Analysis of the TRIPS Architecture

by

Dipak Chand Boyed

Undergraduate Honors Thesis

Supervised by Prof. Stephen W. Keckler

Department of Computer Science

The University of Texas at Austin

Spring 2004

Table of Contents

Abstract 1

1. Introduction 2
   - 1.1. Introduction to the TRIPS Architecture 2
   - 1.2. Nomenclature 3
   - 1.3. Motivation 4
   - 1.4. Project Goals 5
   - 1.5. Background 5
2. Evaluating the TRIPS ISA 7
   - 2.1. The TRIPS Architecture 7
   - 2.2. Methodology 8
   - 2.3. Static and Dynamic Metrics 10
     - 2.3.1. List of Static Metrics 10
     - 2.3.2. List of Dynamic Metrics 11
   - 2.4. Source Code and Tools used 12
3. TRIPS-specific results 14
   - 3.1. Code Size 14
   - 3.2. Reads and Writes 17
   - 3.3. Critical Path and Fan-Out 18
   - 3.4. Useful Instructions and NOPs 20
4. TRIPS versus Alpha 22
   - 4.1. Why the Alpha 21264? 22
   - 4.2. Expectations 23
   - 4.3. Static and dynamic code sizes 23
   - 4.4. Instruction mixes 26
5. Conclusions 28
   - 5.1. Conclusion 28
   - 5.2. Future Work 29
6. References 30
7. Appendix 32
   - A. Location of the code metrics 32
   - B. Usage of the code metrics 32
   - C. Sample output 33
   - D. Micro-benchmarks metrics 34

Abstract

Introducing a new architecture has never been a trouble-free subject for computer architects. Growing complexity and compatibility issues often hinder the progress of new architectures even though the technological trends demand rapid, wholesome changes. TRIPS, is one such novel architecture that targets the problems of wire delays, memory latencies, power consumption and saturated parallelism. TRIPS aims to be a scalable, malleable, dynamically adaptive and non-specialized architecture that supports diverse applications. This thesis analyzes the advantages and disadvantages of the TRIPS architecture and its prototype. Because this thesis is the first to evaluate the TRIPS architecture, it establishes a foundation based on which the evaluation and analysis of the TRIPS architecture can be carried out smoothly in the future. The goals of this thesis are three-fold. The thesis provides details of the tools developed to analyze the TRIPS architecture, reports the metrics that provide data on the effects of the features of the TRIPS architecture on the program output, and evaluates the efficiency of the TRIPS model by comparing it with the Alpha 21264 (RISC) machine for a set of common source programs.

1 Introduction

Introducing a new architecture has never been a trouble-free subject for computer architects. Growing complexity and compatibility issues often hinder the progress of new architectures even though the technological trends demand rapid, wholesome changes. The post-RISC era today where clock speeds and pipeline depths are saturating is an example of the times that require a shift from the current ideas. TRIPS is one such novel architecture that targets the problems of wire delays, memory latencies, power consumption and saturated parallelism by providing an on-chip communication dominated, power-efficient execution model that exposes fine-grained concurrency.

1.1 Introduction to the TRIPS Architecture

The TRIPS (**T**era-Op, **R**eliable, **I**ntelligently adaptive **P**rocessing **S**ystem) architecture is a multi-disciplinary project being conducted at the Department of Computer Science, The University of Texas at Austin. Its goal is to design and develop a single-chip computing system with multiple functional units that provide Tera-op performance over a wide range of applications. TRIPS aims to be a scalable, malleable, dynamically adaptive and non-specialized architecture that supports diverse applications. The TRIPS design counters the problems of pipeline stage saturation and clock speed limits by providing a large grid of execution nodes that distribute the critical execution time and expose latencies throughout the components of the system [1, 4].

The TRIPS processor consists of a three dimensional array of ALU nodes connected in a network where each ALU contains local instruction and data storage buffers. Banks of partitioned instruction and data caches are placed around the ALUs. The system follows a block-atomic model of execution where an entire block of instructions is fetched and mapped onto the execution nodes in the array. Similar to a dataflow style ISA, the TRIPS compiler and scheduler encode each instruction's placement and its consumers, i.e. they allow a statically placed but dynamically issued (SPDI) execution model.

TRIPS resembles a dataflow style architecture especially within the intra-block level where the instructions represent nodes of a dataflow graph and are executed once their operands are available. However, at the inter-block level, the control flow mechanism of the TRIPS architecture differs from a dataflow machine and behaves exactly like a conventional processor with sequential memory semantics. TRIPS can also be related to a VLIW machine by comparing the 3-D array of execution nodes to a VLIW instruction. TRIPS also solves the power and centralization problems being faced by the superscalar computers. As seen from the above discussion, TRIPS combines many features of earlier models. The goal of this thesis is to create a foundation for analyzing and determining how well TRIPS has succeeded in achieving its goals.

