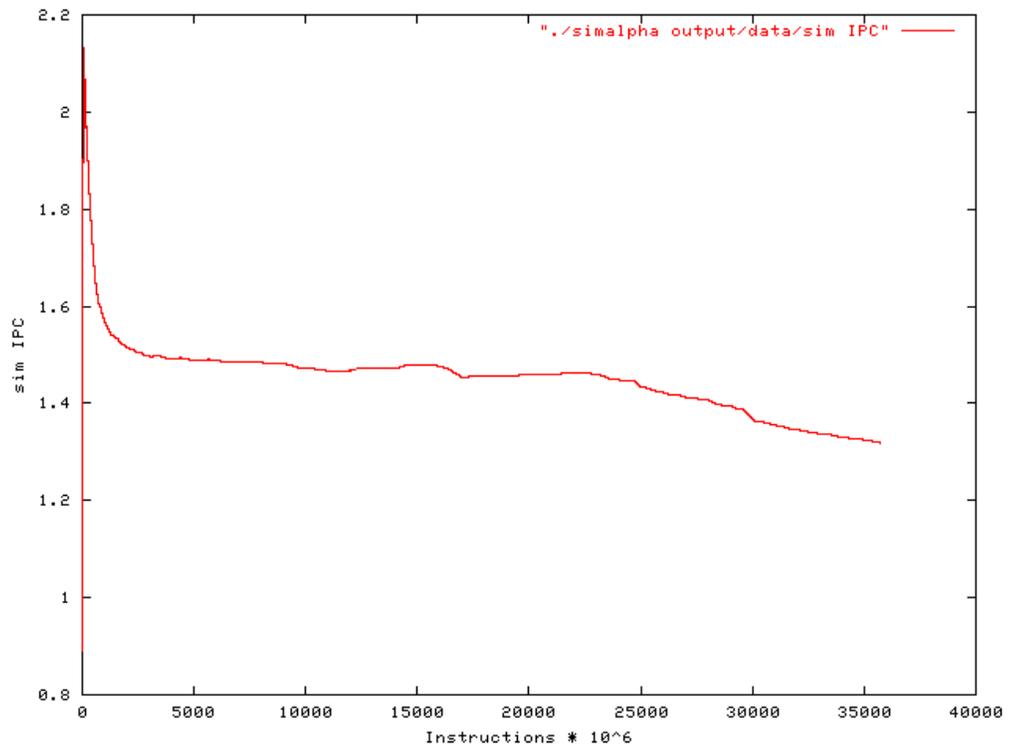
CRS 55x55x1



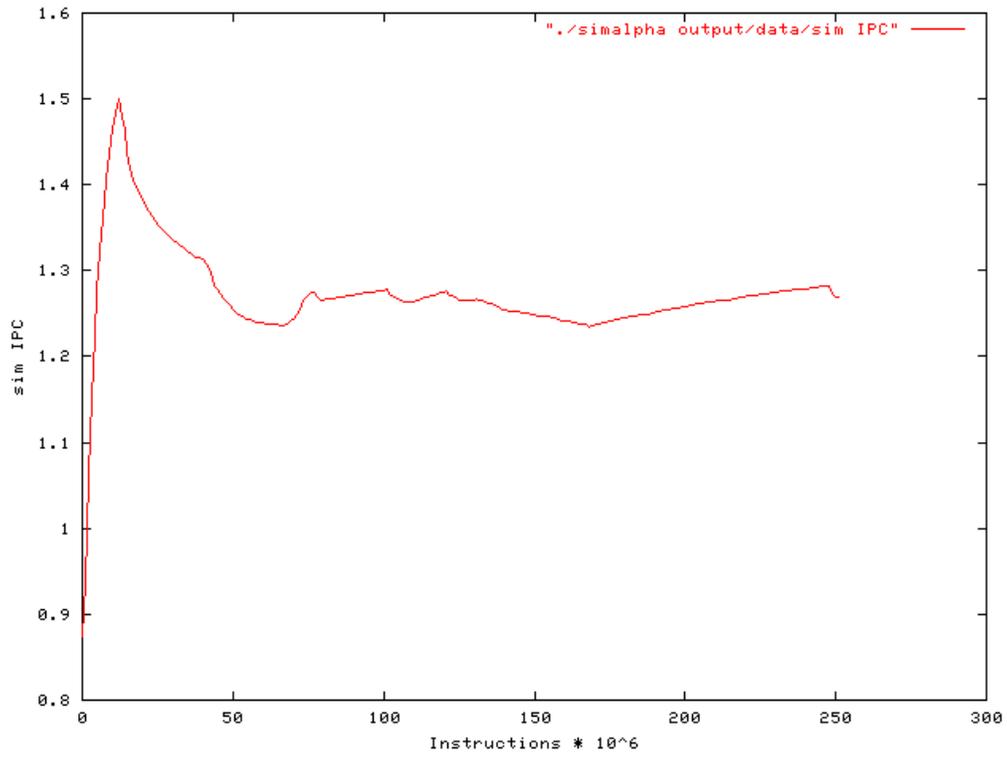
CRS 78x27x1



