Finding and Characterizing New Benchmarks for Computer Architecture Research

A Thesis
In TCC 402

Presented To

The Faculty of the School of the
School of Engineering and Applied Science
University of Virginia

In Partial Fulfillment

Of the Requirements for the Degree

Bachelor of Science in Computer Science

by

Yuriy Zilbergleyt

April 21, 2003

On my honor as a University student, on this assignment I have neither given nor received unauthorized aid as defined by the Honor Guidelines for Papers in TCC Courses.

Signed____

Approved ____________________________________ Date __________
Technical Advisor –– Kevin Skadron

Approved ____________________________________ Date __________
TCC Advisor –– Ingrid H. S. Townsen

Acknowledgements

This thesis would not have been possible without the help of several wonderful individuals. My technical advisor and director of the LAVA Lab, Professor Kevin Skadron of the University of Virginia's Computer Science department, provided me with most of the background research I needed for this thesis and constantly advised me on the many technical problems I encountered during the project. My TCC Advisor, Professor Ingrid Townsend of the University of Virginia's TCC department, helped me write the actual thesis report by setting partial deadlines and critiquing the documents. Karthik Sankaranarayanan, a graduate student in the University of Virginia's Computer Science department helped me get started with the SimpleScalar toolkit and explained to me how to convert cache and branch predictor configurations into command-line parameters for the simulators. Siva Velusamy, also a graduate student in the University of Virginia's Computer Science department, provided me with the default 21264 configuration file which I used as the basis for all my configurations.

Preface

The original objective of this thesis was to convert several different real-world applications into benchmark programs suitable be used by the LAVA lab in conjunction with SimpleScalar. These applications consisted of the Kaffee Java Virtual Machine, the Postgres database software, an MPEG player, and SimpleScalar itself [5]. I was also supposed to modify the established MiBench benchmarking suite so that it could be used on SimpleScalar.

For baseline data, I planned to use the new benchmarks to characterize the behaviors of different combinations of *branch predictor* and *cache* configurations. Since some of these programs are too large to fully simulate in a reasonable amount of time, I intended to use programs based on Basic Block Vector (BBV) techniques developed by Sherwood *et al.* [2] and Memory Reference Reuse Latency (MRRL) techniques developed by Haskins and Skadron [1]. These techniques would have enabled me to simulate in detail only portions of benchmarks, and still end up with a result representative of the entire benchmark's performance.

Unfortunately, I waited too long to get started, and I was only able to make the SimpleScalar benchmarks and a part of the MiBench suite to be simulated under SimpleScalar. There was also not enough time left to experiment with using

Calder’s BBV tool. Because of this, with my technical advisor’s, Professor Skadron’s, advice, I switched the focus of the thesis from finding and characterizing new benchmarks to characterizing the benchmarks I got working under SimpleScalar and analyzing how different cache and branch predictor configurations affected the performance of different benchmarks.

Table of Contents

Section ***Page Number***

Abstract

Computer architects test new microprocessor techniques by executing benchmarks on microprocessor simulators. However, the benchmarks in use are not enough to cover the vast variety of application types that could tax computers in widely different ways. For this thesis, I simulated applications of different types using the SimpleScalar simulator toolkit. I characterized the performance of these applications utilizing multiple cache and branch predictor configurations and analyzed the results.

This research showed that many programs greatly improve performance with a two-way associative data cache over a one-way associative data cache, but with instruction caches the improvement is not so profound. I also learned that GAg branch predictors are worthless when compared to other predictors, and bimodal predictors are the next worst things. GAs, gshare, PAs, and PAg predictors often give pretty much the same results, and their size only seems to matter when it gets really small.

Glossary

Branch predictor A hardware component of a computer. When the location of the next instruction to be executed depends on the result of an instruction that has not been executed yet, the branch predictor makes an educated guess. Since computers can simultaneously execute multiple instructions (depending, among other things, on the types of the instructions) an instruction often goes into execution before the previous instruction has been executed. If the branch predictor's guess is wrong, the microprocessor must "roll back" and discard all instructions it started executing as a result of the guess.

Cache A hardware component of a computer. A cache stores instructions and data that are deemed likely to be accessed in the near future. Caches are much faster than the main memory in which instructions and data usually reside, so accessing the contents of a cache takes less time than accessing the contents of memory. Caches are also far more expensive than main memory, and therefor much smaller. Only extremely tiny programs can completely fit into a cache.

Cache miss The event when the computer tries to access a piece of data or an instruction from the cache and does not find it there. In this case the computer must look in the memory.

Command line A text interface for an operating system where program names and commands must be typed in order to be executed. No clicking.

Command line parameter An input to a program that is typed at the command line and immediately after the program's name (or after another parameter) before pressing "Enter" to start the program.

Compiling The act of translating a program written in a high level programming language like C++ into a series of machine language instructions that can be executed by a computer.

Compiler A program that compiles other programs.

Computer architect A person who designs computer hardware components.

Execution For a program, the act of running the program. For an instruction, the act of taking the steps necessary to carry out the "orders" implicit in the instruction.

Instruction The basic building block of a program. Each instruction of a program is interpreted and executed by a microprocessor.

Makefile A file that details how a program should be compiled. This file is used by the "make" program to automatically compile programs.

Microprocessor An integrated circuit acting as the central processing unit (CPU) of a computer. Analogous to the "brain" of the computer.

Shell A program that interprets certain commands typed at the command line and executes them.

Shell Script Similar to a program but written using commands understood by the chosen *shell* and the working operating system. No compilation necessary.

Simulator For the purposes of this paper, a program that simulates the operation of a computer. Any program that can run on a computer should be able to run on a simulator with the same results, depending on the level of detail of the simulation. Since simulators are implemented in software, they are orders of magnitude slower than the hardware they are simulating.

Suite A collection of software applications grouped together because of some similarity in purpose. Microsoft Office is an example of this.

I. Introduction

This chapter introduces the motivations and objectives of this thesis report. This chapter also includes a literature review and an overview of the contents of subsequent chapters.

I.1 The Need for Benchmarks in Computer Architecture Research

Computer architects are constantly researching new techniques in *microprocessor* design. Instead of spending time and money on implementing the new designs in hardware, computer architects first test these techniques on *simulators*. The Laboratory for Computer Architecture at Virginia (LAVA), for example, tests their designs using a modified version of the SimpleScalar simulator toolkit, which was developed by SimpleScalar LLC [4]. In computer terms, a benchmark is a special program that is used to characterize a computer system's, or, in this case, a microprocessor's, performance in executing the program. The theory is that a benchmark program is representative of real-world applications, so a measure of how well a system performs in the execution of the benchmark is indicative of what the system's performance with actual applications will be like.

Different applications tax computers in different ways, so there is no way that a single benchmark could predict the 0performance of a computer system on all applications. For example, one benchmark might provide reasonable statistics on how well a computer system would run a word processor, but this data could be meaningless when attempting to predict the performance of the system when playing a graphically intensive computer game. There are currently not enough benchmarks to correspond with all the different application types that a computer system might be expected to run.

I.2 Desirable Characteristics of Benchmarks Used With Microprocessor Simulators

All benchmarks must be able to predict with reasonable accuracy how well a computer system would be able to run a certain application. Benchmarks used with microprocessor simulators like SimpleScalar have an additional requirement: they must not take too long to execute. A microprocessor simulator takes far longer to execute a benchmark than the physical processor that is being simulated would. If an otherwise great benchmark takes a year to simulate, then it is useless. Because of this, all benchmarks to be used with simulators must be able to be executed in a reasonable amount of time.

I.3 Project Objectives and Methods

For this thesis I analyzed how different cache and branch predictor configurations affected the performance of different benchmarks. I did all simulations using the SimpleScalar simulator toolkit. The benchmarks I used were from the MiBench benchmarking suite. In addition, I also simulated two of the SimpleScalar simulators, sim-fast and sim-outorder, running other benchmarks.

To automate the simulation process and to compile results from the simulations I wrote a variety of *shell-scripts*. The data was later put into in an Excel spreadsheet so I could analyze it.

I.4 Literature Review

Executing benchmarks on simulators to predict the behavior of theoretical designs is nothing new in computer architecture. Using only one or more representative samples of a benchmark to try to characterize the full benchmark behavior is a more recent trend. Sherwood *et al.* [2] explains that this is because the complexity of processors grows much faster than processing power. Because of this, detailed simulations of microprocessors become slower with each successive advance in computer hardware technology.

Much research has been done by the computer architecture community on methods of picking the right samples of a benchmark to simulate in full detail. In their 1999 paper, Skadron *et al.* simulated the SPEC95 Benchmark Suite using a single 50 million instruction window for each benchmark, with the entire pre-sample period devoted to warm-up. Skadron *et* *al.* chose this window using interval branch-misprediction rates. This paper also relates how many other researchers put the sample window at the very beginning of the benchmark. This would produce erroneous results, as the initial phase of most programs is often very different from the rest of the execution. Three years after this paper, Sherwood *et al.* [2] introduced the method of using Basic Block Vectors and clustering to select multiple sample windows in a benchmark. This technique was tested on the SPEC2000 benchmark suite, and provided better results than those of a single sample. Also in 2002, Haskins and Skadron [1] introduced the technique of using MMRL to calculate the warm-up sections of the benchmark based on chosen samples. This paper also details a number of previously proposed ways to calculate sample and warm-up periods.

In 2001, Guthaus *et al.* [7] released the MiBench benchmarking suite and a technical report describing the suite. This report contains information on the specific benchmarks of the suite, as well as some characterization results which I compared to those obtained by me.