1.2 Nomenclature

The following section defines some of the terms that are commonly associated with the TRIPS architecture and also lists terms that will be frequently used henceforward in this thesis.

GPA: A Grid Processor Architecture consists of an array of ALUs connected in a network fashion. Different blocks of scheduled instructions are mapped to these ALUs, which are executed in a dataflow order. This model allows more scalability by decentralization and distribution of larger structures and reducing their effect on the critical path [2].

SPDI: Static Placement Dynamic Issue is the architecture model where the compiler statically maps each instruction to its execution unit, but the instructions are issued dynamically at run-time.

EDGE: Explicit Data Graph Execution model is a new, post-RISC architectural model that allows SPDI and is ideally suited for GPAs by using a block-atomic execution model and direct on-chip communication between the instructions [3].

Block-atomic execution model: This execution model groups a fixed number of instructions into a block (similar to a basic block or hyperblock) that is statically made by the compiler and is fetched, executed and committed as a single, atomic entity by the ALUs.

TRIPS Block: A TRIPS block refers to a single atomic block that is the basic unit of the block-atomic execution model of TRIPS.

Static Blocks: Static Blocks refer to the compile-time TRIPS blocks that are generated by the compiler for a given program. Each static block has a distinct block name.

Dynamic Blocks: Dynamic Blocks refer to all the TRIPS blocks that are executed at run-time for a given program. The total number of dynamic blocks in a program is the sum of products of the static blocks and the number of times each static block is executed in the program.

Critical Path Instructions: Critical path instructions refer to the longest chain of instructions in the dataflow graph that represents a TRIPS block. The operands for these instructions wait for the values from the earlier instructions in the chain and hence are critical to the performance of the model.

Fan-Out Effect: The fan-out effect refers to the effect of moving the same piece of datum between the execution nodes in the grid of processors when a TRIPS block is mapped to it. A GPA like the TRIPS model increases on-chip communication between the execution nodes by transferring values between them instead of accessing the register file each time. However, moving the same datum to many nodes also requires extra overhead (move) instructions [5].

1.3 Motivation

The goal of this thesis report is to analyze the advantages and disadvantages of the TRIPS architecture and the prototype. I present details on the tools developed to generate code metrics that provide results used to analyze the outputs of a wide set of toy benchmarks and programs on the TRIPS prototype, and relate their behavior to the design of the TRIPS architecture. I also compare the efficiency of the TRIPS ISA with a conventional, Alpha (RISC) machine so that the relative efficiency and performance of the TRIPS ISA can be measured. The ultimate goal in the future would be to conduct a comprehensive analysis of the TRIPS ISA and be able to determine how well the TRIPS architecture meets its goals.

The motivation behind this thesis report was to conduct an initial analysis of the TRIPS Architecture. Evaluating an on-going research project has often not received its due attention because of numerous hindrances and difficulties. This thesis covers some areas where the current infrastructure and tools allow comparison and evaluation of the TRIPS architecture. I hope to establish a foundation based on which the evaluation and analysis of the TRIPS architecture can be carried out smoothly in the future. By providing results of some of the metrics, I will also create a way for future researchers to map the progress of the TRIPS project and relate specific modifications in the model to the changes in the code behavior.

This thesis presents the infrastructure available to evaluate the TRIPS architecture. However, the results presented in the thesis only reflect a mid-stream analysis and in no way represent the final evaluation of the TRIPS processor. The results have been provided only as a guideline for refining the performance of the TRIPS architecture.

1.4 Project Goals

The goals of this thesis are to present to the reader, information and results that are helpful in analyzing various features and aspects of the TRIPS architecture. The thesis provides in-depth information regarding the following aspects in particular:

1. ***Details of the tools developed to analyze the TRIPS architecture.***
2. ***Metrics that provide data on the effects of the TRIPS architecture on program output.***
3. ***Comparison of the TRIPS output with an Alpha 21264 (RISC) machine.***

These goals are achieved by developing and running a group of software-based metrics on the TRIPS output for certain programs and comparing them to the Alpha output. The metrics are used to observe the correct functionality of certain components of the TRIPS tool-chain like the compiler optimizations, observe differences in the program output due to the two different architectures, and attribute specific output behavior to certain specific features in the TRIPS model.

I consider my contribution to the analysis of the TRIPS architecture to have three main axes that correspond directly to the three project goals mentioned above.

1. *Development of the testing scripts and code metrics*: I have created tools and scripts that make it feasible to generate metrics and results used to analyze the architecture.

2. *Results of metrics that outline TRIPS specific behavior*: The metrics when compiled over a group of source programs reveal results that give us some insight into the TRIPS specific features and aspects.