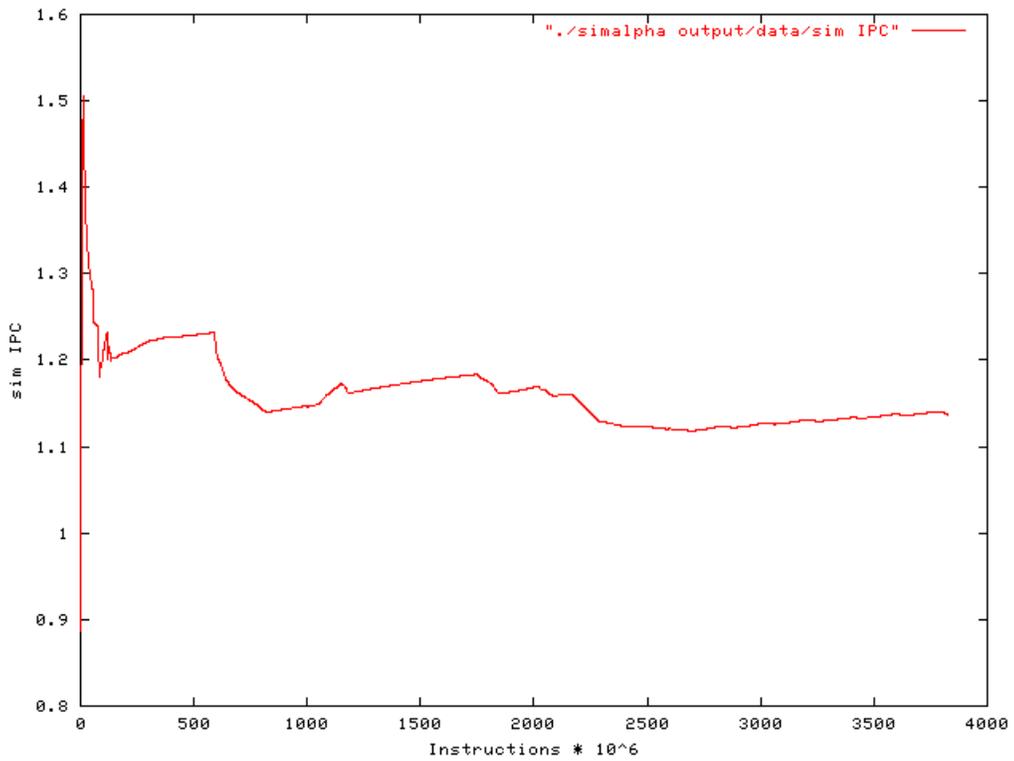
CRS 295x1x1



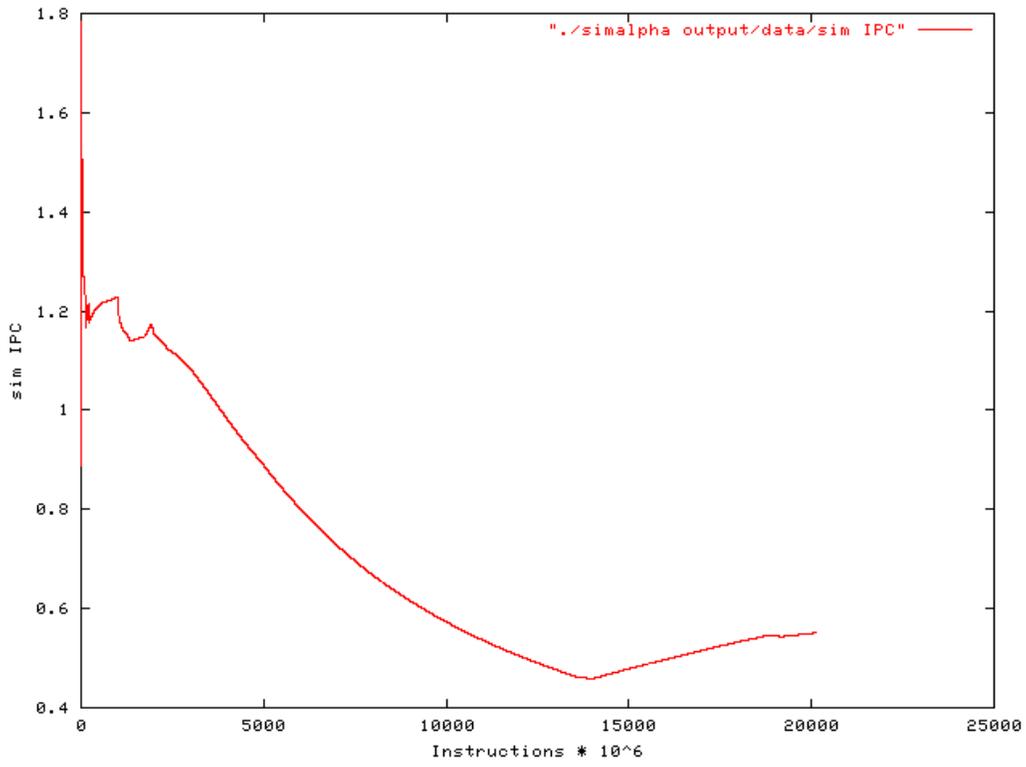
CRS 40x1x1



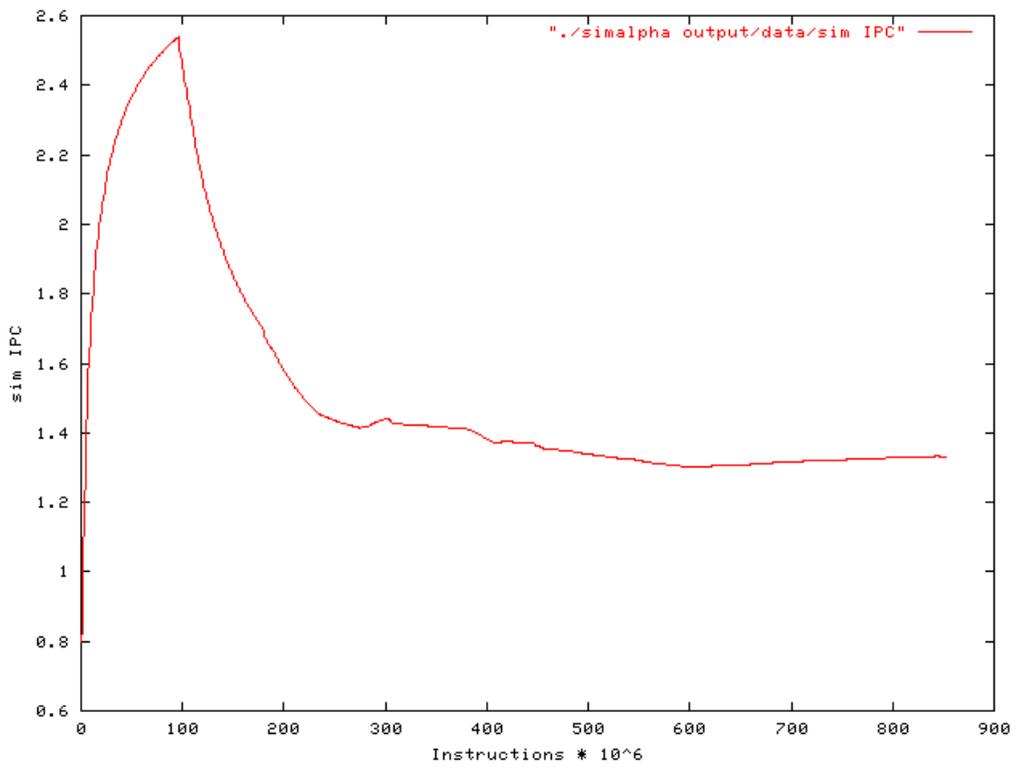