I.5 Overview of the Contents of the Rest of the Report

The rest of this report is arranged as follows:

Chapter 2 details how I went about preparing and characterizing the benchmarks. Chapter 3 presents and discusses the results of the characterizations. Chapter 4 presents my concluding thoughts about the thesis.

II. Procedures

This chapter describes the methods I used in working on this thesis. It also details the various obstacles I encountered during the process.

II.1 *Overview of the Methods and Tools Used in the Thesis*

All compiling and simulations were performed on LAVA lab's computers. Whenever I needed to use these machines, I accessed them remotely from my desktop computer using Van Dyke Technologies' SecureCRT application, which is distributed in the University of Virginia by ITC. The LAVA computers are on the UVA CS computer network, so when logged in to any of these computers I could access files on my CS account.

For simulation, I used the SimpleScalar microprocessor simulator toolkit developed by SimpleScalar LLC [5]. For the actual characterization, I used the sim-outorder simulator from the toolkit. I compiled the SimpleScalar toolkit on LAVA lab's Linux machines and configured it to simulate programs using the Alpha instruction set.

I compiled the benchmarks on Krakatoa, the LAVA lab's Alpha machine, which enabled them to be simulated by the Alpha-configured SimpleScalar simulators. To compile I used the default *Makefiles* which came with the benchmarks. Some of the benchmarks from the MiBench benchmarking suite had problems compiling, and many of those that correctly compiled had problems executing correctly on SimpleScalar. Because I was not able to resolve all of these problems, I only characterized a part of the MiBench suite.

I converted the cache and branch predictor configurations I wanted to use into *command line parameters* for sim-outorder. I then used these to create a different .config file for each configuration and wrote several *shell scripts* to automate the simulation of benchmarks. After I was satisfied that a benchmark could be simulated by sim-outorder without problems, I used these scripts to simulate the benchmark with sim-outorder and the .config files I had prepared earlier.

After each simulation run, sim-outorder outputted the results to a different file. When all the runs were finished, I wrote shell and Perl scripts to parse the files for the statistics that I needed and output them into a form that enabled me to easily copy and paste them into an Excel spreadsheet. I then did my best to graph and analyze the data.

For the larger benchmarks, I planned to use Calder's *SimPoint* software tool to find the samples of the benchmarks that could be simulated instead of simulating the entire benchmark. Afterwards I planned to use Haskins' MRRL (Memory Reference Reuse Latency) tool to calculate the warm-up periods for each sample provided by SimPoint. The sample and warm-up periods would have been used to simulate large benchmarks in a reasonable amount of time. However, due to my lack of knowledge about Makefiles, I was unable to compile SimPoint for SimpleScalar in time to use it for this thesis. Because of this, I was unable to simulate large benchmarks, so all of the benchmarks characterized for this thesis have less than two billion instructions.

II.2 *Compiling the Benchmarks On Krakatoa*

II.2.1 MiBench Compiling Problems

Since I configured the SimpleScalar toolkit to run programs using the Alpha instruction set, I needed the benchmarks to be in Alpha binary format. To do this, I compiled the benchmarks on the LAVA lab's Krakatoa machine, which is an Alpha computer.

The first benchmarks I attempted to compile were the ones from the MiBench suite. Each MiBench benchmark came with instructions on how to compile it. These instructions were usually of the form:

```
Compile Instructions
--------------------
1)  Type "make".  This will create the executables used by the scripts.

Clean Instructions
------------------
1)  Type "make clean".  This will delete post-compile files (i.e. old executables, output,
object files, etc...).
```

Some of the compile instructions required the running of a provided “configure” shell script that tried to discern various pieces of information about the system, in this case Krakatoa, on which the benchmark was to be compiled so this information could be used in the compilation process.

Since MiBench is an established benchmarking suite, I expected to compile these benchmarks without major problems. Unfortunately I was being overly optimistic. In the *sphinx* benchmark, there was a problem with the configure script that I could not figure out how to solve. The benchmarks *lame*, *mad*, *tiff*, and *pgp*, gave various errors which I did not know how to deal with when I tried to compile them using *make*. On the advice of my technical advisor I tried using the GNU version of *make*, found in the /usr/cs/bin directory, instead of the default version. This enabled *lame* to compile, but the problems with the other three benchmarks remained.

The *ispell* benchmark, which gave no errors while compiling, gave an “illegal format hash table” error when executed as a normal application on Krakatoa. It also gave the same error when simulated with the SimpleScalar simulators. Similarly, *rijndael,* produced a “memory fault” during execution on Karakatoa and a “segmentation fault” during simulation. I was not able to figure out why this happened.

II.2.2 Using Static Linking to Resolve Segmentation Faults

After successfully compiling, the *lame, jpeg, rsynth,* and *sha* benchmarks gave “segmentation fault” errors during simulation but not during normal execution on Krakatoa. Again, I could not figure out what was wrong. Later, however, when I compiled the SimpleScalar toolkit as benchmarks on Krakatoa, it had the same problem. I noticed that while giving the “segmentation fault” error when simulated by most SimpleScalar simulators, these benchmarks gave a “bogus opcode” error when simulated using the sim-safe simulator from the toolkit. I searched the SimpleScalar mailing list archive [10] online and found a message by Charles Lefurgy [11] that said the “bogus opcode” problem could be resolved by compiling the benchmarks using “static linking.” In the README file that came with SimpleScalar I found that this could be done by adding the “-static” flag when compiling with the GNU *gcc* compiler, or the “-non_shared” flag when compiling using the DEC *cc* compiler.

The Makefiles of all of the programs giving this bug except for *lame* used *gcc* as the default compiler, and after I modified the Makefiles to include the “-static” flag, the bug was fixed. *Lame*’s Makefile used the *cc* compiler as default for Alpha machines, but the “-non_shared” flag did not get rid of the segmentation fault. Modifying the Makefile to use *gcc* with the "-static" flag instead of *cc* solved the problem.

II.2.3 Required Optimizations

After some of the simulation runs were finished, I found out that all of the benchmarks were supposed to be compiled using level 2 or higher optimizations. Looking through the Makefiles of the benchmarks, I found out that the *lame, rsynth, jpeg* and SimpleScalar benchmarks were compiled with either a too low level of optimization or no optimizations at all. Remedying the problem was a simple matter of adding the "-O2" flag to the Makefiles.

In the case of the SimpleScalar benchmark, adding the -O2 flag resulted in an unforeseen problem: Krakatoa kept running out of virtual memory when trying to compile the benchmark with optimizations. Following advice from Professor Skadron, I modified the Makefile to use the *cc* compiler instead of *gcc*. This meant that I needed to use the "-non_shared" flag for static linking, and I was worried because this had not worked for the *lame* benchmark before. Thankfully my fears were not realized, and SimpleScalar simulators compiled were able to execute SimpleScalar simulators compiled with *cc* on Krakatoa with no problems. Now it was time to check if SimpleScalar simulators properly simulated the compiled benchmarks.

II.3 *Making Sure Benchmarks Are Simulating Correctly*

All SimpleScalar simulators have the option to store simulator output to one file and benchmark output to another file. The simulator output is the information provided by the simulator itself, including statistics about the benchmark, while the benchmark output is whatever output the benchmark would have displayed when run as a normal program. Therefore one way to check if a benchmark was simulated properly is to compare the output of simulated benchmark with the output the benchmark made when executed as a normal application on Krakatoa. For this role I used the *sim-fast* simulator from the SimpleScalar toolkit.

Sim-fast, as its name suggests, is the fastest simulator in the toolkit, but it takes some shortcuts during simulation and does not output many useful statistics. Being so fast, however, makes sim-fast is a good simulator to produce the simulated benchmark outputs I used to check if a benchmark would execute correctly on the far slower *sim-outorder*, the simulator that I used to gather characterization data. After making this check for all benchmarks that were successfully compiled and executed on Krakatoa, there were 8 benchmarks from the MiBench suite with differences between their normal and simulated results: *basicmath, patricia, bitcount, typeset, blowfish, sha, FFT,* and *gsm*.

To try to figure out what the problem was, I used the Unix "diff" command to look at the differences between the outputs. The problem with *bitcount's* output was that it contained the timing data for several different types of operations. Since these operations take a lot longer to execute while simulating, this was not a problem with the simulation, so *bitcount* passed the comparison test. Similarly, the *typeset* output included the date and time that the benchmark was run, so of course it was different. The problems with the other 7 benchmarks were not resolved. The strangest of these was a problem shared by *basicmath* and *patricia*. In the simulated outputs of these benchmarks, some numbers were off by exactly one. For example, a segment of the simulated *basicmath* output that should have read:

```
"Solutions: 1.635838
Solutions: 13.811084
Solutions: -3.947812
Solutions: -8.613092"
```

was instead:

```
"Solutions: 2.635838
Solutions: 14.811084
Solutions: -4.947812
Solutions: -9.613092".
```

One problem that the output comparison method failed to catch was with the *ghostscript* benchmark. *Sim-fast* ran *ghostscript* without any problems, and the simulated output matched the normal output perfectly. However, when I tried to gather characterization information with *sim-outorder*, the simulator outputted the error "fatal: non-speculative fault (2) detected @ 0x120077ae0." I was not able to find any information about this error on the SimpleScalar mailing list archive [10]. This was the only time that a benchmark I worked with was simulated perfectly by *sim-fast* but had problems with *sim-outorder*.

II.4 ***Brief Description of MiBench Benchmarks Characterized in this Project***

I was able to get a total of 11 benchmarks from the MiBench suite to work with *sim-outorder*. All of these were executed with the "large" input sets provided with MiBench. The descriptions that follow quote from the MiBench paper [7].

Bitcount "tests the bit manipulation abilities of a processor by counting the number of bits in an array of integers" using 5 different methods."

