Sampling Benchmarks: Methods for Extracting Interesting Segments of Programs

by

Peter D. Finch

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 21, 1998
[June 1999]

Author____
Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by____
Arvind
Thesis Supervisor

Accepted by____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Sampling Benchmarks: Methods of Extracting
Interesting Segments of Programs

by

Peter D. Finch

Submitted to the
Department of Electrical Engineering and Computer Science

May 21, 1999

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering
And Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

This thesis evaluates schemes for compressing benchmark programs by finding a subset of the program that is sufficiently similar to the whole to be considered representative for statistical study. These schemes are computationally inexpensive means of evaluating a whole program and finding the interesting sub-parts. Detailed analysis can then be run on those sub-parts, hopefully with some assurance that the results are applicable to the entire program. Six different analysis techniques are examined with respect to four different benchmark program classes. The results of this analysis are used to produce a set of heuristics for benchmark sampling. Also, the results provide statistical validation of various sampling techniques.

Thesis Supervisor: Arvind
Title: Professor of Computer Science and Engineering

Acknowledgments

First, the author would like to thank SGI and the 6-A program for three years of support and encouragement. Without their resources and people, this thesis would not have been possible. Amr Zaky of Apple Computer, formerly of SGI, must be recognized. His help in the production of this thesis was invaluable. The author would also like to thank James Tornes, Eric Miller and Marty Deneroff of SGI for their assistance, technical and otherwise. Professor Arvind provided the author with a much needed link to MIT and served as a reminder that, in the end, the thesis needed to be graded, and that the author therefore ought to try to do a good job. Finally the author wants to thank his fiancee, Nora Humphrey, for putting up with him through it all.

Table of Contents

"There are lies, damn lies, and there are statistics. And then there are benchmarks."

-Peter H. Lewis, 1998

1.0 Introduction

1.1 Purpose

This thesis evaluates schemes for compressing benchmark programs by finding a subset of the program that is sufficiently similar to the whole to be considered representative for statistical study. Ideally we use an inexpensive (computationally speaking) means of evaluating the whole program and finding interesting sub-parts. We can then run detailed experiments on the much smaller parts interactively and draw valid conclusions about the original program's behavior. This research will examine several such methods for selecting sub-parts with respect to different program types, and determine their usefulness in finding valid data for research questions like the characterization of memory system performance or I/O system studies. It will attempt to delineate when and how particular extraction techniques are important and useful, and suggest future avenues for the development of such methodologies.

To this end six different analysis techniques and four different benchmarks were produced, each representative of a different style of program. The result of each analysis technique is a subset of the original trace, selected because it has certain statistical properties. We then use information gained through experiments on real runs of the benchmark programs (which are instrumented in a minimally intrusive fashion to assess the representativeness of our compressed trace for the particular behavior we are examining.

1.2 The Nature of the Problem

Benchmark programs are big, and they are getting bigger. The SPECint95 benchmarks each run in around 10 minutes on a modern high-performance workstation. (SPEC, 1999) Running them in detailed simulation would typically take days to weeks each. This is not practical for the conducting of architectural research. And the SPEC benchmarks are widely criticized as being too small to be properly representative of modern workloads. (Patterson, Hennessy, 1996) Often real computer companies have proprietary benchmarks that can take hours to run on their fastest machines, because it is these workloads that customers care about and base their purchase decisions on.

Ideally computer architects would be able to use these real workloads to drive their research. That is, they would like to optimize their machines to run the workloads they will see in the real world as fast as possible. But this is not possible with contemporary technology. Simulation systems typically run factors of hundreds, thousands, or even tens of thousands slower than actual hardware, and that makes it impractical to run more than very small simulations in full detail. (Bedichek, 1990; Cmelik, 1994; Fujimoto, Campbell, 1988; Magnusson, Werner, 1995; Rosenblum, Witchel, 1996)

This means that since architectures must be based on their performance against only very small benchmarks, since these are the only metrics we really have to compare experimentally, great care must go into constructing good, small benchmarks. If you can come up with a 100,000 instruction code segment that accurately reflects the behavior of a large Oracle database benchmark (that might be a trillion instructions itself) (Gray, 1991), then you can use that code segment to drive your architectural research. For example, you could evaluate several possible instruction cache replacement policy designs in several hours, instead of several months, or several possible fetch unit designs in a day, instead of a season.

In fact, this is what people actually do. There is no hope of being able to evaluate an entire large application in a detailed simulation environment, so they select some smaller program, perhaps an excerpt of the large one, or perhaps an analogous piece of code, or perhaps even the same piece of code with a smaller argument (for example, compressing a 128x128 pixel image, rather than a 512x512 pixel one in an image processing software package). This small program is then used to drive simulations, and conclusions are drawn based on the data provided by these simulations that then go and drive new designs.

A scientific mind ought to quickly phrase a question based on this description of the architectural research process. How are these smaller programs selected, and is there any assurance that their behavior is representative of the behavior of the large, real programs? In particular, how do we know that the conclusions we are drawing from our shrunken benchmarks are in fact valid for real programs running on real machines?

This thesis intends to examine these techniques and formally establish the validity of some of the methods used. It will also suggest when each particular technique is useful, and flesh out their strengths and weaknesses.

1.3 Questions About the Validity of Current Pruning Methods

Present-day pruning techniques are a mix of ad-hockery, intuition and intelligent guesswork. But there really is not a great deal of validation of the techniques, and only a small amount of effort has gone into theorizing about good techniques, let alone experimentally validating them.

The most common modern techniques are random and semi-random selection, although there are some more sophisticated techniques in use in specialized environments. Random selection is exactly what it sounds like: every 100th million instructions is chosen and used for experiments, or four one-million instruction segments are selected at random — no basis is used to select segments other than chance or fair distribution. Often though, architects already have some idea of what they want to include. Some segments are chosen at random and a known important segment is forcibly included.

Specialized techniques include path counts and library-only tests. Path counts are used to select segments for micro-architecture research because they provide very good characterization of extremely low-level behavior. (Carroll, 1998) They are discussed in the related work section, but are really outside of the scope of this thesis since they perfectly apply to a simple case, but are expensive to deal with relative to their questionable utility in general cases. Library-only tests simply choose only to execute library code, for example queuing all the OpenGL (Segal, Akelely, 1997) calls in an engineering application and using them to represent the whole benchmark. This is useful when one wants to test a particular piece of hardware driven by the library and is frequently used to exercise graphics attachments. Since this is again very specific, and also easy to do, it is largely outside the scope of this thesis.

This thesis will concentrate on more general purpose techniques. How can we select segments that are representative of a program's gross performance characteristics? How can we predict cache miss rates? Translation Look-aside Buffer (TLB) misses? Concentrations of branches? These artifacts are important to all systems designers, whether they are building supercomputers with all SRAM memory or PCs with tiny caches.

The compression techniques will be evaluated with respect to a LISP benchmark based on Li from SPECINT, an mpeg player benchmark, and two very different segments of tomcatv (from SPECFP), the first half-billion instructions where behavior is irregular and hard on the memory system, and the middle of tomcatv which is very regular and stresses the memory system in a near-vector fashion. These four programs are very different in behavior and represent large classes of common programs.

1.4 Trace-Based Research

Computer companies frequently have large proprietary benchmarks. These would typically include database software for a big-iron manufacturer, real engineering applications for the workstation maker, and perhaps spreadsheet applications for the typical PC company. Often these programs would be developed in concert with the independent software vendor that produces the real program.

There are several ways to learn from these benchmark programs. They can be instrumented to record statistics and run on current generation machines. This produces very accurate results for the whole program, but it is of limited relevance in considering future machines that may differ greatly in design or scale. They can also be used to create traces to drive simulators. Traces are records of all the instructions (and their arguments) processed during the execution of a program. They are a fairly complete picture of the execution of a piece of code, at least from a systems perspective. (Traces can hide low-level timing details that may be important.) These records can then be used to drive simulators, or be combed over for insight into program behavior.

The problem with traces is scale. Real benchmark programs leave huge traces. For example, tomcatv from SPEC95fp executes around 25 billion instructions. A trace may record 4 to 40 bytes (things to store include the instruction itself, the PC of the instruction, the contents of the registers used, any memory addresses accessed for instructions or data, the contents of any memory locations addressed, etc.) per instruction, a total range from 100 to 1000 gigabytes. This is almost too much to store, much less process in an interesting fashion. And, tomcatv is hardly the largest program whose performance people care about. Tomcatv runs in about 5 minutes on modern machines. Programs used to analyze more diverse behaviors often take longer. This occurs in benchmarks for real applications like ProEngineer, or Oracle or Synopsis databases. In these cases, different modes of the software must be investigated, I/O occurs to slow things down, etc. (Ousterhout, 1990; Rosenblum et al, 1995) These benchmarks can take hours to run and execute trillions of instructions. We are unable to do detailed analysis of programs this large.

To see why this is so, consider a naive trace dump driving a cache simulator. Some simulation managing program would read the address stream of the trace and use it to drive a cache simulator, eating the instructions as they go by, thereby negating the storage problem but leaving us with the huge amount of data to process on the fly. Typical slowdowns are a factor of 100, 1000 or even 10,000 for reasonably complicated simulators (Rosenblum et al, 1995; Rosenblum, Witchel, 1996; May, 1987). This potentially turns our 5 minute tomcatv run into a three and a half day ordeal. An hour of transaction processing is a month long experiment. This is simply impractical, and even for small programs painful. Certainly it makes interesting interactive experiments an impossibility.

Architecture researchers are thereby prompted to devise ways to speed up the process. Typically they resort to sampling the benchmark, choosing key segments, and using them to drive experiments and simulators.

These segments are typically selected by hand or at random, as previously mentioned. In specialized cases they may be selected by a more formal scheme. Generally there is no systematic effort to confirm the validity of these techniques.

Detailed simulations can be done with fractions of programs via a variety of schemes, though, so we can at least be confident that our results accurately represent the behavior of the programs even during the sampled phase, though they may not tell us anything about the rest of the program. This can be accomplished in several ways. Often architects use ad-hoc schemes to simulate cache warming, for example. To simulate the middle of a large program they will execute that segment in their cache simulator, and every time they see an address they have not seen before they will toss a weighted coin (weighted by the overall program's miss-rate) and decide wether to miss or not. (Zaky, 1998) While this probabilistic scheme is obviously not 100% accurate it is probably good enough if the segment is large.