CRS 40x24x1



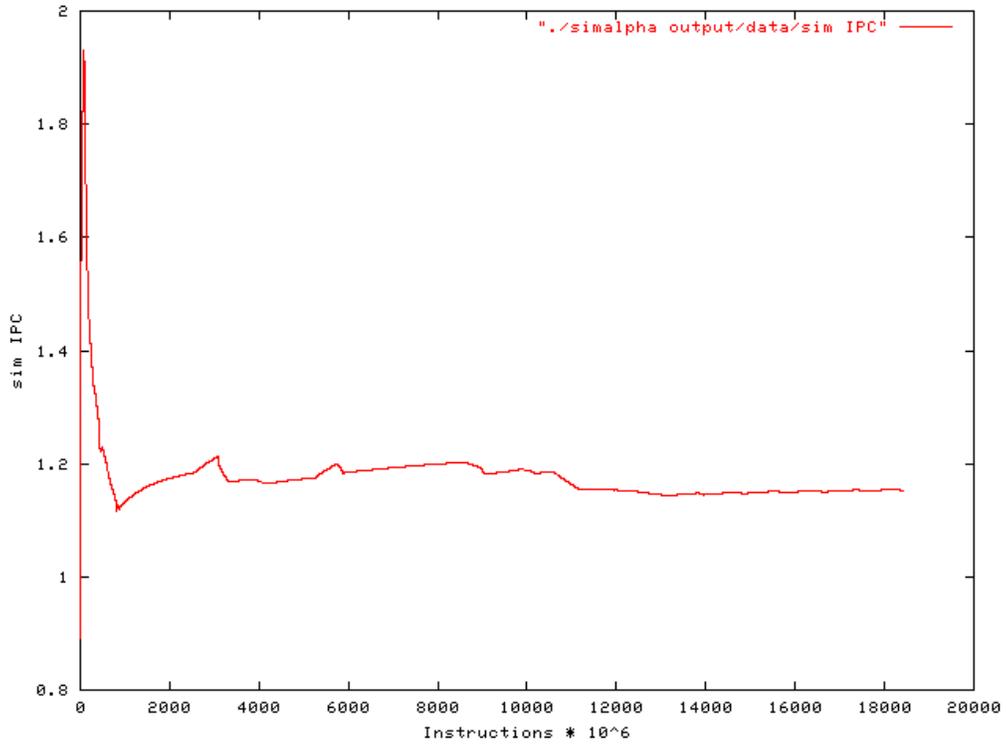
CRS 40x40x1



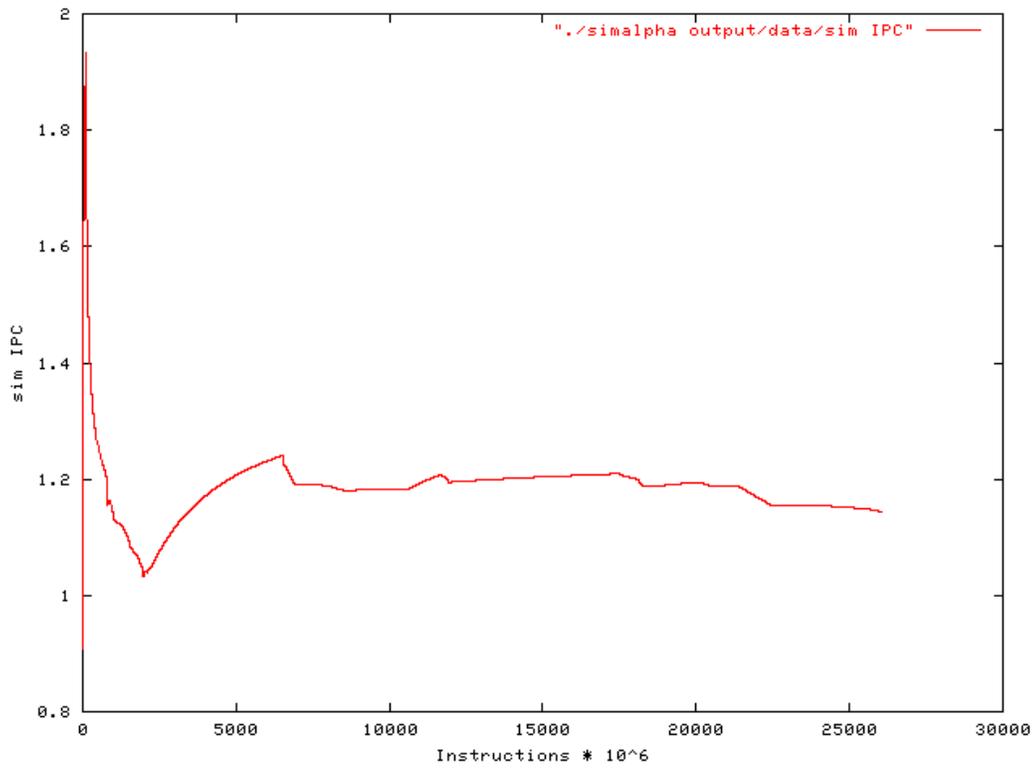