Qsort "test sorts a large array of strings into ascending order using the well known quick sort algorithm."

Susan "is an image recognition package" which can recognize corners and edges in an MRI scan as well as smooth an image. *Susan* consists of three parts which I treated as separate benchmarks: *susan_corners*, *susan_edges*, and *susan_smoothing*.

Dijkstra “constructs a large graph in an adjacency matrix representation and then calculates the shortest path between every pair of nodes using repeated applications of Dijkstra’s algorithm.”

Jpeg makes use of a “representative algorithm for image compression and decompression.” Jpeg consists of two parts which I treated as separate benchmarks: *jpeg_encode* and *jpeg_decode*.

Lame encodes wave files into MP3 format. *Typeset* is a “general typesetting tool, that has a front-end processor for HTML.”

Stringsearch “searches for given words in phrases using case insensitive comparison algorithm.”

Rsynth is a “text to speech synthesis program that integrates several pieces of public domain code into a single program.”

Adpcm “takes 16-bit linear PCM samples and converts them to 4-bit samples” and vice versa. This benchmark consists of two parts which I treated as separate benchmarks: *adpcm_adpcm2pcm* and *adpcm_pcm2adpcm*.

CRC32 "performs a 32-bit Cyclic Redundancy Check (CRC) on a file. CRC checks are often used to detect errors in data transmission."

II.5 *Choosing How to Simulate the Simulator*

Simulating the SimpleScalar simulators as benchmarks posed a bit of a problem. To make the discussion less confusing, I will make the following definitions:

Inner simulator: The simulator being simulated.

Outer simulator: The simulator simulating the inner simulator

Inner benchmark: The program being simulated by the inner simulator.

Outer benchmark: The simulation of the inner benchmark by the inner simulator.

For example, suppose I wanted to characterize the performance of simulator B by using simulator A to simulate simulator B simulating program C. In this case, A would be the "outer simulator," B would be the "inner simulator," C would be the "inner benchmark," and B simulating C would be the "outer benchmark."

As stated before, it takes far longer to simulate a program than it does to simply run it on compatible hardware. Imagine, then, how much more time an outer simulator requires to simulate an outer benchmark! To illustrate this point, I did an experiment with *sim-fast* as the outer simulator, a tiny program called *test-prinf* which comes with SimpleScalar as the inner benchmark, and *sim-fast* and *sim-outorder* compiled on Krakatoa as inner simulators. **Figure 1** shows the result in a table.

Program Run	Instruction Count	Execution Time (sec)
test-printf	98430	Negligible
Sim-fast simulating test-printf	252,364,984	0.5
Sim-outorder simulating test-printf	3,302,886,821	2.5
Sim-fast simulating sim-fast simulating test-printf	?	41
Sim-fast simulating sim-outorder simulating test-printf	?	571

Figure 1 : Difference in execution times and instruction counts between benchmarks and benchmark simulations

The data in Figure 1 shows that even an inner benchmark that executes almost instantaneously by itself and requires only a couple seconds to be simulated, requires minutes for a fast outer simulator to simulate a slow inner simulator simulating this benchmark. On the same computer, it would probably take upwards of two hours to simulate if the outer simulator was changed from *sim-fast* to *sim-outorder*.

For this reason, when simulating SimpleScalar simulators I needed to pick small inner benchmarks. For *sim-outorder* I picked the smallest of the working benchmarks from the MiBench suite - *stringsearch* with the small input set.

On the suggestion of my technical advisor, I chose *anagram* for *sim-fast*. *Anagram* is a test benchmark which comes with SimpleScalar. It takes a dictionary file and a set of strings as inputs and creates anagrams of those strings based on the words in the dictionary. A sample dictionary file and file of input strings comes with SimpleScalar, but using this made the execution several times longer than I wanted. To remedy this, I wrote a Perl script to erase every second word from the dictionary. When this proved not to be enough, I ran the script a second time, this time on the result of the previous run. This effectively cut the original dictionary by three-quarters. To further cut down simulation time, I used the length parameter of *anagram* to specify that all words in an anagram should be at least 5 characters long. The resultant outer benchmark of sim-fast simulating anagram turned out to be 1.1 billion instructions long, which was about the length I wanted.

II.6 *Simulating and Collecting Data*

II.6.1 <u>Preparing the Configurations</u>

To simulate a benchmark on a SimpleScalar simulator, the simulator must be provided with the configuration to use and the benchmark's path as command line parameters. If configuration parameters aren't supplied, the simulator uses default values. To keep from having to type the same or similar set of configuration parameters every time a simulation is run, SimpleScalar simulators make use of the "-config" and "-dumpconfig" flags. If the "-dumpconfig filepath" parameters are used, the simulator stores all the configuration options in the file specified by filepath. Later, the "-config filepath" parameters can be used to run a simulation with all the configurations stored in the file located at filepath.

The configurations I used for characterizations were all a slight variation on the standard Alpha 21264 configuration used by the LAVA lab. Siva Valusamy, a grad student at the LAVA lab, provided me with a file storing this configuration. This file included some parameters which were not used by sim-outorder, so I had to delete them. Next, with the help of Karthik Sankaranarayanan, I translated the configurations I needed to characterize into command line parameters. For example, the parameter for an "8KB one-way associative level 1 instruction cache with LRU replacement policy" is "-cache:il1 il1:128:64:1:l." Afterwards I used the "-config" and "-dumpconfig" flags together to load the 21264 configuration and store it with modified parameters in a different file for each configuration. There were 38 total configurations, and I stored them in files 00.config through 37.config, after documenting in an Excel spreadsheet which number corresponds to which configuration.

At one point, in the middle of the characterization process, I realized that I forgot about an important parameter from the branch predictor configurations. To fix this I wrote a shell script, *changeconfig.sh* that used the "-config / -dumpconfig" combination to add the missing parameter to the config files for the branch predictor configurations. I then had to re-simulate all simulations which had used the defective config files. A similar problem with a similar solution came up when I realized that all the config files set the maximum number of instructions to execute to one billion. Fortunately, I had not simulated any instructions large enough to be affected by this yet, so nothing had to be re-simulated.

II.6.2 Using EIO Traces

Before simulating a benchmark, I used *sim-eio* from the SimpleScalar toolkit to create an EIO trace file for the benchmark. An EIO trace is useful because it captures the benchmark's execution at the time the trace is made, this includes all command line parameters or outside files besides the executable that the benchmark may need. The EIO file can then be simulated just like the benchmark, but without any additional parameters. For example, in order to simulate the *rsynth* benchmark, the parameters "\rsynth_directory_path\say -

a -q -o large_output.au < largeinput.txt" must be supplied to *sim-outorder*. With an EIO file created using these parameters, the parameters supplied to *sim-outorder* become "\eio_directory\path\rsynth.eio." Once EIO files are made, there is no need to remember the unique parameters any benchmark might require. Since the EIO files do not depend on outside files, they can also be grouped together in one directory for easy access.

One problem I encountered was that, for some unknown reason, sim-outorder had trouble executing EIO traces of SimpleScalar simulators. Whenever I tried, I got a segmentation fault. For this reason, the two SimpleScalar benchmarks, *sim-outorder* and *sim-fast* had to be characterized without being captured as EIO traces. EIO files were still used for the two inner benchmarks, *stringsearch* and *anagram*.

II.6.3 Automating Simulation With Shell Scripts

In all, I ended up with 17 benchmarks. With 38 configurations to use, that makes a total of 646 simulations to run. Because I did not want to have to start every single simulation manually, I decided to write a script to do this task for me. This was made simpler by the fact that all configuration files were grouped in the ~/config directory of my CS network account, and the EIO trace files in the ~/MiBenchEIO directory. I could now start a script on a LAVA computer, make it run in the background and log off. Later I could log back on and check on the progress of the script. Since each of the LAVA computers has two processors, two scripts could run on each machine at any one time without interfering with one another.

The first script I wrote, *simulate.sh,* took an EIO filename as a parameter, and looped through every file in the ~/config directory, using it in conjunction with the EIO filename to simulate the EIO file with each configuration. The simulator and benchmark outputs were stored in the ~/results directory with filenames derived from the names of the EIO and config files. After each simulation, the script would append to the file "finished.dat" a notice indicating that the run has finished. For example, if I typed "simulate.sh CRC32.eio," the script would first tell *sim-outorder* to simulate ~/CRC32.eio using the ~/config/00.config (the first file in the config directory,) output the simulation results to ~/results/CRC32.eio.00.config.sdat and output the benchmark results to ~/results/CRC32.eio.00.config.pdat. After this simulation is finished, the line "CRC32.eio.oo.config finished" would be appended to the end of the file finished.dat.

After using it for a bit, I found that *simulate.sh* had some shortcomings. First, while finished.dat showed me which simulation runs had finished, I could never remember which computer the script was running on. For example, if I saw that the last simulation run of CRC32 was finished, that would have meant that the computer the script was running on only had one script running now, so another one could be added. But I would have to keep logging in to different computers until I found the right one. Perhaps more importantly, *simulate.sh* would simulate using every single one of the 38 configuration files in the ~/config directory every time I invoked the script. Since some benchmarks are much larger, and therefore take far longer to simulate, than others, this meant that when all the small benchmarks are characterized, some computers might still be stuck doing a series of 38 very long simulations while other computers have nothing to do.