Recently advanced simulation environments like SimOS have added greatly to the ability to do this kind of work. With multiple levels of simulation varying greatly in detail and performance, and the ability to switch dynamically among them, we can do much better. (Rosenblum, Witchel 1996)A benchmark can be run extremely quickly, with perhaps a 10x slowdown, for example in Embra mode in SimOS which uses binary translation (Bartlett, 1989; Cmelik, Keppel, 1994; Rosenblum, Witchel, 1996), and switched to a higher detail (with perhaps a 10,000x slowdown) mode as it enters an important segment. The various phases can communicate information so the detailed mode has it caches warmed in at least a somewhat more accurate fashion. Other simulation packages like Shade and ATOM (Srivastava, 1994) provide similar functionality. Support for the methodology is widespread.

We are stuck with fractional analysis. Benchmarks are too long to analyze in their entirety What is needed is a validation of techniques for making selections from longer programs; a confirmation that the selected segments really represent what is going on in the whole program. With that information the risk of relying on these abbreviated experiments goes down tremendously.

1.5 Chapter Summary

Chapter 2 will discuss related work, particularly the PathOMatic system developed at SGI for micro-architecture study. Chapter 3 discusses various other possible analysis techniques, particularly those used in this project. Chapter 4 discusses the software developed and the experimental methodology of this thesis. Chapter 5 gives detailed experimental results and chapter 6 gives conclusions and future directions.

2.0 Related Work

2.1 PathOMatic

To illustrate the current state-of-the-art, consider a technique developed by Steve Carrol (Carroll, 1997) at Silicon Graphics: PathOMatic. PathOMatic was developed within MIPS, then SGI's microprocessor group, to compress benchmarks for micro-architectural research. That is, it was designed to select a collection of segments of a larger program that had representative behavior at the level of small groups of instructions. Ideally, these selected sections would produce a representative sample of the various forms of pipeline behavior found throughout the larger application.

Paths are non-cyclic sequences of basic blocks (Aho, Sethi, Ullman, 1986, Holub, 1990). To illustrate, consider the following program which scales an array by a constant then transfers control to another piece of code:

```
pre-amble:   LD n, R1
             LD x, R2
             LD base, R3
loop:        ADD R3, R1, R4    ;; add n + base to get index
             LD R4, R5         ;; load from that address to R5
             MUL R2, R5, R6    ;; scale [index]
             ST R6, R4         ;; store the scaled value
             SUBC R1, 1
             BNE R1, loop
post-amble:  JMP othercode
```

This code segment has three basic blocks and three paths. Pre-amble, loop and post-amble are the basic blocks. Figuring out what the paths are is a little more complicated. When this code is executed for the first time, the PC will start at the first instruction of pre-amble. Control flow will continue to the BNE at the end of loop, and will then return to the start of loop until n has been decremented to zero, when it will then fall through to post-amble. If n were 4 the sequence would be pre-amble, loop, loop, loop, loop, post-amble. This reveals the three acyclic sequences of basic blocks: pre-amble loop, loop, and loop post-amble.

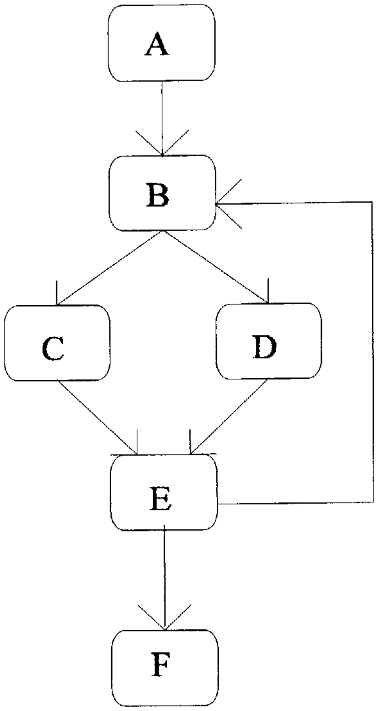
Path behavior in real programs can be much more complicated. A great deal of research has gone into mechanisms for recording path traces efficiently. (Bala, 1996; Ball, Larus, 1996,1994). Doing so with minimum intrusion is a difficult, although now mostly solved (Goldberg, 1991; Malony, Reed, 1992, Ponder, Fateman, 1988) problem.

The relevant piece of information for this thesis from path research is that path usage is very concentrated. (Carroll, 1997; Ball, Larus 1996) A small fraction of all paths in a program constitute the majority of runtime for most programs.

Paths ought to correspond very well with micro-architecture behavior. This is because paths contain a large block of instructions presented in the same order, with the only possible difference being in instruction arguments. (Zaky, 1998) Although this may incline testing towards sections of code where branches are predicted very successfully or cache misses are low, it will likely result in examination of the most important parts of the processor's scheduling, that which is frequently exercised.

2.2 Mechanism of PathOMatic

The mechanism of PathOMatic will be discussed in detail, since it is very illustrative of the technique used in this thesis. Consider the example in the figure below, where A, B, C, D, E, and F are basic blocks.



Execution history:

ABCEBCEBCEBDEBDEBDEF

Parses into four paths:

ABCD, BCE, BDE, BDEF

Yielding the vector:

1	2	2	1

Figure 2.1 Path Example

To run PathOMatic, the benchmark is instrumented to record path counts, and then run. (Ammons, et al, 1997; Ball, Larus, 1994, 1996) In the example, an execution history is examined post-hoc to produce path counts, which is slightly deceptive. Typically advanced techniques allow recording of path counts on the fly, as the benchmark is run. This analysis yields a path count vector, a vector containing a count number corresponding to the number of times each respective path is executed in the benchmark.

These path count vectors are dumped periodically, one for each segment of the program, perhaps every 10 million instructions. Once done, we are left with a set of vectors representing the whole program's execution history; a total vector is also computed.

A linear algebra routine greedily selects segments of the whole program and weights them to minimize the least-squares distance of their weighted sum from the total vector. This weighted set of segments is the resultant representative sample produced by the routine. A similar linear algebra routine is described in great detail in Appendix A.

But, as was mentioned in the introduction, paths are not everything. In particular, path selection schemes are inclined to choose the ordinary. That is, they will select common sequences of code at the expensive of less common ones. For certain performance parameters, like the average time a divide instruction takes to execute, this is fine, but for some other measures, like instruction cache performance, it is precisely the uncommon segments that determine performance. Paths ignore memory systems, other than through implicit inclusion. There is no notion of "similar" paths. Paths are either exactly the same or completely different. Broadly speaking, path statistics simply do not encompass a sufficient range of information about the behavior of a program.

2.3 Extending the Idea to the System.

Path counts are an insufficient means for selecting important segments. They leave too many aspects of the system unconsidered. This is recognized by most architectural researchers who have established alternative techniques as the norm. (Zaky, 1998)

These techniques typically include exploiting randomness or intelligent guessing. Semi-Random selection is analyzed in this work. Semi-Random selection is the author's attempt at achieving an unbiased, broad selection of segments of the program. It is not truly random, as there would be significant statistical work involved in making it so. The justification for randomness is that it seems more robust than other methods, and perhaps will be better in providing broad event coverage.

Intelligent guessing is exactly what it sounds like. Architects look closely at the performance patterns in an application and find what they think are the bottlenecks or important passages. These are selected to drive experiments. Often intelligent guessing is incorporated in other schemes; for example, an architect may specify that a particular segment be included in a set of segments selected by another scheme like PathOMatic or random selection. It has been suggested (Zaky, 1998) that these "guessed" segments be included in a weighting scheme along with segments selected by a more computational scheme, although the author does not know of any actual implementations of such ideas.

2.4 The Limitations of Prior Art

There is a limitation in these techniques, and perhaps in any one technique: they fail to exploit much of the information that can be obtained in a fast instrumented run of a benchmark. A full trace dump provides a wealth of information that can be consumed on the fly with relatively low storage and computational costs. (Agarwal et al, 1988; Borg at all, 1990, Eggers et al, 1990; Goldschmidt, Hennessey, 1992; Stunkel, Fuchs, 1994) This thesis explores some of the possible ideas for using more of the data one can capture.

For example, it will consider simple analysis of the address stream to try to predict cache performance, or track basic blocks instead of paths, or broad classes of basic blocks to allow a less precise definition of "similar". Although these schemes represent only a tiny fraction of the possible ideas, it is hoped that they will serve to represent several broad classes of approach and will spur future research.

3.0 Mechanisms

3.1 Analysis Techniques

The fundamental assumption of these analysis techniques is that a segment of a program that is interesting in one respect, for example by having a representative sample of basic blocks, is interesting in other respects. This is exactly what the thesis experiments aim to test. It is hoped that this will prove true. If it is true, then these correlating behaviors, which are themselves much easier to find than the behaviors they correlate with, can lead us to the important segments. We need to test if cache performance is reflected in simple analysis of the address stream, and so on.

3.2 Basic Block

Basic block counts are the most natural means of selecting program segments. Segments are selected if they, or a weighted sum of them have a similar distribution of basic blocks as the program as a whole.

This makes intuitive sense; if a program is executing the same set of instructions in the same proportion in one section as in another, one would think it must be at least doing something similar in character.

For example executing a loop that strides through an array will produce a similar basic block count each time it is executed. Note, however, that it may not have very similar performance characteristics. Consider the following two passages of code:

```
j=2;
for(i=0;i+=j;i<j*1000) {
      A[i] = A[i-1] + 1;
}
```

and the very similar code

```
j=17;
for(i=0;i+=j;i<j*1000) {
      A[i] = A[i-1] + 1;
}
```

Both pieces of code will compile identically, except for a constant, and hence produce virtually identical basic block counts when run. (Ullman et al, 1986) But the second passage of code will be considerably slower on most modern machines as its awkward array stride will cause repeated cache misses. (Patterson, Hennessey, 1996)

This example illustrates the first problem with using basic block counts to characterize performance: individual basic blocks can take a variable amount of time or resources to execute, depending on data that may vary each time a basic block is executed

The second problem we encounter with basic block counts is that they do not tell us anything directly about control flow. The sequences of basic blocks A, B, C, and D: ABCDDCBAACDBBADC and AAAABBBBCCCCDDDD have the same basic block counts, but they have very different runtime behaviors. On a modern microprocessor the first sequence will produce a dramatically different sequence of micro-architectural events from the second sequence. In the second, structured case, most branches will be predicted and the pipeline kept bubble-free. But in the first case, all bets are off. If you were trying to validate control logic, the first sequence may be a better choice since it exercises a greater variety of paths, but basic block count gives you no indication of this.