3. *Results of metrics that compare TRIPS to another architecture*: Lastly, the metrics also provide a way to compare the overall efficiency of the TRIPS output for a given program with the Alpha output of the same program. This also provides a way to compare the two different architectures.

This thesis is targeted towards two groups of audience. It serves specific information to the members of the TRIPS research group interested in the performance analysis of the TRIPS architecture, as well as provides general information to the public interested in knowing about the TRIPS project.

1.5 Background

The advent of any new architecture calls for a thorough analysis of the model and comparison with already existing models. Any new idea will have both its supporters as well as the opposing camp. With every change in the technology, ideas have come and been debated thoroughly. The RISC versus CISC machine comparisons are perhaps the most well known of these debates [12, 13]. All new architecture including VLIW, SuperScalar models have gone through the phase of analysis and comparison. Individual work has also been done in the area of determining the metric of such

comparisons [7, 8]. The usual metrics used in comparing architectures have been the static and dynamic information [8], usage of addressing mode and instructions [7], instruction mix, code sizes, etc.

TRIPS, being the first of its kind architecture, also presents its own problems when being compared to the existing models. Relating the block execution model and the effect of instructions like move that are present exclusively in TRIPS, to other architectures is particularly difficult. This thesis compares TRIPS with other models. The hope is for future researchers to notice the work and make further progress.

The remainder of the report is organized as follows. Chapter 2 describes the various features of the TRIPS architecture and describes the methodology of the scripts and tools that were written to generate the metrics. Chapter 3 reports the results of the metrics that were ran for different configurations of the TRIPS prototype and some TRIPS-specific features. Chapter 4 concentrates on comparing the TRIPS output with the Alpha machine output and lists the advantages and disadvantages of the two machines over one another. It compares the code sizes and instruction mixes of certain common source code, on the two machines. Chapter 5 discusses and analyses the results of the metrics and provides further insight into area of future work.

2 Evaluating the TRIPS model

2.1 The TRIPS Architecture

The goal of TRIPS is to design a single-chip computing system with multiple functional units that provide Tera-op performance over a wide range of applications. TRIPS aims to be a scalable, malleable, dynamically adaptive and non-specialized architecture that supports diverse applications. The TRIPS design counters the problems of pipeline stage saturation and clock speed limits by providing a large grid of execution nodes that distribute the critical execution time and expose latencies throughout the components of the system [1].

The TRIPS processor consists of a three dimensional array of ALU nodes connected in a network where each ALU contains local instruction and data storage buffers. For the prototype the grid consists of a 4 x 4 network of execution nodes. Banks of partitioned instruction and data caches are placed around the ALUs as shown in Figure 1 below. The system follows a block-atomic model of execution where an entire block of instructions is fetched and mapped onto the execution nodes in the array. The TRIPS compiler and scheduler statically encode 128-instruction blocks onto the grid giving 8 instructions to each node.