To remedy these shortcomings, I wrote another shell script, *simulate2.sh*, and a tiny C++ program, *count*. *Count* took two integers and outputted the integers between them with two digits per number. For example, "count 6 12" would output the line "06 07 08 09 10 11 12." *Simulate2.sh* took, in addition to the EIO file name, two integers and a computer name as command line parameters. *Simulate2.sh* implemented *count* to simulate using only those configuration files which are indicated by the range of the two integers. After a simulation would finish, *simulate2.sh* would output the computer’s number in addition to the simulation information to the file “finished2.dat.” For example, “simulate2.sh CRC32.eio 6 12 010” would start with using the file 06.config, afterwards outputting “CRC32.eio.06 finished on lava010” before going on to configuration file 07.config. The script would stop after simulating CRC32.eio using 12.config and outputting “CRC32.eio.12 finished on lava010” to the file “finished2.dat.” The new script made it possible to split the simulation of the same benchmark across multiple computers.

Since the two SimpleScalar benchmarks did not use EIO trace files, I had to write separate scripts for them. When I found out about the optimization problem (see section II.2.3) I recompiled the benchmark version of SimpleScalar in a different directory, so these scripts needed to be changed. I also had to remake the EIO files for *lame, jpeg_decode, jpeg_encode,* and *rsynth* and re-simulate them. Following Professor Skadron’s advice, I also recompiled the SimpleScalar suite which I used for simulating using the -O2 extension so that simulations would take less time to run. I placed this version of SimpleScalar into another directory, so I changed *simulate2.sh* to use the *sim-outorder* simulator from this directory. I named the new script *simulate3.sh.*

II.6.4 Collecting Simulation Results

After all simulations had finished, I was left with 38 simulation result files for each benchmark. The statistics I was looking for were the average number of instructions per cycle (IPC), level 1 data cache miss rate, level 1 instruction cache miss rate, branch address-prediction rate, and branch direction-prediction rate. It would have taken too much time to manually look for all of these in the result files, so I needed to find a better way.

First, I used the Unix "rename" command to rename the result files so that the filenames would contain only the name of the benchmark and the configuration number. For example, "CRC32.eio.config.00.sdat" became "CRC32.00." Next, I created a different directory for each statistic, and used the *copysdata.sh* script I wrote to copy relevant files into the directories. For example, configurations 22 through 37 deal with branch predictors, so I used *copysdata.*sh to copy the result files for these configurations were copied into the bpred directory.

Next from each of these directories, I used the Unix "grep" command to search the files for lines which included the name of the statistic that directory focuses on, and store these lines into files in the ~/stats directory. For example, the first line in the file "sim_IPC.dat" is "CRC32.00:sim_IPC 3.0284 # instructions per cycle." I then used the Unix "sed" command to change the spaces and colons in these files to commas, so that the number for each statistic would be exactly two commas from the beginning of the line. I then wrote the *organize.ps* perl script to output each file into a format that could be easily copied and pasted into an Excel spreadsheet. After the copying and pasting, all of the data was in Excel tables.

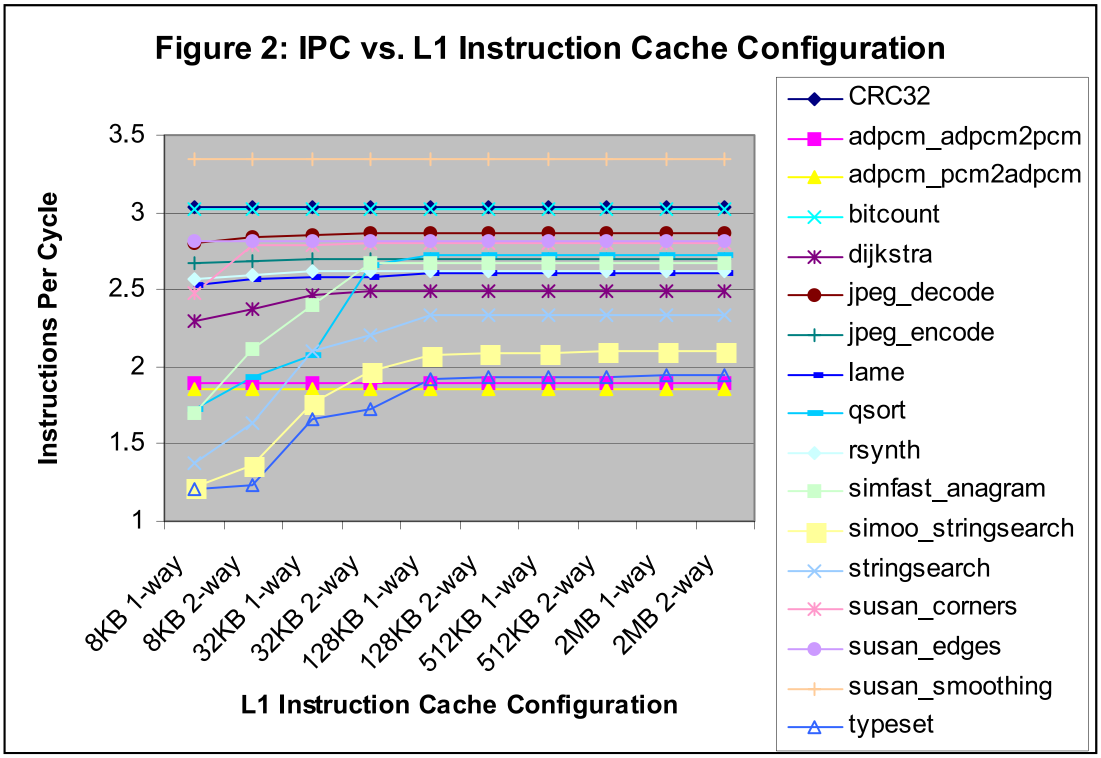
III. Results and Analysis

This chapter presents the characterization results and my analysis of them.

I.1 Level 1 Caches

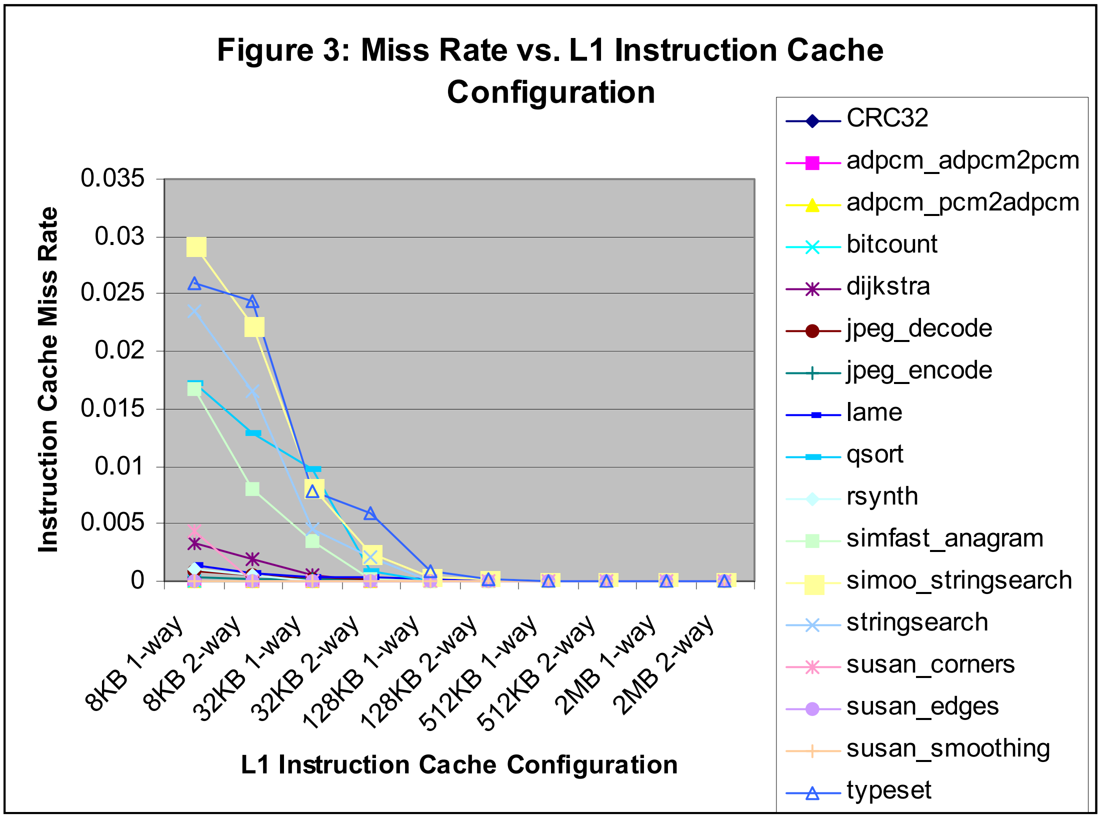
I.1.1 Level 1 Instruction Cache Results

Figure **2** presents the effects of different level 1 instruction cache configurations on the IPC. The cache configurations are arranged in rising order from left to right, with two-way associative caches to the immediate right of one-way associative caches of their respective sizes.



This figure shows that for some benchmarks, like *susan_smoothing* and CRC32, changing the level 1 instruction cache configuration did not affect the IPC values in any appreciable way. For the rest of the benchmarks, increasing the degree of associativity increased the IPC, as did increasing the cache size. Increasing the associativity never overpowered the effect of increasing the cache size, though the problem here might lie in the fact that the size increase was by a factor of four every time. *Simfast_anagram*, *qsort,* and *stringsearch* in particular look like the effects on their associativity might have a greater affect on the IPC than an early factor of two increase in cache size. In any case, most of the benchmarks affected by instruction cache configuration changes reach a plateau by the "128KB one-way associative" configuration, and the rest level off at "128KB two-way associative."