3.3 Instruction Class

Instruction class, a generalization of the basic block method was investigated in some depth. Basic Block relies on blocks being the same if and only if they are exactly the same piece of code

(Actually a coding amalgam that approximates this very closely was used to avoid needing a huge vector to accommodate all basic blocks, when most were never used.). The Instruction Class scheme weakens this definition somewhat in an attempt to lessen the capture of a great number of very similar basic blocks, but instead to capture a great number of different types of basic blocks.

Instead of creating a vector with one entry per basic block in the benchmark, a vector is created with one entry for each class of basic block that exists in the benchmark. For example, if we wished to classify basic blocks according to how many DIVC instructions (Kane, Heinrich, 1992) they contained, and the greatest number of DIVC instructions in any basic block was nine, we could use a vector with ten entries, one to count each of the possible number of DIVCs, from 0 to 9 for each basic block executed.

Three such schemes were evaluated in this thesis.

Scheme 1 sorted basic blocks based on the numbers of instructions they had in two categories, memory operations of all sorts, and arithmetic operations of all sorts (that is, instructions that are neither memory operations nor control flow operations). This was intended to capture work versus memory balance of the code.

Scheme 2 sorted basic blocks based on the number of LW instructions, SW instructions, MUL instructions and DIV instructions. This was intended to capture the expensive operations of the code.

Scheme 3 sorted basic blocks based on the number of LW instructions and the number of SW instructions. This scheme was intended to measure memory performance.

It is easy to envision a large variety of similar schemes. These were chosen for their representativeness of broad classes of ideas.

3.4 Pseudo-Cache

Pseudo cache schemes, a third major category, aim to select segments if they have representative memory access patterns. This is done by analyzing the address trace yielded by Mixie (Mixie is a MIPS-based trace generator) (Killian, 1997; Agarwal et al, 1988; Lebeck, Wood, 1994; Stunkel, Fuchs, 1989) in a simple fashion emulating a trivial cache. This technique is expected to be a significant win when trying to examine cache or memory system performance, because it can distinguish between simple and complex memory requests.

Clearly it is too difficult to run a full cache simulator dynamically as we generate a trace. Consider a trace of 100 billion instructions. If each instruction takes 1000 instructions to process in a cache simulator (this is not an atypical number for a complex simulator; typical slowdowns range from a factor of 100 to 10,000, generally depending on the complexity, and hence accuracy, of the processor model) then we must execute 100 trillion instructions to get good data. That is about a million seconds, 277 hours, or 12 days of execution. For every single cache model you want to test, every time you want to run that test. That is prohibitively costly.

There may be a cheaper alternative to keeping track of full cache behavior all the time. Perhaps one could keep track of only a small portion of that information, and still assess what sections of the program are interesting in terms of it?

This leads to the pseudo-cache methods of selecting salient segments of benchmarks. On the fly the information contained in the address stream is decimated and processed. The processing is simple and similar in fashion to a trivial cache. An attempt is made to conserve some important information in a way that is amenable to our segment selection mathematical scheme.

Pseudo-Cache algorithms as implemented in this thesis maintain a vector with n entries, where n is 4096, 512 or 64. Each address that is loaded from or stored to increments the xth entry of the vector where x is the address modulo 4096, or 512 or 64, depending on the model.

It seems clear that their is a great deal of room for improvement in pseudo-cache decimation procedures. This thesis can only begin to explore the design space, and the author hopes future work will be done here.

The preceding were techniques expected to provide superior performance. What follows is a description of what is usually done.

3.5 Other Methods

Random selection was already introduced in chapter 2. Segments of the program are selected at random and used to drive a simulator. This is used far more frequently then one would expect, because it ought not to have any obvious bias. Its accuracy may be improved by incorporating it into other more complicated selection schemes.

Often a person examines the program and makes a best-guess about what the important regions are for analysis. Thus technique is sometimes coupled with others: a known important segment is forcibly included in a set of segments determined by some more sophisticated analysis technique.

3.6 Technique Characterization

All these techniques seek programs with representative statistics. But what is representative?

In the Basic Block technique we sought a selection of program segments such that some weighted sum of those segments had a similar distribution of basic blocks as the program as a whole. For that technique "representative" was taken to mean "proportional basic block count". That makes intuitive sense, and for certain applications is the right thing to do.

Instruction class schemes seek to do the same thing with the twist of a relaxed definition of “representative”. Segments are considered representative if they have proportional counts of types of basic blocks.

Pseudo-cache techniques seek to select segments with representative address streams, that will hopefully reflect representative cache performance.

Varying the definition of “representative” lets us examine programs for other important properties, which in turn helps us obtain sets of segments suitable for other types of experiments. This definition of “representative” is the most important independent variable in these experiments.

4.0 Tools and Experiments

4.1 Experimental Procedure

The software package developed for this thesis can be divided into three sections. First, we have a data production phase. Mixie, a Silicon Graphics trace-generation tool produces traces as a stream to stdio. (Killian, 1997) Mixie could be replaced by another primary data production tool. For example, a trace generator for another type of machine could be substituted. In the case of PathOMatic, Tinst (Carroll, 1997 a path count tool generates the initial stream of data. There are several similar pieces of software (Ball, Larus, 1996).

Second, these traces, which are too large to store practically, are consumed on the fly by the next software phase. These intermediate programs function to decimate the data stream, retaining the information deemed important. Simple, fast processing on the trace stream records salient information efficiently. In ordinary trace driven simulation, a simulator, for example a cache simulator, would go here. Instead, this project extracts the information to be used to compress the benchmark and form it into data files to serve as a basis for the next stage.

In the third phase, the data files generated by the organizers are used as input to a linear algebra routine written in the Matlab language (MIT I/S, 1994). The Matlab program processes the files and selects the important segments of the benchmarks, and then calculates their weightings. See Appendix A for a full explanation of this program.

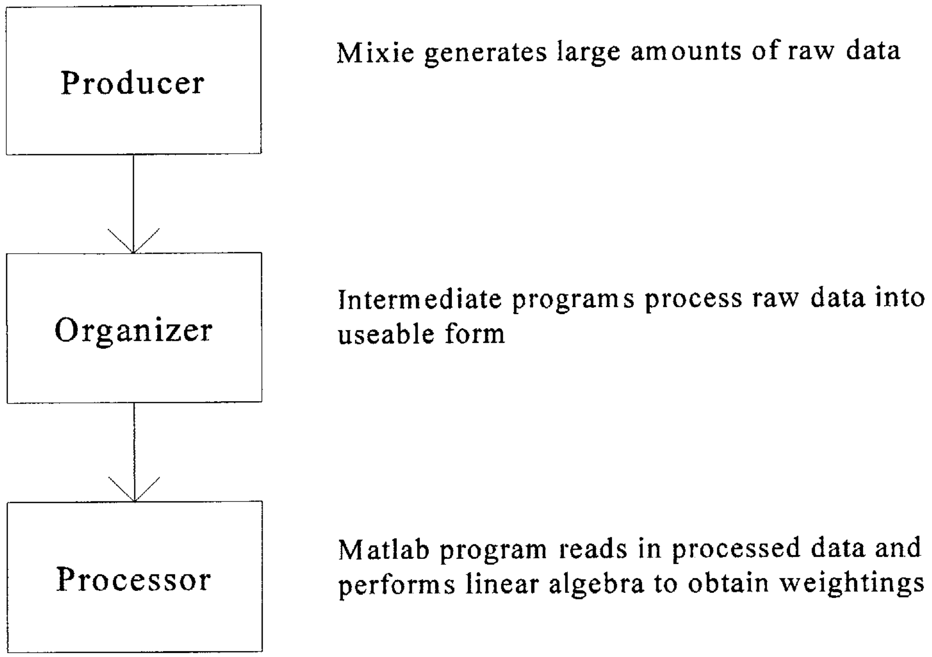

Producer
Mixie generates large amounts of raw data
Organizer
Intermediate programs process raw data into useable form
Processor
Matlab program reads in processed data and performs linear algebra to obtain weightings

Figure 4.1 Software Architecture

4.2 Mixie

In the first stage we have a data-producing program like Mixie, a trace generation tool, or Tinst, a program analysis tool, that generates a large amount of raw data dynamically during the run of a benchmarks program. Mixie, for example can dump detailed information about the instructions being executed as they go by: the program counter they correspond to, their arguments, and memory address they access, etc. This information could be used to drive simulators, were it not for the tremendous volume of data generated.

The traces outputted by Mixie come as a stream that can be redirected to stdio. The data includes for each instruction, the opcode, the register arguments, the PC and any address read from or written to.

Mixie is a trace generation tool specifically for Silicon Graphics computers. Mixie is derived from Pixie, and before that Moxie (Killian, 1997). While Mixie is a reasonably fully functioned trace generation tool capable of dealing with MIPS IV and dynamically shared libraries, it stems from simpler tools that were used, among other things to translate instructions sets to allow compiler development before the availability of real hardware. Mixie is similar to other tools written for Cray (Williams et al, 1990; Gao, Larson, 1995; Cray Research Inc., 1994), Sun (Singhal, Goldberg, 1994), HP (Hunt, 1995), DEC (Digital Equipment Corporation, 1995) and IBM (Maki, 1995; Welbon et al, 1995) hardware.

4.3 Decimation

To cut down on this volume of data we feed it into the next stage of our system, which serves as an organizer of data. On the fly, it processes data and does a great deal of decimation. The organizer implicitly determines what criteria we will use to select salient segments, because it throws away data that we consider less important. Hence we choose the characteristic we wish to use for compression and the organizer saves that data to characterize the program as it goes. Typically it receives data through a pipe from the producer and dumps files periodically during the run of the program. These files will be post-processed to decide on the important parts of the benchmark.

Eight programs were written to perform this decimation, and they fall into four basic categories, Basic Block, Pseudo-Cache, Instruction Class, and Semi-Random.

The first, Basic Block, counts the number of times each basic block is executed in the run of a benchmark. It does this by keeping track of the PCs that start each block. Basic Block can only record 4096 different basic blocks for a program run, but that is more than enough to cover all but a minuscule fraction of the basic blocks executed.

The second, Pseudo-Cache, captures the address stream. Each address N is entered into the N modulo Xth entry of the recording vector, where X is either 4096, 512 or 64. Hence, a representative vector would be one that accesses similar addresses in similar proportions to the whole program.

The third, Instruction Class, classifies the basic blocks as they are executed, recording the numbers with each mix of instructions. Scheme 1 records blocks according to the number of memory operations and the number of arithmetic operations. Two different blocks, each with the same number of instructions of each type, say 2 and 7, would map to the same element of the recording vector. The vector is large enough to accommodate all basic block instruction mixes in the benchmarks. Scheme 2 records blocks based on the number of LW, SW, MUL and DIV instructions. Scheme 3 records blocks based on the number of LW and SW instructions.