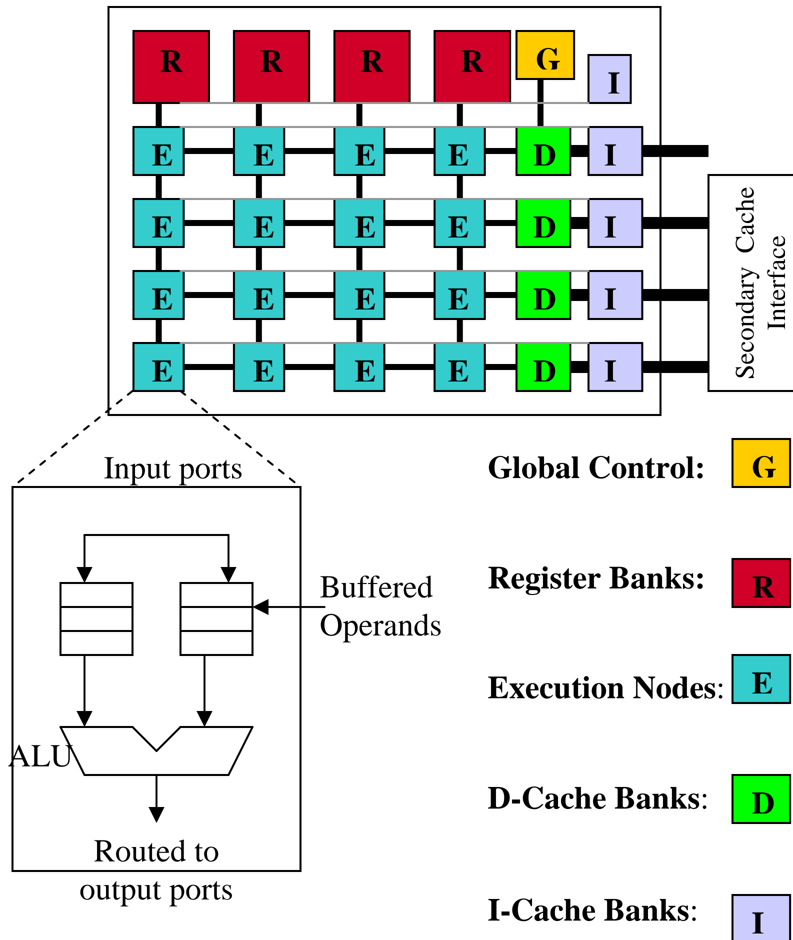


Figure 1. TRIPS architecture model

The figure describes a 4x4 grid of execution nodes where blocks of 128 instructions are mapped. Figure adapted from [3].

2.2 Methodology

The basic idea of the tools created is to examine a TRIPS assembly file (in TIL or TASL [11]) and executable outputs for a given program file and provide software-based metrics. The metrics were the result of the output of some wrapper scripts written in Perl that parsed through the TRIPS code of the source program. Thus the implementation of a given source program could be analyzed on the TRIPS machine by studying the metrics. These metrics can be broadly categorized into (a) static metrics and (b) dynamic metrics.

These metrics fit in nicely within the already existing TRIPS toolchain. The TRIPS toolchain consists of the tools that take in a source program in high-level language at one extreme end, generate intermediate code that feed the other tools in the chain and finally produce the executable output at the other end. The TRIPS compiler produces the TRIPS intermediate language (TIL) for a given C source program. The scheduler maps the intermediate code into the executable nodes by producing the TRIPS assembly language (TASL) file. The assembler then generates the object

code for the TASL file, which is then linked and loaded to produce the executable. Finally the TRIPS architectural simulator runs the executable and also has the capability to produce a trace of the execution of the program.

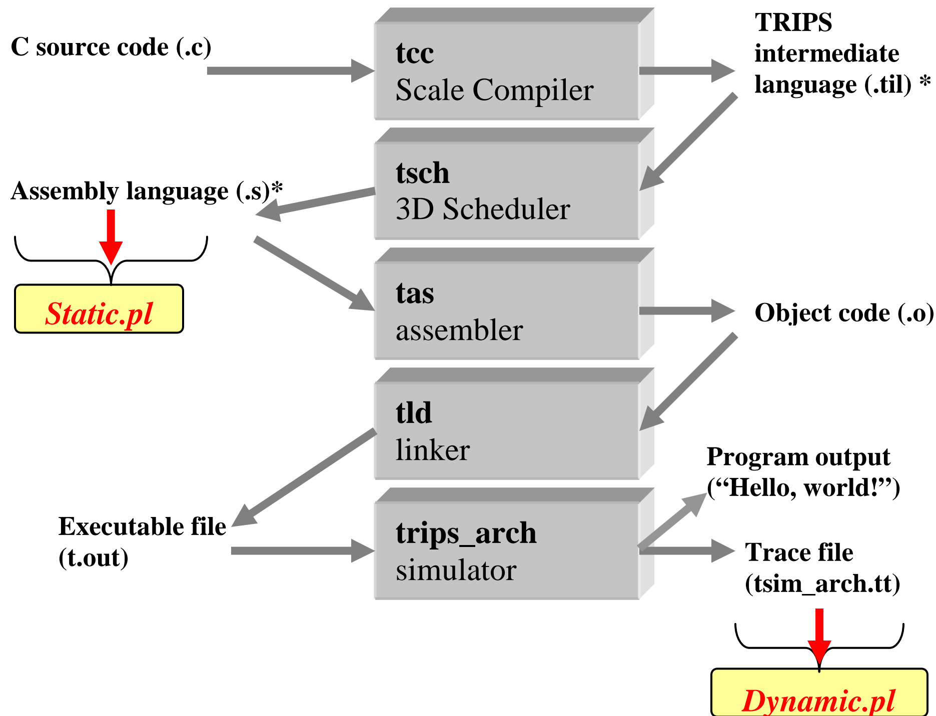


Figure 2. The TRIPS Toolchain and the Code Metrics

The above figure shows the TRIPS toolchain and the places where the scripts for the code metrics fit in. The scripts static.pl and dynamic.pl were written to generate the code metrics. * The TIL and TASL files can also be hand-generated and fed to the scripts.

Static metrics refer to the data and information gathered by studying the static assembly file. These metrics include information such as the static code size, number of TRIPS block, number of reads and writes per block and other metrics that are generated by the compiler. These metrics were generated by the file called **static.pl** that takes as input the TASL assembly language file (*.s) for a given program. The TASL file can be hand-written or generated from the compiler and scheduler for a C source program.