Figure 3 presents a similar graph, this time of the affects of instruction cache configuration changes on the cache miss rates. As could be expected, any increase in the IPC in Figure 2 is mirrored by a decrease in the miss rate in Figure 3. After all, a decrease in the miss rate means that the needed instructions are found in the cache more often, so they can start executing faster. As soon as a benchmark's instruction cache miss rate approaches zero in Figure 2, the IPC reaches its plateau in Figure 3. Benchmarks in Figure 2 which were unaffected by the configuration changes have a near 0 miss rate with all configurations.



III.1.2 Level 1 Data Cache Results

Figures 4 and **5** are the data cache counterparts of Figures 2 and 3 respectively. Looking at them, it is easy to see that for most benchmarks, data cache associativity plays a far greater role than data cache size. This is especially true for the benchmarks *rsynth*, *susan_smoothing*, and *simfast_anagram*. In fact, for *susan_smoothing* and *simfast_anagram*, the 8KB two-way associative level 1 data cache is far better than the 128KB one-way associative level 1 data cache. *Bitcount* and the two *adpcm* benchmark remain relatively unaffected.

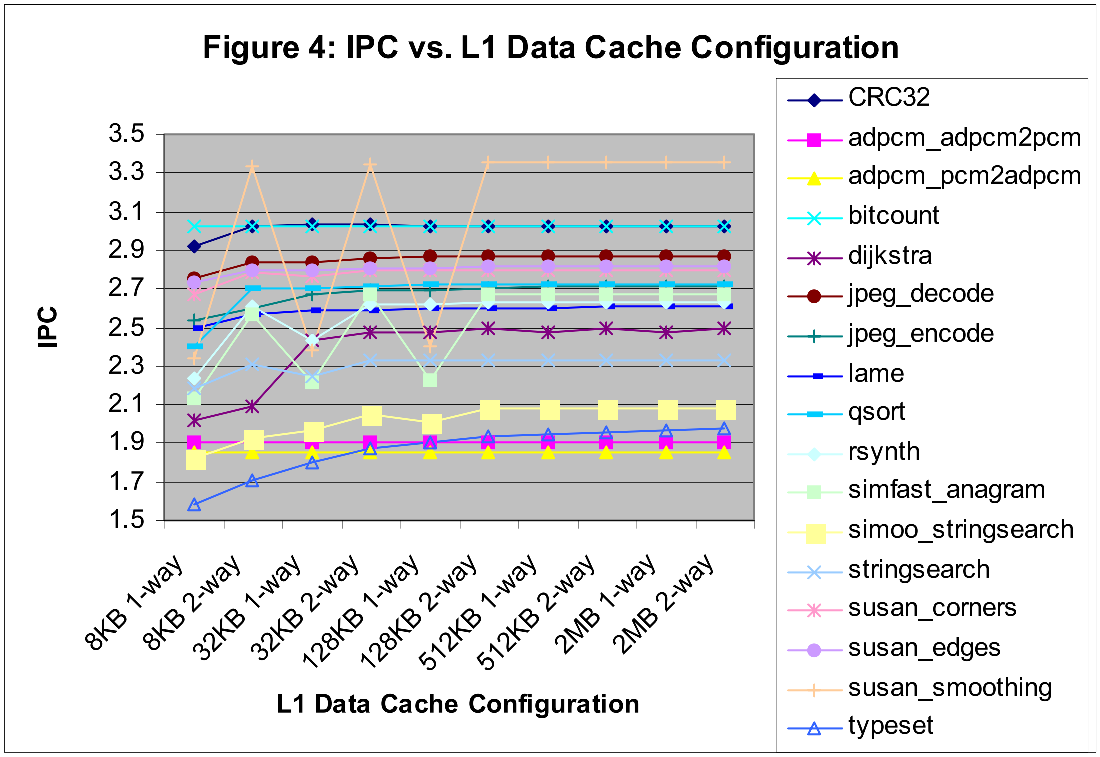


Figure 4: IPC vs. L1 Data Cache Configuration

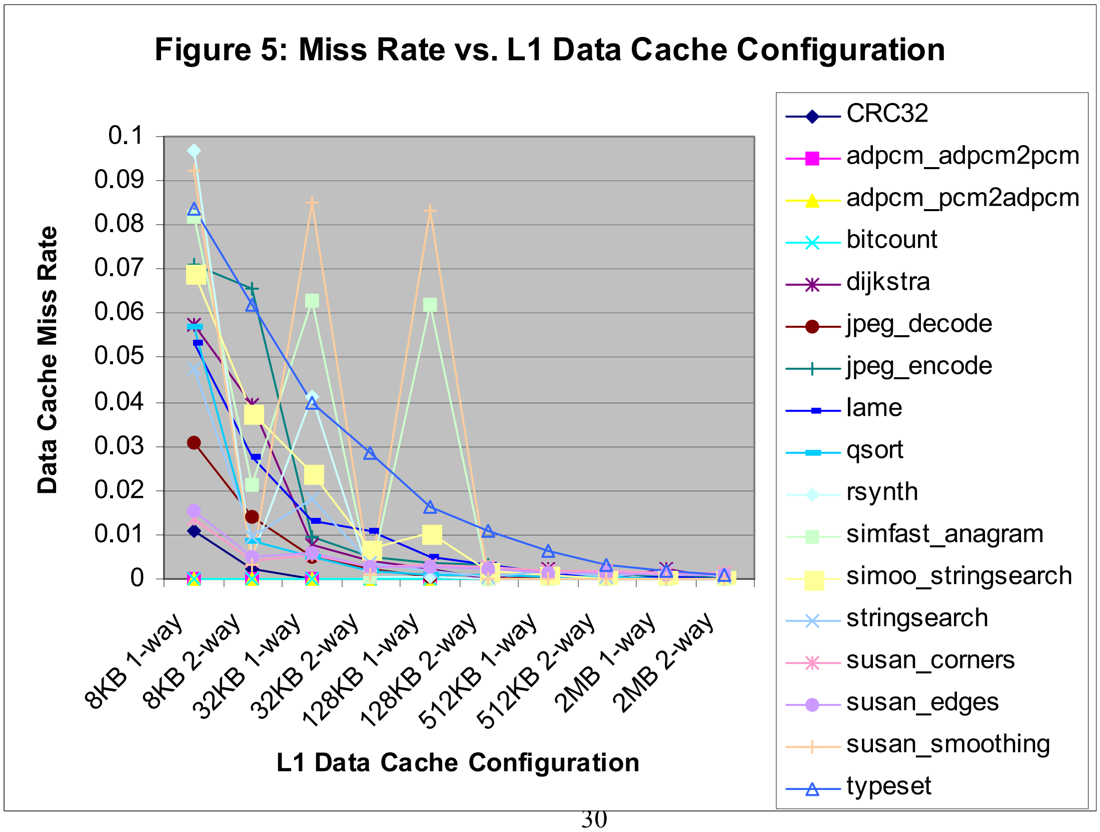


Figure 5: Miss Rate vs. L1 Data Cache Configuration

III.1.3 Level 1 Cache Result Analysis

It is obvious from the results that, for many of the benchmarks, I tested the associativity of the level 1 data cache has a profound effect on the benchmarks’ performance, while the effect of the associativity of the level 1 instruction cash is far smaller. What is not as obvious is *why* this is the case. The maximum data cache miss rates are also three times greater than the maximum instruction cache miss rates.

I think the reason for the greater miss rates in the data cache is that while instructions are usually accessed sequentially, depending on the benchmark, data could be accessed all over the place. This means that in an instruction cache, once a new block has been loaded into the cache, chances are there will not be another block that hashes to the same value for a long time, so associativity does not play as large a role. With data caches, on the other hand, it is easy to imagine variables being stored in any available place in memory. I have no idea, however, why benchmarks like *susan_smoothing* are so greatly affected by data cache associativity, while others are not.

The reason that most benchmarks plateau at higher cache sizes for both data and instruction caches is probably that the cache becomes large enough to completely fit almost all of the benchmark’s instructions or data inside it. There are few cache misses because there are few instructions or data segments not located in the caches.

III.2 Branch Predictors

Figure 6 presents the IPC values for different branch predictor configurations averaged across all benchmarks, and **Figure 7** does the same with direction- and address-prediction rates. While these figures do not represent any specific benchmark, they show that GAg configurations, are, overall, far worse than the other branch predictor configurations I tested, regardless of size. It almost seems like giving a GAg predictor more than 4000 entries is a waste of money. Bimodal configuration are second worst behind the GAg’s. Overall, there does not seem to be a more than 2% difference between the best of Gas, gshare, PAg’s, and PAs’. Size only seems to matter when it gets too low, but this might be a symptom of the relatively small size of the benchmarks.

Some interesting details appear when the behaviors of specific benchmarks are observed. *CRC32* was not affected by branch predictor choice at all. In *susan_smoothing*, the 4K Gag actually came in *second place* in IPC, with the 1M and 32K taking the fourth and fifth place respectively. The only other benchmark in which GAg did *not* come dead last was *adpcm_pcm2adpcm,* in which the bimodal configurations took last place instead. In some benchmarks, GAg configurations performed extremely badly. In *dijkstra,* GAg configurations scored an IPC of 1.32 when compared to the next lowest, a 2.47 by the worst bimodal configuration. Other abysmal performances by GAg occurred in *jpeg_encode, qsort,* and *sim-outorder_stringsearch*.

I do not know why GAg did so well in *susan_smoothing*, but it is seems to be a truly awful branch predictor. I think this means that making everything global is not a good idea. Bimodal configurations, the best of which never once appeared higher than an eighth pace, also seems a bad choice. PAg, PAs, GAs, and gshare branch predictors seem rather evenly matched. Representitives of each of these appeared in the top place of at least twice, though PAg and gshare had the best IPC the most often.