The fourth scheme, Semi-Random takes the 5th, 15th, 25th, 35th, and 45th segments of a benchmark (each benchmark was constrained to be 50 segments long, each of 10 million instructions) and weights them equally by a factor of ten.

Finally, the results of this decimation, a set of files each containing a vector, and a file containing a total vector, are analyzed to produce weighted sets of segments with sums approximating the total of each measured attribute. This algorithm is discussed in great deal in Appendix A.

4.4 Analysis

This analysis is done by a processor program and may be done after the completion of the run of the benchmark program. The segment characterization files are cued and batch processed by a Matlab program that reads in the preprocessed trace data and performs a linear algebraic analysis on it. This analysis leads to a weighted set of benchmark segments which can be used to provide (hopefully) statistically representative experiments for your benchmark. This analysis is fully explained in Appendix A.

Matlab was chosen for the linear algebra analysis because it is easy to program complicated mathematics in. This section of code is not performance limited. Analysis of even large amounts of data can be performed in a few minutes, so no great attempt was made to optimize performance at the expense of coding time or maintainability. Most of the computationally intensive part of the routine are Matlab library (MIT I/S, 1994) code that is implemented in a fairly optimal fashion. Optimization was not seen as crucial to the results of this thesis.

4.5 Verification

These weighted sets of benchmark segments were compared with the whole program in terms of their total graduated loads, graduated stores, TLB misses, mispredicted branches, primary data cache misses, secondary data cache misses, decoded branches, primary instruction cache misses and secondary instruction cache misses. These experiments were run on the Silicon Graphics Octane system with the R10000 microprocessor.

Extensive use was made of Silicon Graphics' performance analysis software and the hardware facilities (SGI, 1998) offered by the R10000 microprocessor (Zagha, 1996). The R10000's hardware performance counters were used to obtain performance statistics from actual runs of benchmark software on real workstation systems. (Ammons et al, 1997; Goldberg, 1991; Maloney et al, 1992) This method was needed to provide a great deal of real data for a variety of performance measures on a variety of programs, across the whole length of the program runs. This was provided by running the benchmarks on workstations using the performance monitoring utilities provided by the R10000's hardware counters and the interface software packages EVF, evrate and perfex (SGI, 1998).

These tools allow relatively non intrusive reading of the microprocessors hardware counters, (Goldberg, 1991; Maloney et al, 1992) dumping counts every 10 million instructions (a tunable number) via a relatively small software routine. There is an incentive to lower the frequency of these recordings, because the monitoring program itself interferes with the accuracy of the reported statistics. Every 10 million instructions is sufficiently infrequent so as not to effect the numbers noticeably.

Finally, the weightings can be used to compute the predicted number of events of a particular type, for example L2 cache misses caused by instruction access, and this can be compared with the real number. This, when done across a variety of measures, gives some indication as to the accuracy of the weighted selections representation of the behavior of the whole program.

5.0 Results

5.1 Result Format

This chapter gives the segments selected and the weights computed for each benchmark. Note that negative weights are possible; it is possible for the best weighting to include the negative of a particular segment's count. While this may seem counterintuitive, possibly justifying an effort to make a computer perform slower on a segment to increase measures, this is not so. Negative weights sometimes appear to eliminate overcounting of one particular aspect of a segment, for example. The negative weight cancels the overemphasis brought by another segment.

The weighted sets of benchmark segments were compared with the whole program in terms of their total graduated loads, graduated stores, TLB misses, mispredicted branches, primary data cache misses, secondary data cache misses, decoded branches, primary instruction cache misses and secondary instruction cache misses. These table appear in full in Appendix B, and are summarized later in this chapter, and in chapter 6.

Graphs illustrating captured behavior for each program and each statistic recorded are included in full in Appendix C. These graphs reveal the nature of patterns in each benchmark. They are somewhat useful in understanding why a particular mix of sections was picked. Typically this because a program has two distinct phases; in a first approximation, one is chosen and weighted highly. With lower error bounds, both are chosen and the weights are more moderate.

Semi-Random is not given its own table because it always chooses the same thing, an even distribution of five segments through the program, equally weighted.

5.2 A Word on Error Bounds

For each benchmark/selection scheme, three weighted sets are given. These are the results of computing sets that have vector sums less than 0.1 total vector lengths error in prediction, 0.01

and 0.001 respectively for the selection scheme. That is, for example, the weighted sum of the basic block vectors selected with an 0.01 error bound is less than 0.01 times the length of the total basic block vector different from the total basic block vector. It is 99% representative in that metric.

5.3 The Benchmarks

Extensive data on the behavior of our three benchmarks is included in Appendix C, however, the author feels it is important to include some observations here.

Four benchmarks were used in this thesis. They include a LISP benchmark based on Li from SPECINT (SPEC, 1999), an mpeg player, instructions 0 to 500 million of tomcatv from SPECFP, and instructions 2 billion to 2.5 billion. (SPEC, 1999) These benchmarks were chosen because they presented a broad set of program types, and they fit within the size constraints of the software package. While there are no fundamental limits, performance or otherwise, on the size of programs to be analyzed, particularly because the analysis need only be done once for as many experiments as one wants on the shrunken benchmark, there were constraints in the software developed that prohibited tests of more than around a billion instructions. These benchmarks are all under 600 million instructions. Tomcatv is 24.5 billion instructions long, so subsections were analyzed.

The LISP benchmark consist of an abbreviated version of Li, consisting of the code from SPECint95, but with only a small subset of the inputs. Tomcatv is the full benchmark from SPECfp95 on its full input. The Mpeg player is 12 seconds of an mpeg of a moving bicycle, with the -no_display option selected.

5.4 Weightings

Table of segments selected and weightings for a given compression scheme and error budget for Lisp benchmark:

	0.1	0.01	0.001
Basic Block	12(5.5788), 48(34.3728)	4(8.5983), 12(5.5788), 48(34.3728)	1(0.9964), 2(1.0083), 4(2.4146), 6(3.6805), 8(1.9045), 12(5.5747), 18(13.3192), 48(21.0693)
Address Modulo 4096	17(49.4798)	14(14.6960), 17(35.4397)	1(1.0389), 4(3.4727), 12(4.7684), 14(5.9751), 17(1.4451), 35(33.2630)
Address Modulo 512	17(49.4356)	7(11.3523), 17(38.7080)	1(0.9477), 5(3.0267), 7(5.6716), 15(6.1012), 17(2.2302), 23(16.2251), 46(15.7867)
Address Modulo 64	17(49.3189)	7(11.7796), 17(38.3425)	7(10.5435), 12(4.2397), 17(7.5260), 23(27.7148)
Instruction Class Scheme 1	17(49.5815)	15(12.5406), 17(37.8017)	15(15.0930), 17(8.6279), 34(26.2947)
Instruction Class Scheme 2	17(49.9211)	8(14.6792), 17(35.5491)	8(14.6792), 17(35.5491)
Instruction Class Scheme 3	17(49.9178)	8(14.6667), 17(35.5593)	8(14.6667), 17(35.5593)

Table of segments selected and weightings for a given compression scheme and error budget for tomcatv (first 500 million instructions) benchmark:

	0.1	0.01	0.001
Basic Block	14(7.1012), 35(33.1822)	1(1.6672), 7(8.3217), 14(6.8338), 25(33.1709)	1(1.6672), 7(8.3217), 14(6.8338), 25(33.1709)
Address Modulo 4096	26(50.0076)	26(50.0076)	26(50.0076)
Address Modulo 512	26(50.0077)	26(50.0077)	26(50.0077)
Address Modulo 64	26(50.0025)	26(50.0025)	26(50.0025)
Instruction Class Scheme 1	22(50.0196)	22(50.0196)	22(50.0196)
Instruction Class Scheme 2	22(50.0087)	22(50.0087)	22(50.0087)
Instruction Class Scheme 3	22(50.0077)	22(50.0077)	22(50.0077)

Table of Segments selected and weightings for a given compression scheme and error budget for tomcatv (instructions 2 billion to 2.5 billion) benchmark:

	0.1	0.01	0.001
Basic Block	3(30.4814)	3(30.4814)	3(30.4814)
Address Modulo 4096	37(34.8035), 48(14.0535)	37(22.2379), 38(11.7727), 48(15.9320)	36(15.4424), 37(16.2024), 38(17.7953), 48(0.5524)
Address Modulo 512	28(33.4004), 48(15.7537)	11(2.7398), 28(31.2388), 48(15.8679)	6(4.7925), 11(11.5508), 28(22.1498), 48(11.4289)
Address Modulo 64	25(34.0127), 45(15.7850)	25(34.0127), 45(15.7850)	8(5.1521), 24(4.2497), 25(28.8802), 45(11.6175)
Instruction Class Scheme 1	22(33.9384), 39(16.0984)	5(11.6970), 6(6.0650), 19(5.0670), 22(17.5144), 39(9.6468)	5(15.3472), 6(15.3114), 7(14.3326), 19(0.3599), 22(1.2207), 39(0.6865), 50(2.7472)
Instruction Class Scheme 2	N/A	N/A	N/A
Instruction Class Scheme 3	28(45.6033)	28(45.6033)	28(45.6033)

Table of segments selected and weightings for a given compression scheme and error budget for mpeg_play benchmark:

	0.1	0.01	0.001
Basic Block	3(10.1058), 22(38.6334)	3(5.7300), 4(3.6266), 19(6.7646), 22(13.5163), 25(7.6615), 28(6.0498)	1(1.0120), 2(0.9840), 3(1.0427), 4(1.3640), 5(1.6789), 7(1.0148), 8(1.4022), 9(1.5068), 11(1.2600), 12(3.9195), 15(1.1851), 16(1.5726), 19(0.7521), 22(6.9909), 25(9.7726), 28(5.5172), 33(4.4573), 45(-1.1899)
Address Modulo 4096	28(44.4628)	28(44.4628)	5(4.7261), 16(15.7733), 28(5.4026), 33(9.6037), 41(8.9378)
Address Modulo 512	28(44.4657)	28(44.4657)	2(5.9297), 12(3.6829), 16(7.5369), 21(4.7675), 28(5.8292), 33(10.5598), 43(6.4010)

Address Modulo 64	28(44.3648)	28(44.3648)	2(5.7028), 12(3.0732), 16(4.3611), 17(5.7035), 21(5.4657), 28(3.3246), 33(6.6467), 34(5.1466), 43(5.3588)
Instruction Class Scheme 1	30(45.0531)	23(13.9065), 30(12.0187), 37(19.2627)	2(3.8375), 3(7.3417), 23(10.8037), 28(5.8148), 30(8.0239), 31(2.5373), 37(6.4508)
Instruction Class Scheme 2	20(43.9925)	20(43.9925)	20(30.3809), 35(14.3007)
Instruction Class Scheme 3	25(42.5328)	25(42.5328)	14(13.3603), 25(30.3239)

All measures are relative to graduated instructions. The R10000 performance counters can index against other values, but this was chosen for the sake of consistency and for certain technical reasons.