Dynamic metrics refer to the data and information gathered by studying the trace file of the architectural simulator. These metrics include information such as the dynamic code size, total instruction mix, flow of critical instructions and fan-outs per block. These metrics were generated by the file called **dynamic.pl** that takes as inputs the TASL assembly language file (*.s) and the architectural simulator's trace file (tsim_arch.tt). The dynamic metrics script combines the static information of the assembly file to the dynamic execution trace of the simulator.

In general, the dynamic metrics weights each TRIPS block according to the number of times it was actually executed and hence is a better guideline for analyzing the TRIPS toolchain for a given source program. The static metrics have still been provided as they serve as a good way to study

the compiler outputs, compare the TRIPS atomic-block format to the Alpha output and serve as an input to the dynamic metrics.

2.3 Static and Dynamic Metrics

The metrics generated by the scripts include both static as well as dynamic information. These metrics are useful in analyzing the TRIPS toolchain and in comparing the TRIPS output to that of the Alpha machine. There are some metrics that give details about very TRIPS-specific data such as reads and writes in a block, while others like the overall code size and instruction mix, compare the TRIPS and Alpha machine outputs. Table 1 below gives a list of all the static and dynamic metrics that the scripts can generate. All of the metrics that have been provided here are also discussed in further detail.

Comparing	Static Metrics	Dynamic Metrics
TRIPS *(Chapter 2)*	• Code size for different compiler optimizations • Number of block for different compiler optimizations • Reads and writes per block	• Reads and writes per block • Useful instructions and NOPs per block • Moves per block • Length of critical path instructions • Length of maximum fan-out of a value per block
TRIPS versus Alpha *(Chapter 3)*	• Static code size • Static Instruction mix	• Dynamic code size • Dynamic Instruction mix

Table 1. Summary of Static and Dynamic Metrics and what they compare
The table lists out the metrics that the scripts generate according to the type of data (static or dynamic) and according to whether they compare TRIPS alone or TRIPS and Alpha.

2.3.1 List of Static Metrics

Static Code Sizes for different compiler configurations.
This metric calculates the variance in the static code size of different programs by adding different compiler optimizations. It is used to verify the compiler optimizations and to observe the result of applying different optimizations. The static code size provides a measure of the complexity of the source programs, which must be kept in mind while observing the dynamic information.

Number of Static TRIPS Blocks for different compiler configurations.
This metric is used to relate a given TRIPS block to the program code and the effect of compiler optimizations on the number and size of TRIPS block per program. This metric is in many ways analogous to the static code size and when combined with it gives a measure of the average length of TRIPS blocks created by the compiler for a given source code.

Average number of static reads and writes per TRIPS block

This metric calculates the number of times we access the read and write instructions per TRIPS static block. The read and write instructions are used to retrieve an existing value from a register and write a new value to a register respectively. This metric gives a measure of the usage of the Read and Write Queues for a given block.

TRIPS versus Alpha Static Code Size and Instruction Mix

These metrics compare the size of the scheduled TRIPS code to the Alpha code and point out the differences in the way the same code is scheduled. The different types of instructions representing the same program in the two machines are also compared. These metrics are a measure of the efficiency of the two instruction set architectures and can be useful references for calculating the progress of the TRIPS toolchain.

2.3.2 List of Dynamic Metrics

Average number of dynamic reads and writes per TRIPS block

This metric is used to calculate the actual number of times we access the Read and Write Queues for a given program. It directly corresponds to the starting points (read) and ending leaves (writes) of all the instruction trees representing the TRIPS block. This metric gives the measure of the usage of the Read and Write Queues for a given program.

Average number of useful Instructions and NOPs per TRIPS block

This metric tells us what fraction of the 4x4x8 TRIPS block (i.e. how many of the 128 execution nodes) is being used and what fraction of the capacity of the TRIPS block is currently unused. It can be useful in studying the compiling and scheduling powers and will become an important metric in evaluating the success of hyperblock formation.

Average number of moves per TRIPS block

The GPA model of TRIPS requires a lot of moving of data between the execution nodes using the move instruction. This metric evaluates the usage of move instruction over all the TRIPS block in a program and calculates its fraction out of the total instruction mix.

Analysis of the Critical Path lengths in TRIPS blocks

This metric is a measure of the total amount of instruction level parallelism present per TRIPS block by indicating the longest chain of dependent instructions in a given TRIPS block. By comparing the length of the critical path to the size of the block, we can measure the degree of instruction level parallelism. This metric traces the data dependency tree that represents every TRIPS block and returns the length of the longest path from a root (entry point of the block) to any leaf (exit point of the block).