CRS 72x1x1



CRS 72x36x1

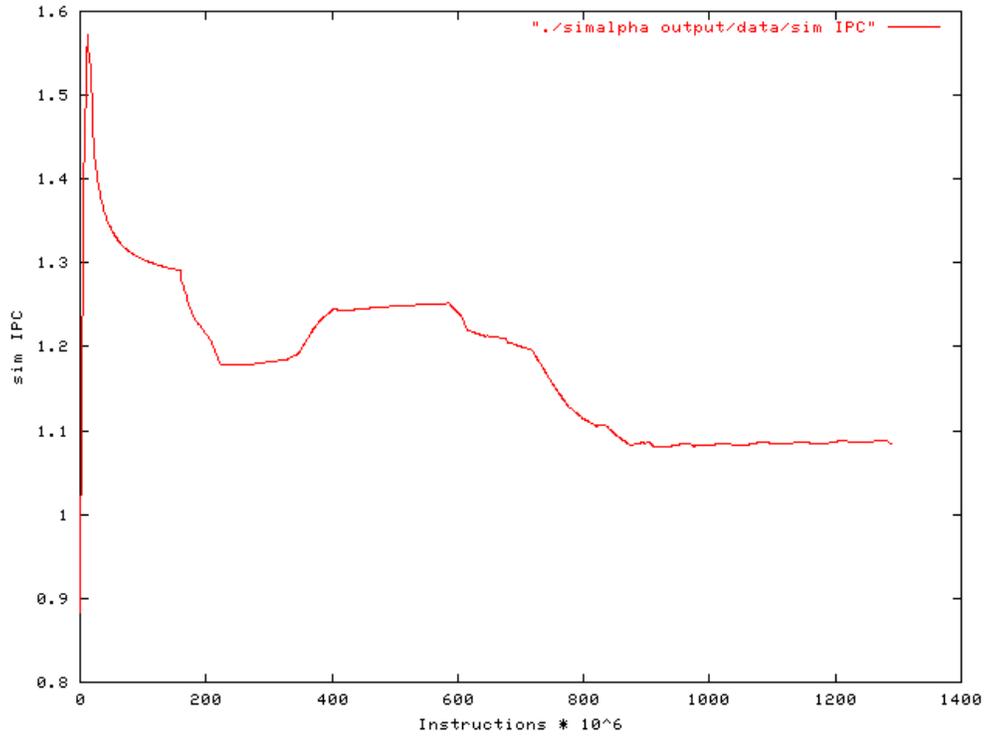


CRS 72x72x1

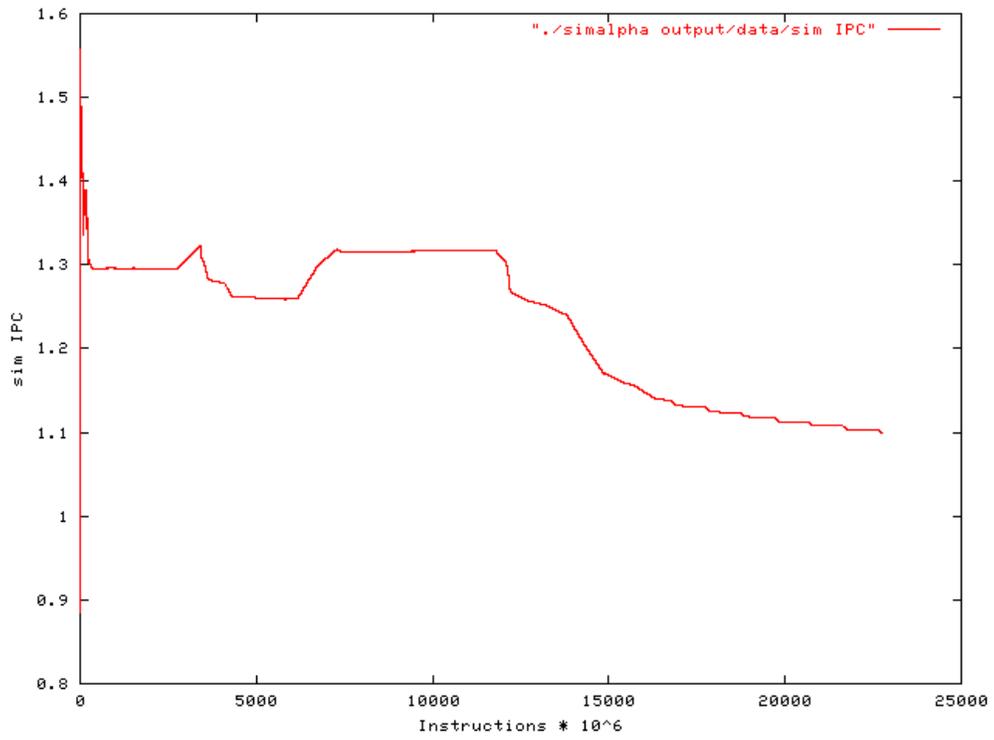