5.5 Summarized Results

Once the compressed benchmarks were compared against the whole-program figures, the results were averaged and tabulated as follows. This data can be used to evaluate the various compression schemes, and in particular their ability to predict various measures of performance.

Data for Semi-Random selection is included in the 0.001 error level compression chart, since that is what it is most comparable to.

Key:
LDs : Graduated Loads
STs : Graduated Stores
TLB : TLB Misses
BR-X : Miss-predicted Branches
L1D$: Level 1 Data Cache Misses
L2D$: Level 2 Data Cache Misses
BR : Decoded Branches
L1I$: Level 1 Instruction Cache Misses
L2I$: Level 2 Instruction Cache Misses

Scheme versus predicted quantity — average error magnitude, 0.1 error budget

	LDs	STs	TLB	BR-X	L1D$	L2D$	BR	L1I$	L2I$
BB	0.1874	0.1334	0.1523	0.1319	0.1374	0.1699	0.2062	0.1362	0.1795
am4096	0.0258	0.0069	0.4706	0.0956	0.0772	0.1144	0.0144	0.0907	0.5930
am512	0.0334	0.0266	0.4894	0.0759	0.1012	0.1373	0.0082	0.0877	0.5932
am64	0.0314	0.0275	0.5016	0.0774	0.1037	0.1409	0.0065	0.0901	0.5987
IC-1	0.0189	0.0343	0.5087	0.0315	0.0919	0.1208	0.0104	0.0747	0.6092
IC-2	0.0309	0.0151	0.6737	0.0242	0.0792	0.1587	0.0110	0.0602	0.7906
IC-3	0.0548	0.0870	0.6851	0.2552	0.1403	0.2131	0.0667	0.0925	0.6563

Scheme versus predicted quantity — average error magnitude, 0.01 error budget

	LDs	STs	TLB	BR-X	L1D$	L2D$	BR	L1I$	L2I$
BB	0.1025	0.0268	0.0512	0.0602	0.0216	0.0614	0.1017	0.0688	0.1571
am4096	0.0185	0.0104	0.3761	0.0675	0.0662	0.1399	0.0033	0.0527	0.4862
am512	0.0294	0.0215	0.3948	0.0704	0.0792	0.1480	0.0044	0.0652	0.4993
am64	0.0294	0.0262	0.4101	0.0707	0.0833	0.1576	0.0033	0.0665	0.5032
IC-1	0.0306	0.0165	0.4055	0.0315	0.0423	0.1885	0.0112	0.0521	0.5330
IC-2	0.0324	0.0094	0.5837	0.0215	0.0580	0.2339	0.0115	0.0182	0.6757
IC-3	0.0559	0.0827	0.6177	0.2531	0.1244	0.2694	0.0670	0.0610	0.5478

Scheme versus predicted quantity — average error magnitude, 0.001 error budget

	LDs	STs	TLB	BR-X	L1D$	L2D$	BR	L1I$	L2I$
BB	0.0904	0.0196	0.0612	0.0287	0.0123	0.0443	0.0891	0.0612	0.1487
am4096	0.0039	0.0083	0.1753	0.0054	0.0135	0.0403	0.0074	0.0115	0.1095
am512	0.0260	0.0228	0.1047	0.0310	0.0397	0.0609	0.0165	0.0298	0.0970
am64	0.0116	0.0159	0.1949	0.0268	0.0259	0.0571	0.0066	0.0199	0.1520
IC-1	0.0156	0.0022	0.1491	0.0008	0.0353	0.1492	0.0020	0.0191	0.2138
IC-2	0.0120	0.0127	0.5887	0.0189	0.0556	0.1895	0.0115	0.0146	0.6764
IC-3	0.0384	0.0811	0.5951	0.2520	0.1167	0.2731	0.0635	0.0456	0.6042
Random	0.0394	0.0462	0.2099	0.1078	0.0641	0.0854	0.0489	0.0620	0.0971

Average error per measured event for the Lisp benchmark

	0.1 error	0.01 error	0.001 error
Graduated Loads	0.0339	0.0083	0.0023
Graduated Stores	0.0382	0.0058	0.0028
TLB misses	1.2158	0.9093	0.4355
Misspredicted branches	0.0528	0.0247	0.0043
Primary data cache misses	0.1094	0.0180	0.0119
Secondary data cache misses	0.0439	0.1690	0.1129
Decoded branches	0.0385	0.0090	0.0021
Primary instruction cache misses	0.1534	0.0453	0.0189
Secondary instruction cache misses	1.7546	1.3876	0.5369

Average error per measured event for the first half billion instructions of tomcatv

	Level 1	Level 2	Level 3
Graduated Loads	0.0283	0.0006	0.0006
Graduated Stores	0.0283	0.0040	0.0040
TLB misses	0.3740	0.3183	0.3183
Misspredicted branches	0.0295	0.0020	0.0020
Primary data cache misses	0.0879	0.0570	0.0570
Secondary data cache misses	0.1933	0.1567	0.1567
Decoded branches	0.0287	0.0012	0.0012
Primary instruction cache misses	0.0294	0.0016	0.0016
Secondary instruction cache misses	0.2122	0.2238	0.2238

Average error per measured event for instruction 2 to 2.5 billion of tomcatv

	Level 1	Level 2	Level 3
Graduated Loads	0.0817	0.0805	0.0734
Graduated Stores	0.1111	0.1049	0.0808
TLB misses	0.1519	0.1390	0.1038
Misspredicted branches	0.1896	0.1920	0.1902
Primary data cache misses	0.1120	0.1068	0.0797
Secondary data cache misses	0.1511	0.1381	0.1038
Decoded branches	0.1055	0.1018	0.1022
Primary instruction cache misses	0.0939	0.0856	0.0645
Secondary instruction cache misses	0.0966	0.0904	0.1078

Average error per measured event for mpeg_play benchmark

	Level 1	Level 2	Level 3
Graduated Loads	0.0820	0.0924	0.0455
Graduated Stores	0.0250	0.0103	0.0184
TLB misses	0.1731	0.1922	0.1111
Misspredicted branches	0.1376	0.1469	0.0399
Primary data cache misses	0.1131	0.0966	0.0257
Secondary data cache misses	0.2135	0.2075	0.0796
Decoded branches	0.0256	0.0166	0.0198
Primary instruction cache misses	0.0892	0.0678	0.0373
Secondary instruction cache misses	0.1476	0.1586	0.1870

6.0 Conclusion

6.1 Summary

Non-random compression techniques work, although they are not as robust as would be desirable. For example, with a 0.001 error budget (weighted sum vector differs from total vector by 0.001), Basic Block averages less than 6.2% error across all measures. The three Pseudo-Cache schemes average under 4.9% error across all measures. Least successful of all, the Instruction Class schemes average under 15.7% error across all measures. Semi-Random averages under 8.5% error across all measures.

With tight error constraints, Basic Block is the most consistent of the schemes. Instruction Class is highly variable, and is a very poor predictor of TLB misses and secondary cache misses of both types. Pseudo-Caches scheme are fairly consistent, with most of the error contributed by failure to predict TLB misses as well as Basic Block.

It seems that Instruction Class algorithms relaxed the definition of similar basic blocks too much. A look at the decimated data is revealing; the vast majority of basic blocks tend to fall into a small number of categories, having a couple of loads and a couple of stores. There aren't enough categories to differentiate blocks. Basic Block on the other hand spreads its data over a much larger number of vector entries. Pseudo-Cache data is spread wider still, with most vector elements having at least one basic block count towards them. Curiously in the Pseudo-Cache decimated data sets there is clear evidence of array strides and patterns within patterns. Weights of the various strides vary throughout the progression of a program, thereby allowing selection. A more intelligent analysis of this data may lead to a better compression scheme.

It is clear looking at the prediction errors that some things are just hard to predict. TLB misses, for example are predicted poorly by all schemes, averaging 26.7% error at a 0.001 allowed weighted-sum error for non-random schemes. Even Semi-Random averaged almost 21% error.

Similarly Level 2 cache misses caused by instruction misses are hard to predict. To a lesser extent, Level 2 misses caused by data misses are difficult to predict. The rarer the event, the harder it is to select segments containing it in representative proportion.

There is an optimal number of segments to select. In these experiments, analysis selected more than five segments very rarely. Semi-Random chose five segments by definition. Choosing many more than five segments did not result in greatly increased accuracy. However, choosing only one or two segment was not a robust mechanism. Since these benchmarks were all around the same size, 500 million instructions or so, we are unable to devise the precise relationship between benchmark size and number of segments to select, but we can suggest that it be more than one or two and fewer than, say 10% of the program.

A more robust and accurate scheme would incorporate these ideas, using an intelligent selection scheme, perhaps an advanced Pseudo-Cache routine, coupled with a random selector to add breadth to the sample, and feed the result to a weighting algorithm. Such a scheme would have a particular number of segments to select a priori, rather than an error budget, and would select the best segments to fill such a set rather than use a greedy algorithm. The greedy algorithm is unnecessary, as performance is already quite good. Finding an optimal set of five or so segments is not at all impractical computationally. This scheme would both find the tricky code segments, and reliably represent the whole. The weighting scheme would sort out proportions.

6.2 Suggestions for Additional Work

Hybrid schemes incorporating randomness and intelligent selection schemes need to be investigated in detail. Such a scheme ought to be the ultimate refinement of the methods suggested

Similarly, there are differences in the susceptibility of each measure to prediction. General instruction distributions, like number of loads or mispredicted branches, are easy to predict. The

rarer the event, though, the harder it is to predict. Instruction cache misses are harder to predict than data. Secondary cache misses are a great deal harder to predict than primary cache misses, which are fairly easy. TLB misses are very hard to predict.
here.

However, a great deal of work can be done to improve the intelligent selection mechanisms. This includes improving the mechanism of the decimators, considering alternate data sources and fundamentally questioning the goals of such selection.