Analysis of the maximum length of fan-out in TRIPS blocks

This metric gives us an approximation of overhead used to transfer data between execution nodes per TRIPS block. It returns the maximum number of times any data is moved between the execution nodes in a given TRIPS block. The metric is not the total measure of all the moves in a

block but only the measure of one datum that is moved the most number of times in a block. This metric represents the overhead of moving data between the nodes in a GPA model like TRIPS.

TRIPS versus Alpha Dynamic Code Size and Instruction Mix
These metrics perhaps combine to be the most important measure of evaluating the overall efficiency of the TRIPS model. They compare the weighted, dynamic number of instructions that are executed for the same program in the two machines and the weighted, dynamic list of instruction types present in the program code. These metrics give us an idea of the amount of work being done by both the machines and in the future will be the single-most important metric to track the efficiency and progress of the TRIPS model.

2.4 Source Code and Tools used

In order to run the scripts and generate results for evaluating the TRIPS model, a test suite of toy programs and frequently executed functions from some SPEC 2000 benchmarks are used. These programs are written in C and have also been validated and compiled on the gcc version 3.3.2 compiler. Most of the C source files are either loop-centric or recursive. They are as follows:

File Name	Description
Matrixmultiply	Integer multiplication of 10x10 matrices
Ackermann	Fastest growing primitive recursive function
Binary search	O(log n) search algorithm
FFT	Fast fourier transform
Factorial	Recursive computation of 120!
Fibonacci	Recursive computation of 20th Fibonacci number
a_number()	Loop from ammp
longest_match()	Loop from gzip

Table 2. List of C source code

The inputs to the scripts require a TRIPS assembly language (TASL) file for a given program, which can either be hand-generated or compiled on the TRIPS toolchain components like the Scale compiler and scheduler. One can also generate a TASL file from a hand-coded TRIPS intermediate (TIL) file that is run on the scheduler. The TASL file is then linked and executed on the TRIPS architectural simulator to generate the trace file. These scripts were also run on some hand-generated TIL and TASL versions and other source files of microbenchmarks, the results of which are shown in Appendix D.

All the C source programs used in this thesis were successfully compiled and executed on the following versions of the tools listed below.

Tools	Description	Version Info.
tcc	Wrapper script	1.109
scale	Scale Compiler	Sun May 25 00:15:07 CDT 2004
tsch	Scheduler	1.45
tas	Assembler	1.147 2004/04/26 20:20:26
tld	Linker	GNU v2.12.90 2002/04/29
tsim_arch	Simulator	0.29+

Table 3. List of tools used and version number information

The above information might be useful for readers interested in reproducing the results or tracking the changes in the results with newer versions of the tools.

Having established the methodology, procedure and the tools used to generate the results, the following section and the following chapter will now present the results generated from the running the static and dynamic scripts on the above mentioned C source files. The following section will list the metrics that apply only to the TRIPS architecture model while the next chapter details the metrics (dynamic code size and instruction mix) used to compare the TRIPS and Alpha machine.

3 TRIPS-Specific Results

The TRIPS architectural model has many unique features that differ from conventional RISC and sequential semantics machines. Having already established the methodology of the tools and scripts written to produce the analysis metrics, this chapter focuses on the second goal of the thesis, i.e. discussing the results of the metrics that correspond to the specific features and details of the TRIPS architecture.

Both static and dynamic information is gathered to produce the output of the metrics that discuss the results shown below. The TRIPS specific results have been broken into four separate categories that cover the areas of static and dynamic code sizes, metrics on the static and dynamic usage of reads and writes, length of critical path instructions, maximum fan-out effect, moves in a TRIPS block and average number of useful instructions in the TRIPS block.

3.1 Code Size

The most unique and striking feature of the TRIPS architectural model is the block-atomic execution model, hence it is natural that the first metrics that was generated was related to the TRIPS block [Figure 3]. The first metric counts the number of TRIPS blocks used to represent a given source program by parsing through the static information from the TASL assembly file [11].

The decrease in the number of the static TRIPS block generated by the compiler is observed by turning on increasing levels of optimizations (-O flag). Adding slight optimizations and loop

unrolling, loop-flattening optimizations to the code usually causes a slight decrease in the total number of TRIPS block. In addition, with the successful implementation of the hyperblock formation we will be able to see an even greater decrease in the number of TRIPS block. I decided to use the default (-O3) optimization level as a standard for all the other metrics and not the hyperblock (–O4) optimization level because of the on-going work on the hyperblock implementation is not complete and verifiable as of this writing. The hyper block optimization could be used as the standard in the future upon obtaining enough confidence on its successful implementation.

Figure 3. Number of Static TRIPS block

The static blocks are generated by the different compiler optimization levels. The four different categories correspond to the –O0, -O2, -O3 (default) and –O4 optimization levels of the tcc compiler respectively.