3 CRS Degrees of Freedom = 3

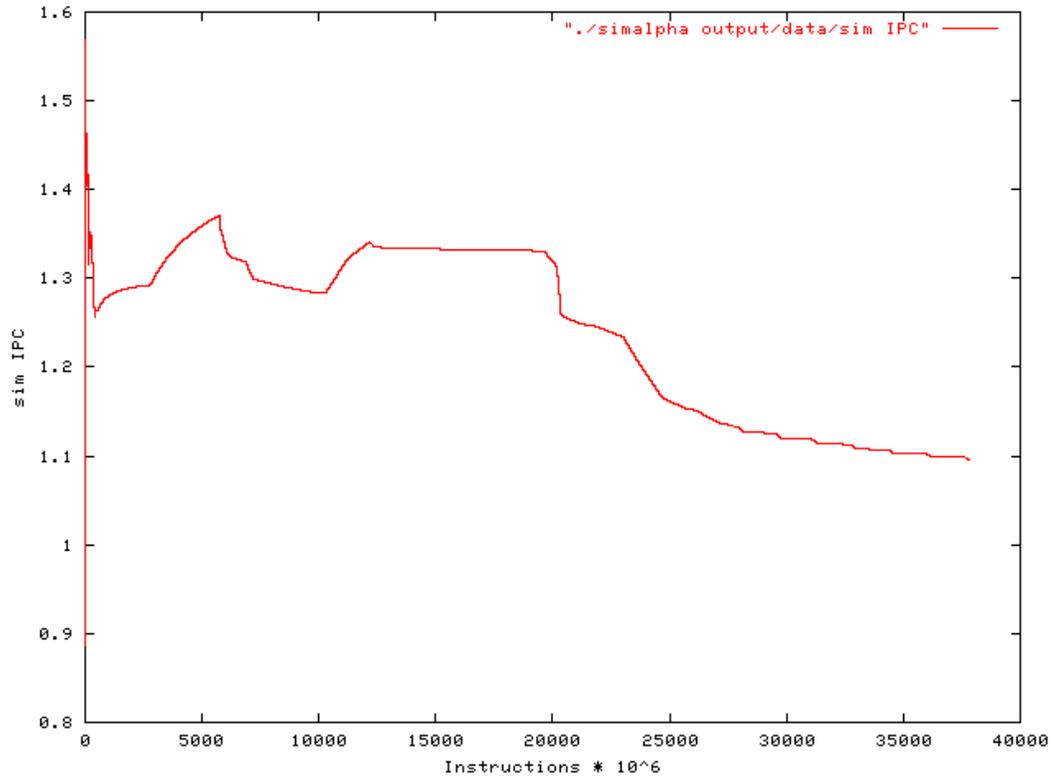
CRS 40x1x3



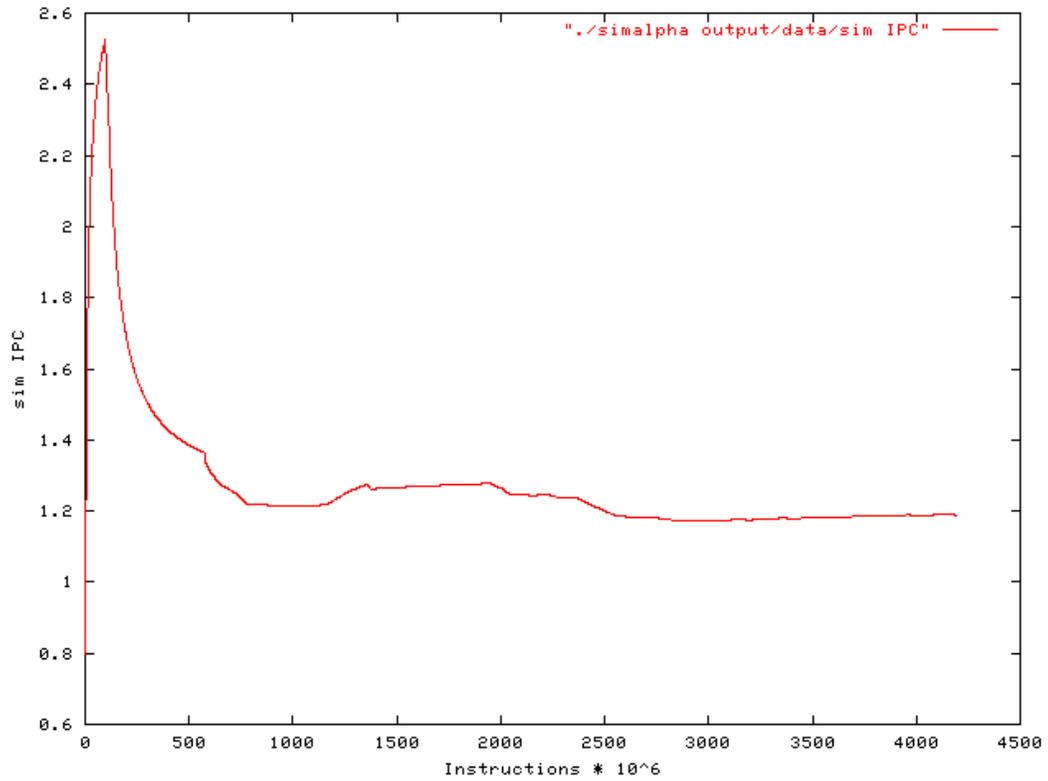