First, while Basic Block is stable in its present form, much can be done with Pseudo-Cache. Alternative computationally cheap cache models could easily be attempted. With respect to caches there may be an elementary problem with our selection schemes. They all seek normal behavior, when the interesting cache problems are all the result of abnormal behavior, the load of an address never seen, or the access of an array with an odd stride. Selecting "representative" segments may be selecting primarily for cache hits. Some sort of anti-representative metric could be investigated as a possible compression scheme.

Sources of data besides traces need to be considered for compression. For example, it may be interesting to use data generated by the R10000's performance counters as the basis for compression. This could certainly help in the prediction of cache or TLB misses, and it could allow an efficient Pseudo-Cache using the R10000's cache as the model.

Lastly, we need to question the assumptions of this analysis. What are we trying to obtain in our compressed segments? Representative numbers of events? Representative numbers of events weighted by the time it takes to deal with each event (For example, should cache misses count for more than cache hits)? Comprehensive coverage of the space of events? Our methods selected for the first goal. They were applied to the second with some success. The third goal was not addressed, although obviously it could be important for verification purposes, or certain types of performance analysis, such as guaranteeing worst-case performance, as must often be done in real-time systems.

Experimentally, a larger variety of larger benchmarks could be investigated. This would allow the checking of the scaling of the number of segments that need to be selected with the size of the program. Evaluation of the quality of predictions could be performed on a variety of different machines. And a greater variety of events could be checked for their predictability, particularly if more capable hardware was available, as it undoubtably will be in the future

There is a great deal left to be done. While this thesis opened the field to analysis, it uncovered more questions then it answered. The scope of the paper does not allow a thorough explanation of all the ideas. Still, there is a mechanism here that is very useful. Further exploration will turn it into an even more reliable tool for architecture research.

7.0 Work Cited

Agarwal, A., Sites, R., Horowitz, M., "ATUM: A New Technique for Capturing Address Traces Using Microcode," 13th Annual Symposium on Computer Architecture, June, 1988, pages 119-127

Ammons, G., Ball, T., Larus, J. R., "Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling," ACM Conference on Programming Language Design and Implementation 1997, pages 85-96

Bala, V., "Low Overhead Path Profiling," Technical Report, Hewlett-Packard Labs, 1996

Ball, T., Larus, J. R., "Efficient Path Profiling," MICRO-29, December 1996

Ball, T., Larus, J.R., "Optimally Profiling and Tracing Programs," ACM Transactions on Programming Languages and Systems, vol.16, no.4, pages 1319-1360, July 1994

Bartlett, J.F., "SCHEME->C: A Portable Scheme-to-C Compiler," WRL Research Report 89/1, Digital Equipment Western Research Laboratory, 1989

Bedichek, R., "Some Efficient Architecture Simulation Techniques," Winter 1990 Usenix Technical Conference, January 1990

Borg, A., Kessler, R. E., Wall, D. W., "Generation and Analysis of Very Long Address Traces," International Symposium on Computer Architecture, May 1990, pages 270-279

Borg, A., Kessler, R. E., Lazana, G., Wall, D. W., "Long Address Traces from RISC Machines: Generation and Analysis," Digital Equipment Western Research Laboratory, 1989

Carroll, Steve, "Cutting Benchmarks Down to Size," Presentation slides, August 1998

Cmelik, R. F., Keppel, D., "Shade: A Fast Instruction Set Simulator for Execution Profiling," SIGMETRICS, Nashville, TN, 1994

Cray Research Inc., "UNICOS Performance Utilities Reference Manual," Cray Research Publication SR-2040, January 1994

Digital Equipment Corporation, "pfm - The 21064 Performance Counter Pseudo-Device," DEC OSF/1 Manual Pages, 1995

Eggers, S.J., Keppel, D.R., Koldinger, E.J., Levy, H.M., "Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor," SIGMETRICS International Conference on Measuring and Modeling of Computer Systems, May, 1990

Fujimoto, R.M., Campbell, W.B., “Efficient Instruction Level Simulation of Computers,” Transactions of the Society for Computer Simulation, vol.5, no.2, pages 109-124, 1988

Gao, H., Larson, J.L., “Workload Characterization Using the Cray Hardware Performance Monitor,” Journal of Supercomputing, vol.9, pages 391-412, 1995

Goldberg, A. “Reducing Overhead in Counter-Based Execution Profiling,” Technical Report CSL-TR-91-495, Stanford University Computer System Laboratory, October 1991

Goldschmidt, S.R., Hennessy, J.L., “The Accuracy of Trace-Driven Simulations of Multi-Processors,” CSL-TR-92-546, Stanford University Computer Systems Laboratory, September 1992

Gray, J., Ed., “The Benchmark Handbook for Database and Transaction Processing Systems,” Morgan Kaufmann Publishers, 1991

Holub, A.I., “Compiler Design in C,” Prentice-Hall, Englewood Cliffs, NJ, 1990

Hunt, D., “Advanced Performance Features of the 64-bit PA-8000,” COMPCON ‘95, March 1995

Kane, G., Heinrich, J., “MIPS RISC Architecture,” Prentice-Hall, Englewood Cliffs, New Jersey, 1992

Killian, E., Mixie Documentation, Silicon Graphics, 1997

Lebeck, A.R., Wood, D.A., “Cache Profiling and the SPEC Benchmarks: A Case Study,” IEEE Computer, vol.27, no.10, pages 15-26, October 1994

Lewis, P.H., “How Fast Is Your System? Whose Test Are You Using?,” www.nytimes.com, September 10, 1998

Magnusson, P., Werner, B., “Efficient Memory Simulation in SimICS,” 28th Annual Simulation Symposium, Phoenix, April 1995

Maki, J., “POWER-2 Hardware Performance Monitor Tools,” November 1995

Malony, A.D., Reed, D.A., Wijshoff, H.A.G., “Performance Measurement Intrusion and Perturbation Analysis,” IEEE Transactions on Parallel and Distributed Systems, vol.3, no.4, pages 433-450, July 1992

May, C., “MIMIC: A Fast S/370 Simulator,” Proceedings of the ACM SIGPLAN 1987 Symposium on Interpreters and Interpretive Techniques: SIGPLAN Notices, vol.22, no.6, pages

1-13, St. Paul Minnesota, June 1987

MIT I/S, "MATLAB on Athena," Massachusetts Institute of Technology, 1994

Ousterhout, J., "Why Aren't Operating Systems Getting Faster as Fast as Hardware?," Proceedings of the Summer 1990 Usenix Conference, pages 247-256, June 1990

Patterson, D.A., Hennessy, J.L., "Computer Architecture A Quantitative Approach 2nd edition," Morgan Kaufmann Publishers, San Francisco, CA, 1996

Ponder, C, Fateman, R.J., "Inaccuracies in Program Profilers," Software-Practice and Experience, vol.18, pages 459-467, May 1988

Rosenblum, M., Bugnion, E., Herrod, S., Witchel, E., Gupta, A., "The Impact of Architectural Trends on Operating System Performance," SOSP, Colorado, 1995

Rosenblum, M., Herrod, S., Witchel, E., Gupta, A., "Complete Computer System Simulation: The SimOS Approach," IEEE Parallel and Distributed Technology, Fall 1995

Rosenblum, M., Witchel, E., "Embra: Fast and Flexible Machine Simulation, " SIGMETRICS '96, May 1996

Segal, M., Akeley, K., "The Design of the OpenGL Graphics Interface," Silicon Graphics Computer Systems, July 1997

SGI, "R10000 Performance Counters Documentation", Silicon Graphics, 1998

Singhal, A., Goldberg, A.J., "Architectural Support for Performance Tuning: A Case Study on the SPARCcenter 2000," Proceedings of the 21st Annual International Symposium on Computer Architecture, pages 48-59, April 1994

"SPEC Newsletter," Standard Performance Evaluation Corporation

Srivastava, A., Eustace, A., "ATOM: A System for Building Customized Program Analysis Tools," SIGPLAN Notices, June 1994, vol.29, no.6, pages 196-205

Stunkel, C., Fuchs, W., "TRAPEDS: Producing Traces for Multicomputers Via Execution Driven Simulation," International Conference on Measurement and Modeling of Computer Systems, 1989

Ullman, J.D., Sethi, R., Aho, A.V., "Compilers Principles, Techniques, and Tools," Addison-Wesley, Reading MA, 1986

Welbon, E.H., Chan-Nui, C.C., Shippy, D.J., Hicks, D.A., "POWER-2 Performance Monitor," PowerPC and POWER-2: Technical Aspects of the New IBM RISC System/6000, IBM Corporation, SA23-2737, pages 55-63, 1995

Williams, E., Myers, C.T., Koskela, R., "The Characterization of Two Scientific Workloads Using the CRAY X-MP Performance Monitor," Supercomputing '90, pages 142-152, November 1990

Zagha, M., Turner, S., Larson, B., Itzkowitz, M., "Performance Analysis Using the MIPS R10000 Performance Counters," Supercomputing '96 Proceedings, November 1996, Pittsburgh, PA

Zaky, A., Personal communication, October, 1998

Appendix A: Analysis Code and Description

algebra.m

Algerbra.m is the core of the segment selection operation. It is a Matlab routine that takes as input a set of files, each of which contains a vector describing a segment of code. Each element of that vector is a count of the number of occurrences of a particular event, for example the execution of a particular basic block, in that segment. Algebra.m reads in all these files and sums their data. It now has one giant vector counting all the occurrences of every recorded event in the execution of the benchmark. The program then attempts to build up a weighted sum of a small subset of the segment vectors that approximate the total vector to within error bounds.

This is done in a greedy fashion. Algebra.m selects the segment vector that best approximates the total vector, computing the weighting via a least squares routine. If the weighted selected vector approximates with small enough error, the program terminates returning the vector and weight. If it doesn't, it then tries to minimize error by selecting a second vector and running the least squares routine again, and so on, until the accumulated set of segment vectors, when weighted, approximate the total vector to within the error bound.