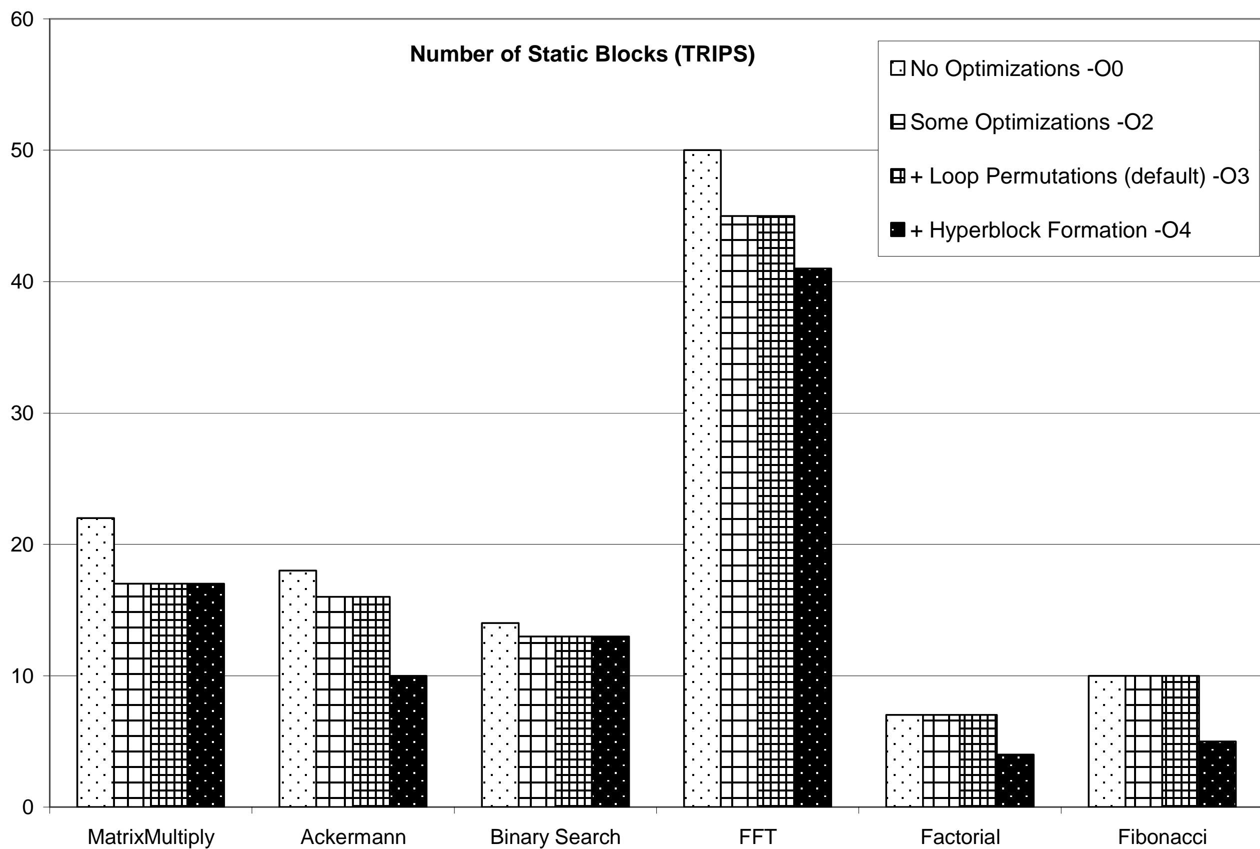


Figure 3 represents the number of blocks that were generated by the compiler for a given source code. However, this is only half the story told as it does tell us anything about the size of the blocks, the other half involves measuring the total number of instructions that were created. Figure 4 below shows us the total static code size for the given programs. It lists the total number of instructions generated by the compiler for the different optimization levels.