Of the general configurations I tested, only bimodal and gshare were characterized in the MiBench paper [7]. In Figure 3 of that paper, it clearly shows that gshare is better than bimodal in nearly all cases, which agrees with my results.

Figure 6: Average IPC as Affected by Branch Predictor Configurations

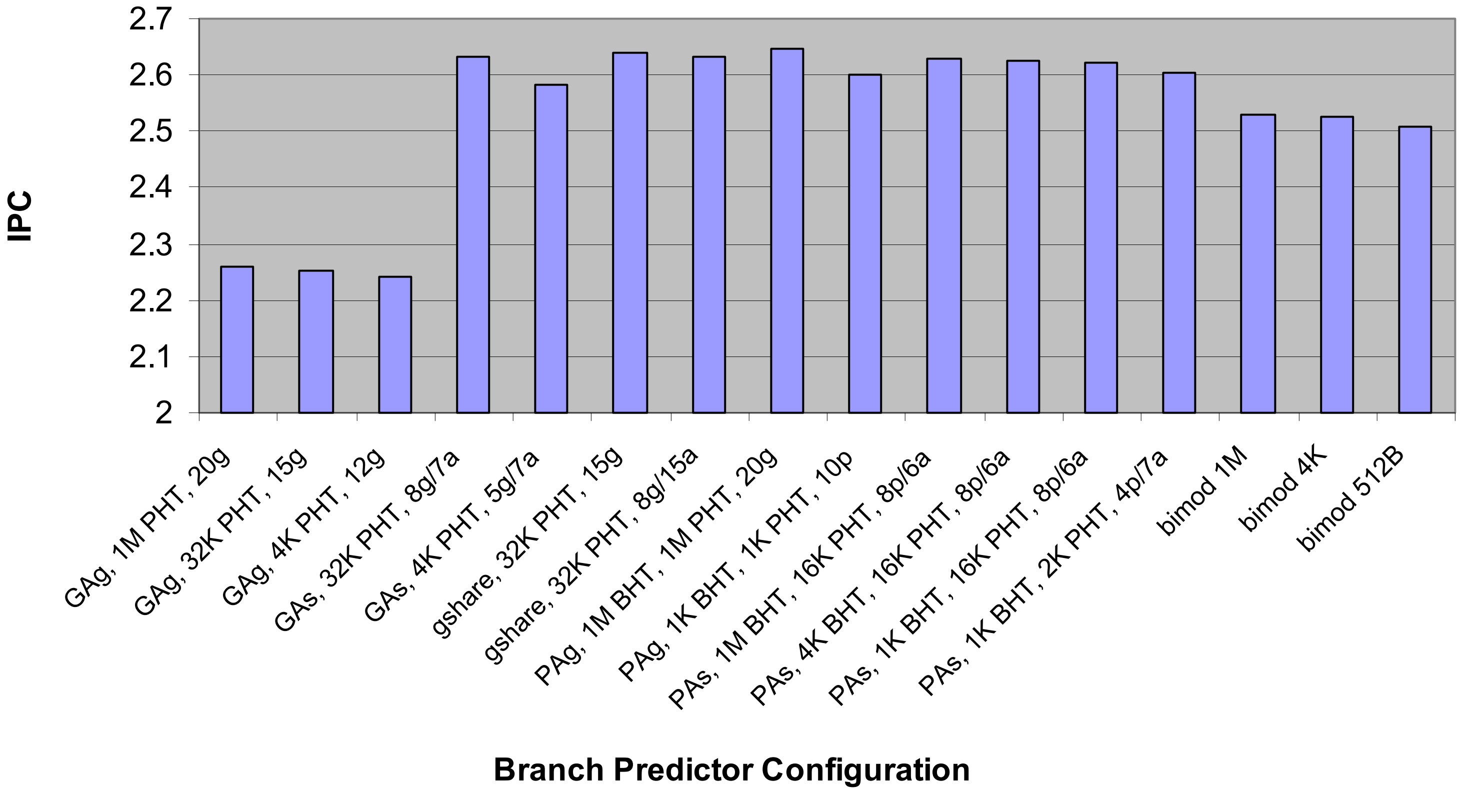
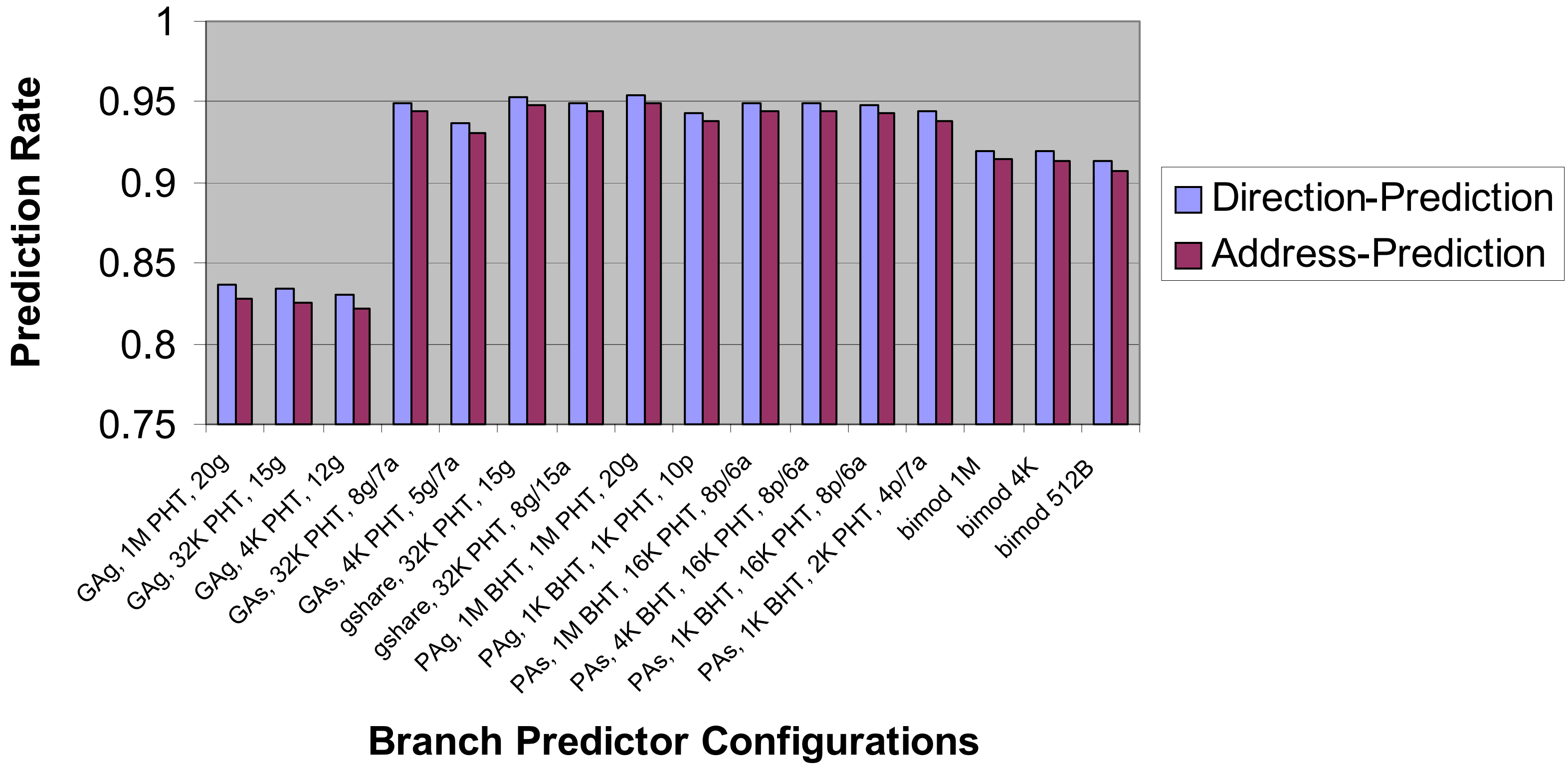


Figure 7: Effect of Branch Predictor Configurations on Average Prediction Rates



IV. Conclusion

For this thesis, I characterized the performance of different types of applications with multiple cache and branch predictor configurations and analyzed the results.

I learned that many programs greatly improve performance with a two-way associative data cache over a one-way associative data cache, but with instruction caches the improvement is not so profound. I also learned that GAg branch predictors are worthless when compared to other predictors, and bimodal predictors are the next worst things. GAs, gshare, PAs, and PAg predictors often give pretty much the same results, and their size only seems to matter when it gets really small.

For future studies of the effects of different cache and branch predictor configurations, I would recommend using much larger benchmarks in combination with the SimPoint and MRRL tools. An obvious recommendation would be to find and characterize new benchmarks, as was the original purpose of this thesis, and to find out why so many of the MiBench benchmarks either refused to compile or gave faulty results when simulated.

V. Bibliography

[1] Haskins, John Jr., and Kevin Skadron. "Memory Reference Reuse Latency: Accelerated Sampled Microarchitecture Simulation." UVA Computer Science Technical Report CS-2002-19, (2002.)

[2] Sherwood, Timothy, et al. "Automatically Characterizing Large Scale Program Behavior." Department of Computer Science and Engineering University of California, San Diego (2002.) ASPLOS X International Conference, San Jose, November, 2002.

[3] Skadron, Kevin, et al. "Branch Prediction, Instruction-Window Size, and Cache Size: Performance Tradeoffs and Simulation Techniques." IEEE Transactions on Computers 48.11 (1999): 1260-81.

[4] Skadron, Kevin. The LAVA Lab Homepage. http://lava.cs.virginia.edu/. Department of Computer Science, University of Virginia. (2002.)