Note that we do not try to find, say, the best six vectors, as this would be computationally difficult. Our greedy scheme is less computationally intensive and, empirically speaking, has produced very good results. However, it is not completely clear that this optimization is necessary.

```
algebra.m:
n = input ('Enter the number of rows in each matrix:  ');
f = input ('Enter the highest file number:  ');
allowable = input ('Enter the allowable error:  ');

M = zeros (n, f+1);
```

```
for x = 0 : f,
        % set up filename
        name0 = sprintf('%s.%4.4d.%s' ,'test1',x,'dat');

        fid = fopen(name0);

        F = fscanf (fid, '%d');

        for y = 1 : n,
                M(y, x+1) = F(2*y, 1);
        end
        fclose (fid);
end

B = zeros (n, 1);
% set up filename

name0 = sprintf('%s.%s.totals','test1', 'dat');
fid = fopen(name0);

G = fscanf (fid, '%d');
for y = 1 : n,
        B(y, 1) = G(2*y, 1);
end

fclose (fid);

SELECT = zeros (1, f+1);
error = 1;
c = 0;
r = n;
X = zeros (1, f+1);

while (error > allowable)
        for x = 1 : f+1,
                c = c + SELECT(1, x);
        end

        A = zeros (r, c+1);


        acn = 1;
```

```
for x = 1 : f+1,
      if (SELECT(1, x) == 1)
            A(:, acn) = M(:, x);
            acn = acn + 1;
      end
end

BEST = ones (1, f+1);

for x = 1 : f+1,
      if (SELECT(1, x) == 0)
            A(:, acn) = M(:, x);
            X=A\B

            A

            %error calculation
            Temp = A * X;


            Temp

            Temp = Temp - B;

            squaresum = 0;
            for y = 1 : n,
                  Square (y, 1) = (Temp (y, 1))*(Temp (y, 1));
                  squaresum = squaresum + Square (y, 1);
            end
            calcerror = sqrt(squaresum);

            BEST (1, x) = calcerror/(norm(B));


      end
end

lowest = 1;
iol = 0;


for x = 1 : f+1,
        if (BEST(1, x) < lowest)
            lowest = BEST(1, x);
            iol = x;
        end
    end

    error = lowest;
    SELECT(1, iol) = 1;
end

%Produce answer.

for x = 1 : f+1,
    if (SELECT(1,x) == 1)
        A(:, acn) = M(:, x);
    end
end


%Print the selection

SELECT

%Print the weighting

X=A\B
```

Appendix B: Predictive Accuracy Data

Error in prediction of graduated loads for Lisp benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.2006	0.0275	0.0002
Address Modulo 4096	0.0085	-0.0074	0.0005
Address Modulo 512	0.0093	-0.0026	0.0017
Address Modulo 64	0.0117	-0.0038	0.0012
Instruction Class Scheme 1	0.0064	-0.0068	0.0025
Instruction Class Scheme 2	-0.0004	-0.0049	-0.0049
Instruction Class Scheme 3	-0.0003	-0.0049	-0.0049

Error in prediction of graduated loads for tomcatv (first 500 million instructions) benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.1941	-0.0005	-0.0005
Address Modulo 4096	0.0009	0.0009	0.0009
Address Modulo 512	0.0009	0.0009	0.0009
Address Modulo 64	0.0010	0.0010	0.0010
Instruction Class Scheme 1	0.0001	0.0001	0.0001
Instruction Class Scheme 2	0.0004	0.0004	0.0004
Instruction Class Scheme 3	0.0004	0.0004	0.0004

Error in prediction of graduated loads for tomcatv (instructions 2 billion to 2.5 billion) benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.3263	0.3263	0.3263
Address Modulo 4096	0.0083	0.0100	0.0004
Address Modulo 512	0.0376	0.0284	0.0224
Address Modulo 64	0.0298	0.0298	0.0282
Instruction Class Scheme 1	0.0262	0.0260	0.0011
Instruction Class Scheme 2	N/A	N/A	N/A
Instruction Class Scheme 3	0.0622	0.0622	0.0622

Error in prediction of graduated loads for mpeg_play benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	-0.0287	0.0555	0.0346
Address Modulo 4096	-0.0855	-0.0855	-0.0136
Address Modulo 512	-0.0856	-0.0856	-0.0788
Address Modulo 64	-0.0831	-0.0831	-0.0159
Instruction Class Scheme 1	-0.0429	-0.0895	-0.0586
Instruction Class Scheme 2	0.0918	0.0918	0.0307
Instruction Class Scheme 3	0.1561	0.1561	0.0861

Error in prediction of graduated stores for Lisp benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.1753	0.0187	-0.0035
Address Modulo 4096	-0.0127	0.0017	-0.0006
Address Modulo 512	-0.0118	0.0044	0.0018
Address Modulo 64	-0.0094	0.0043	0.0040
Instruction Class Scheme 1	-0.0148	-0.0026	0.0006
Instruction Class Scheme 2	-0.0217	0.0045	0.0045
Instruction Class Scheme 3	-0.0217	0.0045	0.0045

Error in prediction of graduated stores for tomcatv (first 500 million instructions) benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.1943	-0.0005	-0.0005
Address Modulo 4096	0.0009	0.0009	0.0009
Address Modulo 512	0.0009	0.0009	0.0009
Address Modulo 64	0.0010	0.0010	0.0010
Instruction Class Scheme 1	0.0001	0.0001	0.0001
Instruction Class Scheme 2	-0.0002	-0.0002	-0.0002
Instruction Class Scheme 3	0.0004	0.0004	0.0004

Error in prediction of graduated stores for tomcatv (instructions 2 billion to 2.5 billion) benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.0664	0.0664	0.0664
Address Modulo 4096	0.0080	0.0331	0.0013
Address Modulo 512	0.0877	0.0748	0.0414
Address Modulo 64	0.0958	0.0958	0.0528
Instruction Class Scheme 1	0.0897	0.0399	0.0038
Instruction Class Scheme 2	N/A	N/A	N/A
Instruction Class Scheme 3	0.3191	0.3191	0.3191

Error in prediction of graduated stores for mpeg_play benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	-0.0974	0.0216	0.0081
Address Modulo 4096	-0.0058	-0.0058	0.0303
Address Modulo 512	-0.0059	-0.0059	-0.0471
Address Modulo 64	-0.0036	-0.0036	-0.0056
Instruction Class Scheme 1	0.0325	-0.0052	-0.0041
Instruction Class Scheme 2	0.0235	0.0235	0.0334
Instruction Class Scheme 3	0.0066	0.0066	0.0004

Error in prediction of TLB misses for Lisp benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.0733	-0.0177	0.0246
Address Modulo 4096	-1.3999	-1.0040	0.2008
Address Modulo 512	-1.3978	-1.0376	-0.0813
Address Modulo 64	-1.3921	-1.0259	-0.3782
Instruction Class Scheme 1	-1.4048	-0.9773	0.0607
Instruction Class Scheme 2	-1.4213	-1.1514	-1.1514
Instruction Class Scheme 3	-1.4211	-1.1515	-1.1515

Error in prediction of TLB misses for tomcatv (first 500 million instructions) benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.4753	0.0848	0.0848
Address Modulo 4096	-0.2502	-0.2502	-0.2502
Address Modulo 512	-0.2502	-0.2502	-0.2502
Address Modulo 64	-0.2501	-0.2501	-0.2501
Instruction Class Scheme 1	0.4641	0.4641	0.4641
Instruction Class Scheme 2	0.4642	0.4642	0.4642
Instruction Class Scheme 3	0.4642	0.4642	0.4642

Error in prediction of TLB misses for tomcatv (instructions 2 billion to 2.5 billion) benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.0587	0.0587	0.0587
Address Modulo 4096	0.0289	0.0469	-0.0126
Address Modulo 512	0.1062	0.0882	0.0389
Address Modulo 64	0.1637	0.1637	0.0851
Instruction Class Scheme 1	0.1320	0.0546	0.0054
Instruction Class Scheme 2	N/A	N/A	N/A
Instruction Class Scheme 3	0.4218	0.4218	0.4218

Error in prediction of TLB misses for mpeg_play benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	-0.0017	0.0434	0.0767
Address Modulo 4096	-0.2032	-0.2032	-0.0877
Address Modulo 512	-0.2033	-0.2033	-0.0485
Address Modulo 64	-0.2006	-0.2006	-0.0057
Instruction Class Scheme 1	0.0340	0.1260	-0.0660
Instruction Class Scheme 2	-0.1355	-0.1355	-0.1504
Instruction Class Scheme 3	0.4334	0.4334	0.3429

Error in prediction of misspredicted branches for Lisp benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.2241	0.0383	0.0057
Address Modulo 4096	0.0268	-0.0098	0.0005
Address Modulo 512	0.0277	-0.0029	-0.0016
Address Modulo 64	0.0300	-0.0048	-0.0013
Instruction Class Scheme 1	0.0248	-0.0083	-0.0008
Instruction Class Scheme 2	0.0181	-0.0100	-0.0100
Instruction Class Scheme 3	0.0182	-0.0099	-0.0099

Error in prediction of misspredicted branches for tomcatv (first 500 million instructions) benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.1959	0.0030	0.0030
Address Modulo 4096	0.0014	0.0014	0.0014
Address Modulo 512	0.0014	0.0014	0.0014
Address Modulo 64	0.0015	0.0015	0.0015
Instruction Class Scheme 1	0.0020	0.0020	0.0020
Instruction Class Scheme 2	0.0023	0.0023	0.0023
Instruction Class Scheme 3	0.0023	0.0023	0.0023

Error in prediction of misspredicted branches for tomcatv (instructions 2 billion to 2.5 billion) benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.0713	0.0713	0.0713
Address Modulo 4096	0.1033	0.0079	0.0002
Address Modulo 512	0.0236	-0.0266	-0.0730
Address Modulo 64	0.0240	0.0240	-0.0882
Instruction Class Scheme 1	0.0070	-0.1140	-0.0001
Instruction Class Scheme 2	N/A	N/A	N/A
Instruction Class Scheme 3	0.9082	0.9082	0.9082

Error in prediction of misspredicted branches for mpeg_play benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	-0.0363	0.1283	0.0347
Address Modulo 4096	0.2508	0.2508	0.0194
Address Modulo 512	0.2507	0.2507	-0.0481
Address Modulo 64	0.2524	0.2524	0.0163
Instruction Class Scheme 1	-0.0286	-0.0016	0.0291
Instruction Class Scheme 2	-0.0523	-0.0523	-0.0443
Instruction Class Scheme 3	0.0921	0.0921	0.0874

Error in prediction of primary data cache misses for Lisp benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.2384	0.0384	0.0053
Address Modulo 4096	0.0903	0.0125	0.0125
Address Modulo 512	0.0911	0.0148	-0.0144
Address Modulo 64	0.0932	0.0113	-0.0016
Instruction Class Scheme 1	0.0884	-0.0121	-0.0131
Instruction Class Scheme 2	0.0821	-0.0184	-0.0184
Instruction Class Scheme 3	0.0822	-0.0183	-0.0183