CRS 40x24x3



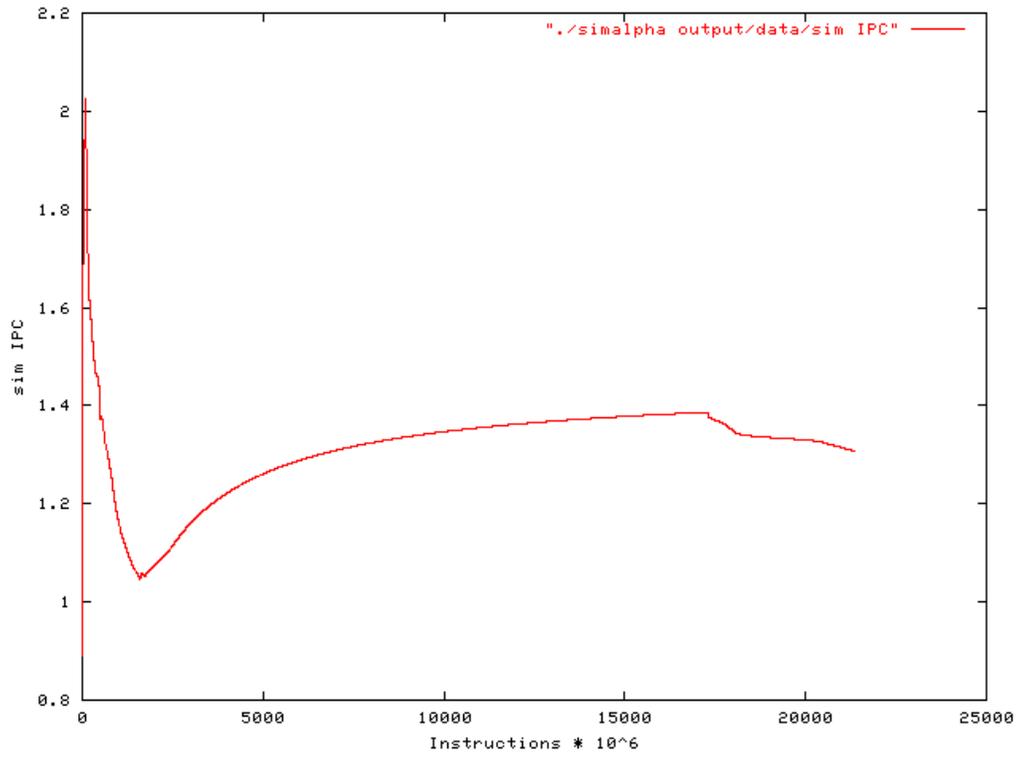
CRS 40x40x3



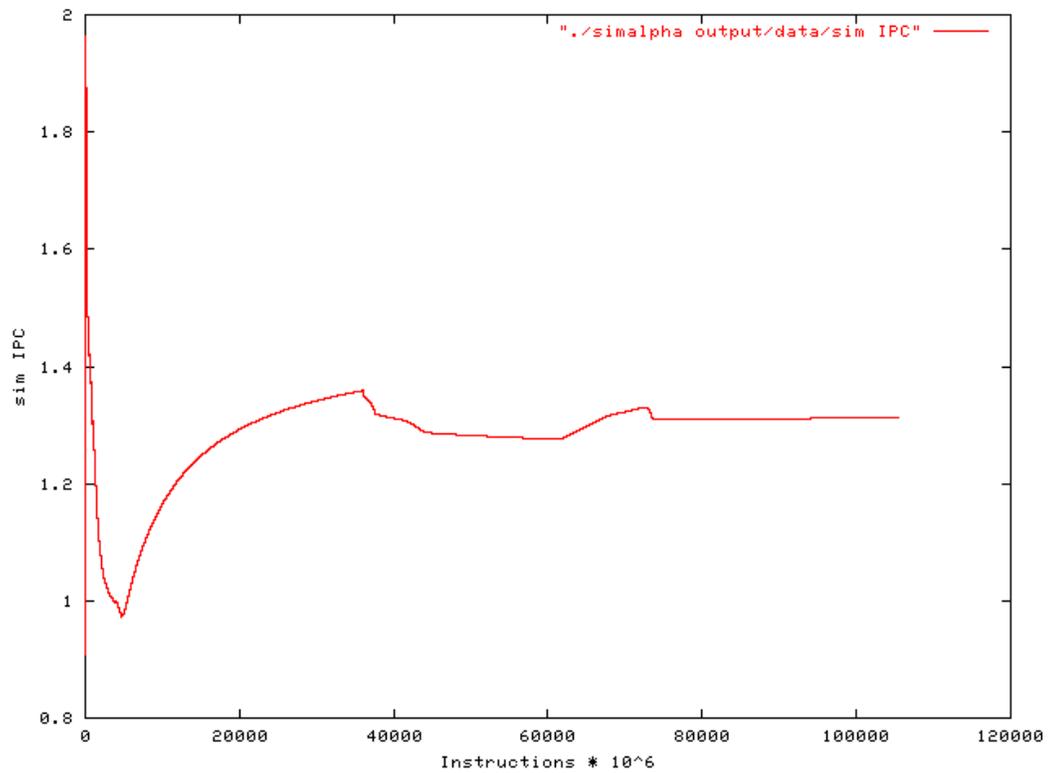