[5] SimpleScalar LLC Homepage. http://www.simplescalar.com. (2002.)

[6] Burger, Doug, and Todd Austin. "The SimpleScalar Tool Set, Version 2.0." University of Wisconsin-Madison Computer Sciences Department Technical Report #1342, (1997.)

[7] Guthaus, Matthew, et al. "MiBench: A free representative embedded benchmark." University of Michigan Electrical Engineering and Computer Science. IEEE 4th Annual Workshop on Workload Characterization (2001.)

[8] MiBench Homepage. http://www.eecs.umich.edu/mibench/ (2003.)

[9] PICList Perl Command split() web page. http://www.piclist.com/techref/language/perl/split.htm (2003.)

[10] The SimpleScalar Mailing List Archive. http://ord.eecs.umich.edu/ss_archives/ (2003.)

[11] Leforgy, Charles. Message on the SimpleScalar Mailing List Archive. http://ord.eecs.umich.edu/ss_archives/0191.html (1999.)

[12] Deborah S. and Eric J. Ray. Visual Quick Start Guide Unix. Peachpit Press, (1998.)

APPENDIX: Branch Predictor Results

IPC Values

	Config 00	Config 22	Config 23	Config 24	Config 25
Program	Default 21264	GAg, 1M PHT, 20g	GAs, 32K PHT, 8g/7a	GAg, 32K PHT, 15g	gshare, 32K PHT, 15g
CRC32	3.0284	3.0279	3.0284	3.0279	3.0284
adpcm_adpcm2pcm	1.8996	1.7334	2.1827	1.7301	2.1643
adpcm_pcm2adpcm	1.8535	1.6838	2.0343	1.679	2.0137
bitcount	3.0243	2.9446	3.1202	2.9434	3.1321
dijkstra	2.4958	1.3145	2.5033	1.3164	2.503
jpeg_decode	2.8671	2.6625	2.8696	2.6577	2.8742
jpeg_encode	2.6955	2.1261	2.7056	2.1238	2.7036
lame	2.5969	2.3139	2.6043	2.3135	2.5961
qsort	2.7192	2.0478	2.7908	2.0458	2.7876
rsynth	2.6246	2.2176	2.6416	2.1886	2.6539
simfast_anagram	2.673	2.3012	2.7292	2.2986	2.7531
simoo_stringsearch	2.0684	1.5905	2.1997	1.5768	2.1721
stringsearch	2.3354	2.0725	2.4675	2.0495	2.4687
susan_corners	2.7945	2.7543	2.7895	2.7543	2.7837
susan_edges	2.812	2.6951	2.8409	2.6926	2.8586
susan_smoothing	3.3508	3.3054	3.2616	3.3053	3.3506
typeset	1.9066	1.5896	1.979	1.5744	2.0248

	Config 26	Config 27	Config 28	Config 29
Program	gshare, 32K PHT, 8g/15a	GAs, 4K PHT, 5g/7a	GAg, 4K PHT, 12g	PAg, 1M BHT, 1M PHT, 20g
CRC32	3.0284	3.0284	3.0279	3.0283
adpcm_adpcm2pcm	2.1827	2.0747	1.7262	2.1472
adpcm_pcm2adpcm	2.0343	1.9171	1.6741	1.9366
bitcount	3.1202	3.1161	2.9422	3.1562
dijkstra	2.5034	2.4959	1.3168	2.5018
jpeg_decode	2.8704	2.865	2.6577	2.8783
jpeg_encode	2.7078	2.6888	2.119	2.7224
lame	2.6049	2.5906	2.3019	2.6103
qsort	2.793	2.7408	2.0216	2.7396
rsynth	2.642	2.6063	2.1369	2.6456
simfast_anagram	2.7351	2.7115	2.285	2.7205
simoo_stringsearch	2.2052	2.0351	1.5416	2.2703
stringsearch	2.4682	2.3385	2.0151	2.6024
susan_corners	2.7926	2.7686	2.7452	2.7868
susan_edges	2.8408	2.8264	2.6892	2.8389
susan_smoothing	3.2616	3.2616	3.3318	3.3299
typeset	1.9907	1.8535	1.5697	2.0507

IPC Values

	Config 30	Config 31	Config 32
Program	PAs, 1M BHT, 16K PHT, 8p/6a	PAs, 4K BHT, 16K PHT, 8p/6a	PAs, 1K BHT, 16K PHT, 8p/6a
CRC32	3.0284	3.0284	3.0284
adpcm_adpcm2pcm	2.1723	2.1723	2.1723
adpcm_pcm2adpcm	1.9602	1.9602	1.9602
bitcount	3.121	3.121	3.121
dijkstra	2.5044	2.5043	2.5044
jpeg_decode	2.8892	2.8891	2.8871
jpeg_encode	2.7023	2.7018	2.7012

lame	2.603	2.6023	2.5984
qsort	2.722	2.722	2.722
rsynth	2.6465	2.6464	2.6451
simfast_anagram	2.7184	2.7184	2.7152
simoo_stringsearch	2.2032	2.1879	2.151
stringsearch	2.5045	2.5045	2.4876
susan_corners	2.7853	2.7852	2.785
susan_edges	2.8382	2.8382	2.8381
susan_smoothing	3.2515	3.2515	3.2515
typeset	2.0218	2.009	1.9916

	Config 33	Config 34
Program	PAs, 1K BHT, 2K PHT, 4p/7a	PAg, 1K BHT, 1K PHT, 10p
CRC32	3.0284	3.0283
adpcm_adpcm2pcm	2.1604	2.1354
adpcm_pcm2adpcm	1.9512	1.928
bitcount	3.1111	3.129
dijkstra	2.5042	2.5021
jpeg_decode	2.8771	2.8752
jpeg_encode	2.6978	2.6898
lame	2.5911	2.5883
qsort	2.7215	2.7294
rsynth	2.6427	2.6368
simfast_anagram	2.7028	2.7099
simoo_stringsearch	2.0746	2.0797
stringsearch	2.3948	2.3976
susan_corners	2.7846	2.7709
susan_edges	2.8285	2.8235
susan_smoothing	3.2515	3.2515
typeset	1.9447	1.949

IPC Values

	Config 35	Config 36	Config 37
Program	bimod 1M	bimod 4K	bimod 512B
CRC32	3.0284	3.0284	3.0283
adpcm_adpcm2pcm	1.7724	1.7724	1.7724
adpcm_pcm2adpcm	1.6148	1.6148	1.6148
bitcount	3.0437	3.0437	3.0437
dijkstra	2.4909	2.4906	2.4791
jpeg_decode	2.862	2.8615	2.8554
jpeg_encode	2.6955	2.6956	2.6944
lame	2.592	2.5909	2.5772
qsort	2.6756	2.6694	2.6208
rsynth	2.6049	2.6049	2.5972
simfast_anagram	2.6178	2.6177	2.6078
simoo_stringsearch	2.031	1.9979	1.8917
stringsearch	2.2043	2.2043	2.1718
susan_corners	2.7787	2.7786	2.774
susan_edges	2.8145	2.8145	2.8154
susan_smoothing	3.2514	3.2514	3.2513
typeset	1.8965	1.8859	1.8174

Address hit rates

	Config 00	Config 22	Config 23	Config 24	Config 25
Program	Default 21264	GAg, 1M PHT, 20g	GAs, 32K PHT, 8g/7a	GAg, 32K PHT, 15g	gshare, 32K PHT, 15g
CRC32	0.9999	0.9997	0.9999	0.9997	0.9999
adpcm_adpcm2pcm	0.7646	0.7162	0.8427	0.7148	0.8386
adpcm_pcm2adpcm	0.8004	0.7354	0.8562	0.7333	0.8538
bitcount	0.9509	0.9205	0.9677	0.9187	0.9693
dijkstra	0.9913	0.6297	0.9926	0.6297	0.9925
jpeg_decode	0.9573	0.8284	0.9576	0.8258	0.9615
jpeg_encode	0.9553	0.8264	0.9571	0.8253	0.9557
lame	0.9445	0.8037	0.9453	0.8001	0.9453
qsort	0.9679	0.7984	0.978	0.7982	0.9776
rsynth	0.9816	0.915	0.9851	0.9035	0.9962
simfast_anagram	0.9417	0.8112	0.9485	0.8063	0.9487
simoo_stringsearch	0.9326	0.7857	0.9485	0.781	0.9425
stringsearch	0.9304	0.8554	0.9477	0.8508	0.9458
susan_corners	0.9267	0.9022	0.9243	0.9013	0.922
susan_edges	0.9136	0.8704	0.9239	0.8701	0.9268
susan_smoothing	0.9945	0.9265	0.941	0.9265	0.9944
typeset	0.9097	0.7541	0.9281	0.7461	0.9395

	Config 26	Config 27	Config 28	Config 29
Program	gshare, 32K PHT, 8g/15a	GAs, 4K PHT, 5g/7a	GAg, 4K PHT, 12g	PAg, 1M BHT, 1M PHT, 20g
CRC32	0.9999	0.9999	0.9997	0.9999
adpcm_adpcm2pcm	0.8427	0.8145	0.7134	0.8344
adpcm_pcm2adpcm	0.8561	0.807	0.731	0.8291
bitcount	0.9677	0.9673	0.9182	0.9734
dijkstra	0.9926	0.9914	0.6301	0.9926
jpeg_decode	0.9576	0.9545	0.819	0.9669
jpeg_encode	0.9572	0.9517	0.8239	0.9589
lame	0.9454	0.9385	0.7959	0.9499
qsort	0.9784	0.9701	0.7954	0.9714
rsynth	0.9853	0.9787	0.8916	0.9949
simfast_anagram	0.9498	0.943	0.7988	0.9487
simoo_stringsearch	0.9505	0.9187	0.7664	0.963
stringsearch	0.9486	0.9265	0.8433	0.9725
susan_corners	0.9247	0.9179	0.8981	0.9245
susan_edges	0.9238	0.9197	0.869	0.9186
susan_smoothing	0.941	0.941	0.9331	0.9889
typeset	0.9318	0.8842	0.7386	0.9496