Error in prediction of primary data cache misses for tomcatv (first 500 million instructions) benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.2211	-0.0049	-0.0049
Address Modulo 4096	-0.0266	-0.0266	-0.0266
Address Modulo 512	-0.0266	-0.0266	-0.0266
Address Modulo 64	-0.0265	-0.0265	-0.0265
Instruction Class Scheme 1	0.1046	0.1046	0.1046
Instruction Class Scheme 2	0.1048	0.1048	0.1048
Instruction Class Scheme 3	0.1048	0.1048	0.1048

Error in prediction of primary data cache misses for tomcatv (instructions 2 billion to 2.5 billion) benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.0332	0.0332	0.0332
Address Modulo 4096	0.0026	0.0364	-0.0007
Address Modulo 512	0.0977	0.0861	0.0498
Address Modulo 64	0.1086	0.1086	0.0633
Instruction Class Scheme 1	0.1048	0.0513	0.063
Instruction Class Scheme 2	N/A	N/A	N/A
Instruction Class Scheme 3	0.3251	0.3251	0.3251

Error in prediction of primary data cache misses for mpeg_play benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	-0.0569	-0.0099	0.0058
Address Modulo 4096	-0.1892	-0.1892	-0.0143
Address Modulo 512	-0.1893	-0.1893	-0.0679
Address Modulo 64	-0.1866	-0.1866	-0.0122
Instruction Class Scheme 1	0.0696	-0.0013	-0.0172
Instruction Class Scheme 2	0.0507	0.0507	0.0435
Instruction Class Scheme 3	0.0492	0.0492	0.0187

Error in prediction of secondary data cache misses for Lisp benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.2221	0.1045	0.0266
Address Modulo 4096	-0.0116	-0.0998	-0.0072
Address Modulo 512	-0.0107	-0.0719	-0.0072
Address Modulo 64	-0.0083	-0.0750	-0.0099
Instruction Class Scheme 1	-0.0136	-0.3401	-0.2477
Instruction Class Scheme 2	-0.0206	-0.2459	-0.2459
Instruction Class Scheme 3	-0.0205	-0.2457	-0.2457

Error in prediction of secondary data cache misses for tomcatv (first 500 billion instructions) benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.2914	0.0350	0.0350
Address Modulo 4096	0.1266	0.1266	0.1266
Address Modulo 512	0.1266	0.1266	0.1266
Address Modulo 64	0.1267	0.1267	0.1267
Instruction Class Scheme 1	0.2272	0.2272	0.2272
Instruction Class Scheme 2	0.2272	0.2274	0.2274
Instruction Class Scheme 3	0.2274	0.2274	0.2274

Error in prediction of secondary data cache misses for tomcatv (instructions 2 billion to 2.5 billion) benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.0416	0.0416	0.0416
Address Modulo 4096	0.0353	0.0490	0.0028
Address Modulo 512	0.1278	0.1095	0.0610
Address Modulo 64	0.1404	0.1404	0.0759
Instruction Class Scheme 1	0.1259	0.0522	0.0060
Instruction Class Scheme 2	N/A	N/A	N/A
Instruction Class Scheme 3	0.4357	0.4357	0.4357

Error in prediction of secondary data cache misses for mpeg_play benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.1246	0.0646	0.0738
Address Modulo 4096	-0.2840	-0.2840	-0.0245
Address Modulo 512	-0.2841	-0.2841	-0.0487
Address Modulo 64	-0.2881	-0.2881	-0.0154
Instruction Class Scheme 1	-0.1163	-0.1346	-0.1159
Instruction Class Scheme 2	0.2284	0.2284	0.0951
Instruction Class Scheme 3	0.1689	0.1689	0.1836

Error in prediction of decoded branches for Lisp benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.2058	0.0306	0.0016
Address Modulo 4096	0.0132	-0.0065	0.0003
Address Modulo 512	0.0141	-0.0025	0.0001
Address Modulo 64	0.0164	-0.0039	-0.0007
Instruction Class Scheme 1	0.0112	-0.0077	-0.0001
Instruction Class Scheme 2	0.0044	-0.0060	-0.0060
Instruction Class Scheme 3	0.0045	-0.0059	-0.0059

Error in prediction of decoded branches for tomcatv (first 500 million instructions) benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.1934	-0.0011	-0.0011
Address Modulo 4096	-0.0010	-0.0010	-0.0010
Address Modulo 512	-0.0010	-0.0010	-0.0010
Address Modulo 64	-0.0009	-0.0009	-0.0009
Instruction Class Scheme 1	-0.0017	-0.0017	-0.0017
Instruction Class Scheme 2	-0.0014	-0.0014	-0.0014
Instruction Class Scheme 3	-0.0014	-0.0014	-0.0014

Error in prediction of decoded branches for tomcatv (instructions 2 billion to 2.5 billion) benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.3428	0.3428	0.3428
Address Modulo 4096	0.0379	-0.0002	0.0001
Address Modulo 512	0.0122	-0.0086	-0.0165
Address Modulo 64	0.0007	0.0007	-0.0198
Instruction Class Scheme 1	-0.0056	-0.0247	-0.0004
Instruction Class Scheme 2	N/A	N/A	N/A
Instruction Class Scheme 3	0.2336	0.2336	0.2336

Error in prediction of decoded branches for mpeg_play benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	-0.0829	0.0321	0.0110
Address Modulo 4096	0.0056	0.0056	0.0282
Address Modulo 512	0.0055	0.0055	-0.0484
Address Modulo 64	0.0078	0.0078	-0.0050
Instruction Class Scheme 1	0.0231	-0.0106	-0.0059
Instruction Class Scheme 2	0.0272	0.0272	0.0270
Instruction Class Scheme 3	0.0271	0.0271	0.0129

Error in prediction of primary instruction cache misses for Lisp benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.1201	0.0003	-0.0101
Address Modulo 4096	-0.1559	-0.0503	-0.0139
Address Modulo 512	-0.1549	-0.0606	-0.0004
Address Modulo 64	-0.1522	-0.0579	0.0040
Instruction Class Scheme 1	-0.1583	-0.0677	-0.0239
Instruction Class Scheme 2	-0.1662	-0.0401	-0.0401
Instruction Class Scheme 3	-0.1662	-0.0401	-0.0401

Error in prediction of primary instruction cache misses for tomcatv (first 500 million instructions) benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.1949	-0.0005	-0.0005
Address Modulo 4096	0.0006	0.0006	0.0006
Address Modulo 512	0.0006	0.0006	0.0006
Address Modulo 64	0.0007	0.0007	0.0007
Instruction Class Scheme 1	0.0028	0.0028	0.0028
Instruction Class Scheme 2	0.0030	0.0030	0.0030
Instruction Class Scheme 3	0.0031	0.0031	0.0031

Error in prediction of primary instruction cache misses for tomcatv (instructions 2 billion to 2.5 billion) benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	-0.2069	-0.2069	-0.2069
Address Modulo 4096	-0.0724	0.0261	-0.0065
Address Modulo 512	0.0613	0.0658	0.0517
Address Modulo 64	0.0763	0.0763	0.0632
Instruction Class Scheme 1	0.1122	0.1041	0.0241
Instruction Class Scheme 2	N/A	N/A	N/A
Instruction Class Scheme 3	0.0343	0.0343	0.0343

Error in prediction of primary instruction cache misses for mpeg_play benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	-0.0228	0.0675	0.0271
Address Modulo 4096	-0.1337	-0.1337	-0.0248
Address Modulo 512	-0.1338	-0.1338	-0.0663
Address Modulo 64	-0.1312	-0.1312	0.0116
Instruction Class Scheme 1	-0.0253	-0.0338	-0.0256
Instruction Class Scheme 2	-0.0114	-0.0114	-0.0008
Instruction Class Scheme 3	0.1665	0.1665	0.1047

Error in prediction of secondary instruction cache misses for Lisp benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.3450	0.2186	0.0794
Address Modulo 4096	-1.9818	-1.5095	0.0182
Address Modulo 512	-1.9791	-1.6022	-0.0747
Address Modulo 64	-1.9721	-1.5903	-0.1525
Instruction Class Scheme 1	-1.9879	-1.6453	-0.2860
Instruction Class Scheme 2	-2.0084	-1.5738	-1.5738
Instruction Class Scheme 3	-2.0082	-1.5740	-1.5740

Error in prediction of secondary instruction cache misses for tomcatv (first 500 million instructions) benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	0.1113	-0.1925	-0.1925
Address Modulo 4096	-0.1721	-0.1721	-0.1721
Address Modulo 512	-0.1721	-0.1721	-0.1721
Address Modulo 64	-0.1720	-0.1720	-0.1720
Instruction Class Scheme 1	0.2859	0.2859	0.2859
Instruction Class Scheme 2	0.2860	0.2860	0.2860
Instruction Class Scheme 3	0.2861	0.2861	0.2861

Error in prediction of secondary instruction cache misses for tomcatv (instructions 2 billion to 2.5 billion) benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	-0.2150	-0.2150	-0.2150
Address Modulo 4096	0.0594	0.1048	0.0850
Address Modulo 512	0.0632	0.0645	0.0094
Address Modulo 64	0.0903	0.0903	0.0777
Instruction Class Scheme 1	0.0856	0.0016	-0.1933
Instruction Class Scheme 2	N/A	N/A	N/A
Instruction Class Scheme 3	0.0663	0.0663	0.0663

Error in prediction of secondary instruction cache misses for mpeg_play benchmark for a given compression scheme and error budget:

	0.1	0.01	0.001
Basic Block	-0.0465	-0.0022	-0.1079
Address Modulo 4096	0.1585	0.1585	0.1625
Address Modulo 512	0.1584	0.1584	0.1318
Address Modulo 64	0.1603	0.1603	0.1571
Instruction Class Scheme 1	0.0773	0.1991	0.0901
Instruction Class Scheme 2	0.1674	0.1674	0.1695
Instruction Class Scheme 3	-0.2646	-0.2646	-0.4904

Semi-Random selection

	Lisp	tomcatv 0-0.5b	tomcatv 2-2.5b	mpeg_play
Loads	0.0019	-0.0002	0.0283	-0.1273
Stores	0.0072	0.0000	-0.0419	-0.1355
TLB misses	0.1093	0.5893	-0.0627	-0.0783
Misspredicted branches	-0.0122	0.0017	-0.3479	-0.0694
Level 1 data cache misses	-0.0393	0.0303	-0.0357	-0.1509
Level 2 data cache misses	-0.0446	0.0047	-0.0643	-0.2280
Branches	-0.0020	-0.0001	-0.0662	-0.1274
Level 1 instruction cache misses	0.0353	0.0004	0.0602	-0.1522
Level 2 instruction cache misses	0.1226	-0.1741	0.0429	-0.0488