CRS 72x1x3



CRS 72x36x3

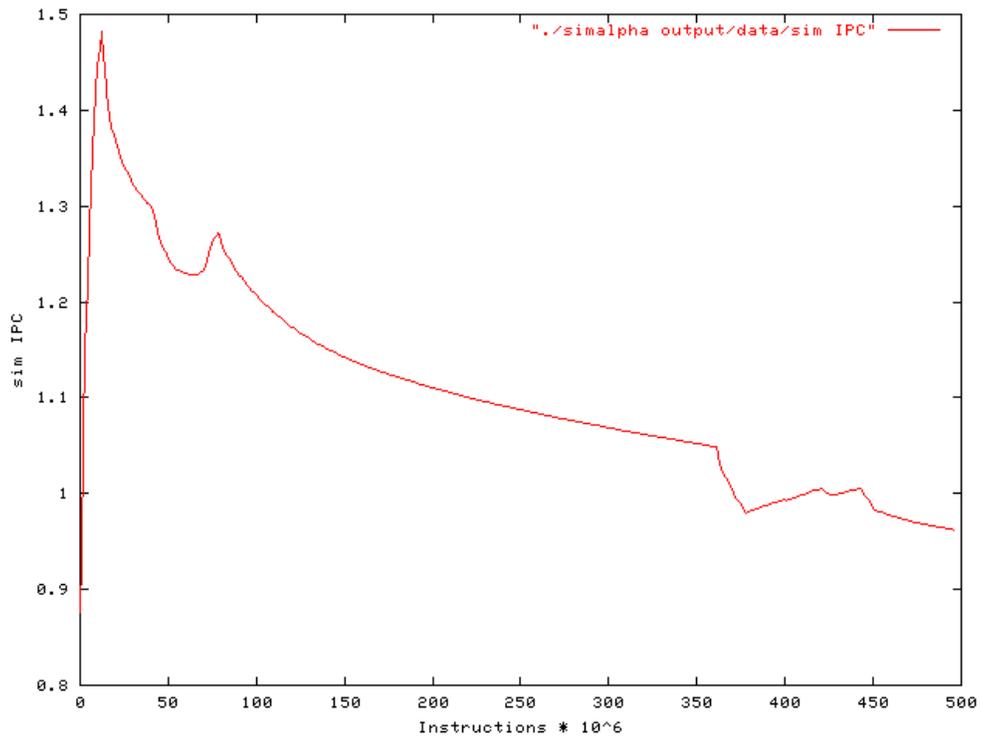


CRS 72x72x3

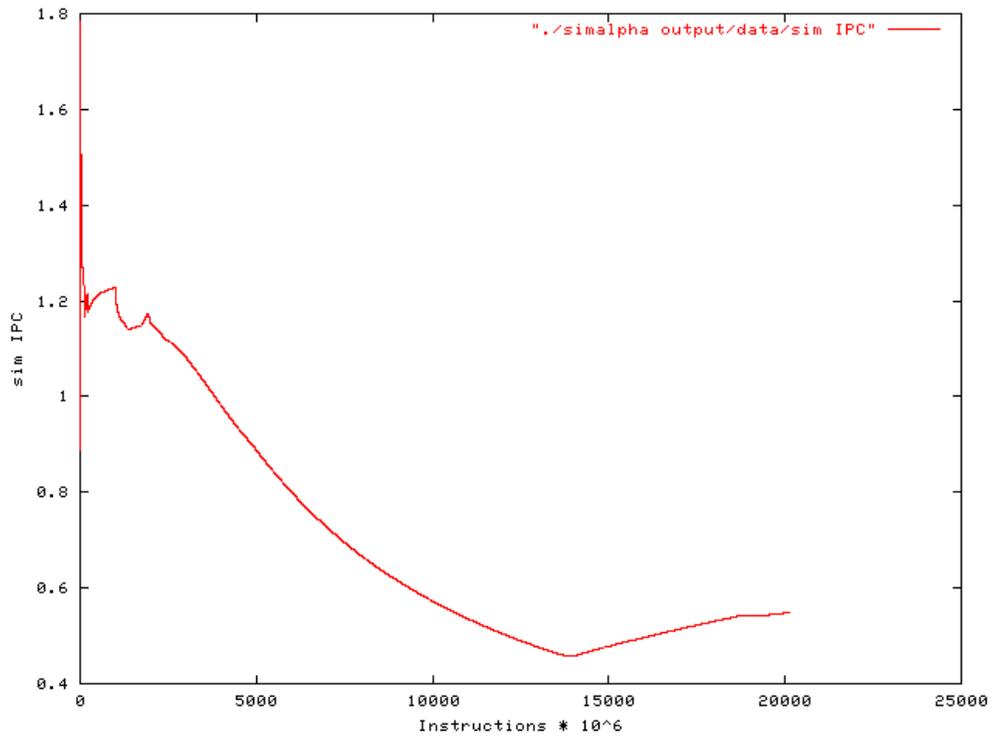