Address hit rates

	Config 30	Config 31	Config 32
Program	PAs, 1M BHT, 16K PHT, 8p/6a	PAs, 4K BHT, 16K PHT, 8p/6a	PAs, 1K BHT, 16K PHT, 8p/6a
CRC32	0.9999	0.9999	0.9999
adpcm_adpcm2pcm	0.8396	0.8396	0.8396
adpcm_pcm2adpcm	0.8332	0.8332	0.8332
bitcount	0.9679	0.9679	0.9679
dijkstra	0.9929	0.9928	0.9928
jpeg_decode	0.9736	0.9736	0.9725
jpeg_encode	0.9566	0.9564	0.956

lame	0.9458	0.9454	0.9435
qsort	0.9711	0.9711	0.9711
rsynth	0.9949	0.9949	0.9946
simfast_anagram	0.9479	0.9479	0.9471
simoo_stringsearch	0.9541	0.9519	0.9416
stringsearch	0.9572	0.9572	0.9543
susan_corners	0.9223	0.9223	0.9223
susan_edges	0.9183	0.9182	0.9182
susan_smoothing	0.9375	0.9375	0.9375
typeset	0.9424	0.9389	0.9333

	Config 33	Config 34
Program	PAs, 1K BHT, 2K PHT, 4p/7a	PAg, 1K BHT, 1K PHT, 10p
CRC32	0.9999	0.9999
adpcm_adpcm2pcm	0.8366	0.8315
adpcm_pcm2adpcm	0.8297	0.8255
bitcount	0.9675	0.9691
dijkstra	0.9928	0.9925
jpeg_decode	0.9628	0.9634
jpeg_encode	0.9542	0.9533
lame	0.9397	0.9376
qsort	0.9707	0.9708
rsynth	0.9943	0.9931
simfast_anagram	0.9452	0.9466
simoo_stringsearch	0.9302	0.9293
stringsearch	0.9388	0.9367
susan_corners	0.9221	0.9194
susan_edges	0.9161	0.9143
susan_smoothing	0.9375	0.9375
typeset	0.9198	0.9208

Address hit rates

	Config 35	Config 36	Config 37
Program	bimod 1M	bimod 4K	bimod 512B
CRC32	0.9999	0.9999	0.9999
adpcm_adpcm2pcm	0.7139	0.7139	0.7139
adpcm_pcm2adpcm	0.6516	0.6516	0.6516
bitcount	0.9591	0.9591	0.9591
dijkstra	0.9911	0.9911	0.9886
jpeg_decode	0.9536	0.9534	0.9458
jpeg_encode	0.9539	0.9539	0.9532
lame	0.9368	0.9363	0.9311
qsort	0.9624	0.9608	0.9505
rsynth	0.9777	0.9777	0.9764
simfast_anagram	0.9275	0.9274	0.9218
simoo_stringsearch	0.9269	0.9223	0.8962
stringsearch	0.908	0.908	0.901
susan_corners	0.921	0.921	0.9195
susan_edges	0.912	0.912	0.9122
susan_smoothing	0.9375	0.9375	0.9375
typeset	0.9076	0.9017	0.8715

Directional hit rates

	Config 00	Config 22	Config 23	Config 24	Config 25
Program	Default 21264	GAg, 1M PHT, 20g	GAs, 32K PHT, 8g/7a	GAg, 32K PHT, 15g	gshare, 32K PHT, 15g
CRC32	0.9999	0.9997	0.9999	0.9997	0.9999
adpcm_adpcm2pcm	0.7646	0.7162	0.8427	0.7148	0.8386
adpcm_pcm2adpcm	0.8004	0.7354	0.8562	0.7333	0.8539
bitcount	0.9509	0.9269	0.9677	0.9245	0.9693
dijkstra	0.9914	0.6332	0.9927	0.6333	0.9927
jpeg_decode	0.959	0.8326	0.9595	0.8302	0.9636
jpeg_encode	0.9555	0.8269	0.9575	0.826	0.9562
lame	0.9494	0.8091	0.9503	0.8054	0.9505
qsort	0.9717	0.8158	0.9818	0.8134	0.9814
rsynth	0.9835	0.9378	0.9859	0.9265	0.9969
simfast_anagram	0.9836	0.8534	0.9903	0.848	0.9905
simoo_stringsearch	0.9421	0.8018	0.9569	0.7973	0.9511
stringsearch	0.9317	0.8589	0.9515	0.8547	0.9502
susan_corners	0.9271	0.9028	0.9248	0.9018	0.9225
susan_edges	0.9138	0.8706	0.9241	0.8703	0.927
susan_smoothing	0.9945	0.9265	0.9411	0.9265	0.9944
typeset	0.9347	0.7837	0.9517	0.7765	0.963

	Config 26	Config 27	Config 28	Config 29
Program	gshare, 32K PHT, 8g/15a	GAs, 4K PHT, 5g/7a	GAg, 4K PHT, 12g	PAg, 1M BHT, 1M PHT, 20g
CRC32	0.9999	0.9999	0.9997	0.9999
adpcm_adpcm2pcm	0.8427	0.8145	0.7134	0.8344
adpcm_pcm2adpcm	0.8561	0.807	0.731	0.8291
bitcount	0.9677	0.9676	0.9243	0.9734
dijkstra	0.9927	0.9915	0.6336	0.9926
jpeg_decode	0.9595	0.9568	0.8234	0.9685
jpeg_encode	0.9575	0.9523	0.8245	0.9592
lame	0.9504	0.9436	0.8013	0.954
qsort	0.982	0.9728	0.8107	0.9745
rsynth	0.9861	0.9803	0.9154	0.9958
simfast_anagram	0.9915	0.9848	0.841	0.9903
simoo_stringsearch	0.9585	0.9291	0.7833	0.9683
stringsearch	0.9526	0.9293	0.8476	0.9738
susan_corners	0.9252	0.9183	0.8986	0.9249
susan_edges	0.924	0.9199	0.8692	0.9188
susan_smoothing	0.941	0.941	0.9331	0.9889
typeset	0.9554	0.9109	0.7689	0.9729

Directional hit rates

	Config 30	Config 31	Config 32
Program	PAs, 1M BHT, 16K PHT, 8p/6a	PAs, 4K BHT, 16K PHT, 8p/6a	PAs, 1K BHT, 16K PHT, 8p/6a
CRC32	0.9999	0.9999	0.9999
adpcm_adpcm2pcm	0.8396	0.8396	0.8396
adpcm_pcm2adpcm	0.8332	0.8332	0.8332
bitcount	0.9679	0.9679	0.9679
dijkstra	0.9929	0.9929	0.9928
jpeg_decode	0.9751	0.9751	0.974
jpeg_encode	0.9568	0.9567	0.9563
lame	0.9506	0.9502	0.9484
qsort	0.9733	0.9733	0.9733

rsynth	0.996	0.996	0.9958
simfast_anagram	0.9895	0.9895	0.9888
simoo_stringsearch	0.9601	0.9579	0.9485
stringsearch	0.9609	0.9609	0.9573
susan_corners	0.9227	0.9227	0.9227
susan_edges	0.9184	0.9184	0.9184
susan_smoothing	0.9375	0.9375	0.9375
typeset	0.9662	0.9627	0.9572

	Config 33	Config 34
Program	PAs, 1K BHT, 2K PHT, 4p/7a	PAg, 1K BHT, 1K PHT, 10p
CRC32	0.9999	0.9999
adpcm_adpcm2pcm	0.8366	0.8315
adpcm_pcm2adpcm	0.8297	0.8255
bitcount	0.9675	0.9691
dijkstra	0.9928	0.9925
jpeg_decode	0.9642	0.9649
jpeg_encode	0.9545	0.9536
lame	0.9445	0.9424
qsort	0.9728	0.9739
rsynth	0.9956	0.9945
simfast_anagram	0.9869	0.9883
simoo_stringsearch	0.9377	0.9381
stringsearch	0.9421	0.9394
susan_corners	0.9225	0.9199
susan_edges	0.9162	0.9144
susan_smoothing	0.9375	0.9375
typeset	0.9445	0.945

Directional hit rates

	Config 35	Config 36	Config 37
Program	bimod 1M	bimod 4K	bimod 512B
CRC32	0.9999	0.9999	0.9999
adpcm_adpcm2pcm	0.7139	0.7139	0.7139
adpcm_pcm2adpcm	0.6516	0.6516	0.6516
bitcount	0.9591	0.9591	0.9591
dijkstra	0.9912	0.9912	0.9892
jpeg_decode	0.9554	0.9552	0.9476
jpeg_encode	0.9541	0.9541	0.9534
lame	0.9412	0.9407	0.9371
qsort	0.9645	0.9629	0.9537
rsynth	0.9795	0.9795	0.9784
simfast_anagram	0.9722	0.9722	0.9666
simoo_stringsearch	0.9382	0.9336	0.9072
stringsearch	0.9111	0.9111	0.9042
susan_corners	0.9214	0.9214	0.9199
susan_edges	0.9122	0.9122	0.9124
susan_smoothing	0.9375	0.9375	0.9375
typeset	0.9341	0.9282	0.8987