4 VBR Degrees of Freedom = 1

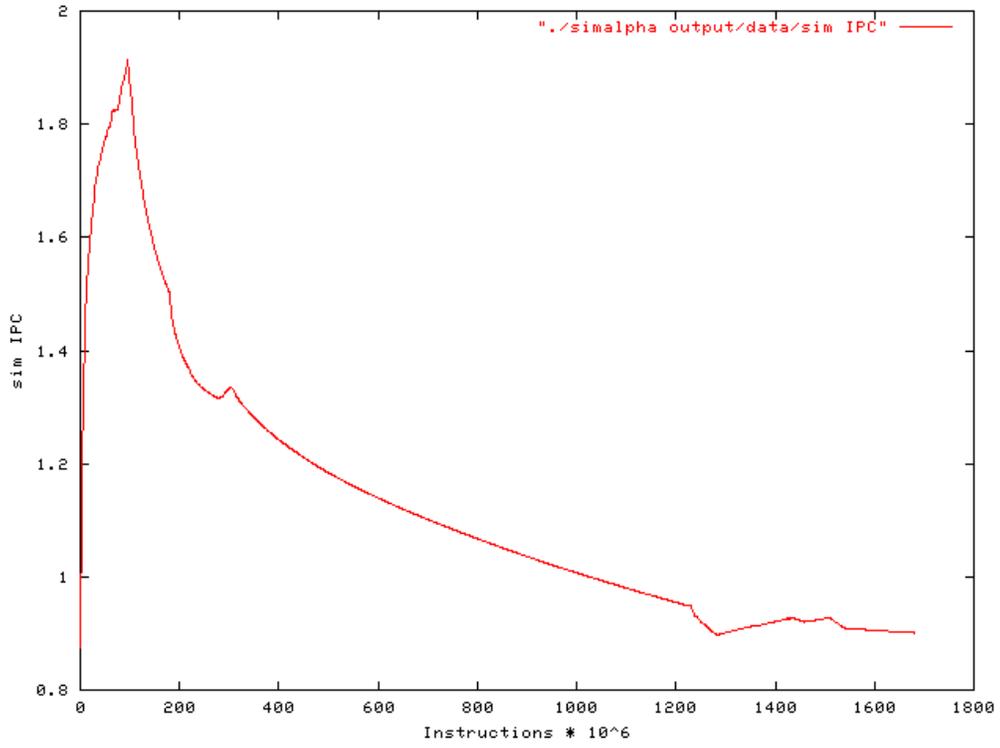
VBR 40x1x1



VBR 40x40x1



VBR 72x1x1



VBR 72x72x1