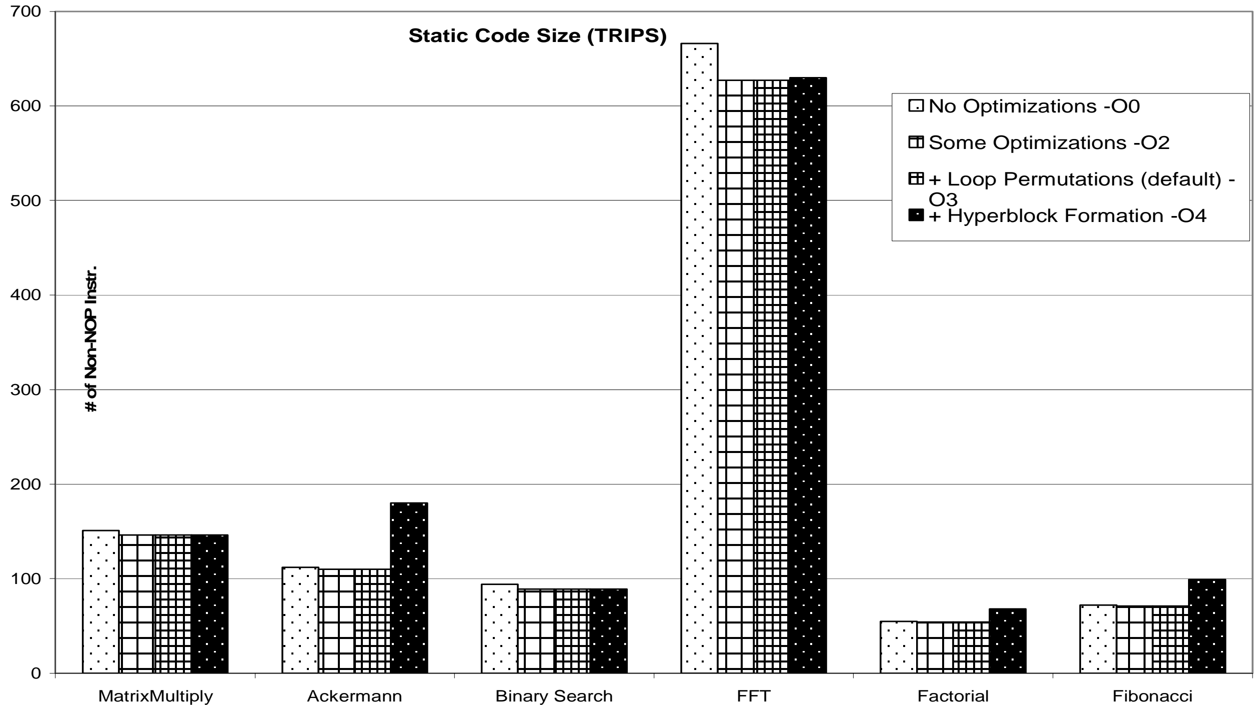


Figure 4. Static Code Size

This graph shows the change in the static code size (total number of instructions generated) generated by the compiler for different optimization levels.

On combining the results of Figures 3 and 4, I obtain the average size of the TRIPS static block generated by the compiler for the given programs. Table 4 below lists the average number of instructions generated per TRIPS block by the compiler. The numbers show that the TRIPS block's capacity is mostly unused and strictly demands the use of hyperblock formations. This table present data, which is a perfect example of the reference, future TRIPS researchers can use to evaluate the effectiveness of the hyperblocks when its implementation is complete.

Program	Avg. # of Instructions scheduled/ Block
Matrixmultiply	8.59
Ackermann	6.88
Binary search	6.85
FFT	13.93
Factorial	7.71
Fibonacci	7.1
A_number()	7.27
longest_match()	9.69

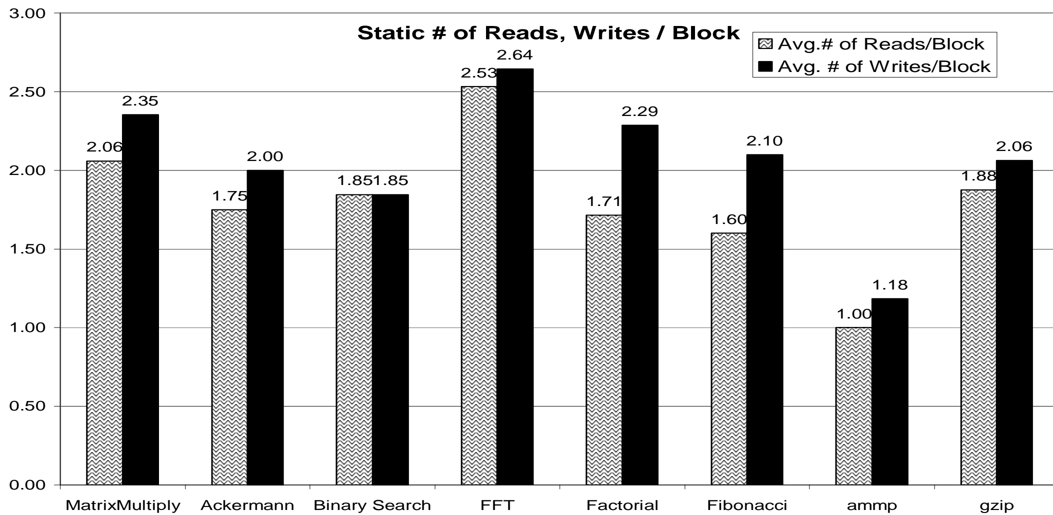
Table 4. Average number of Instructions per static TRIPS block

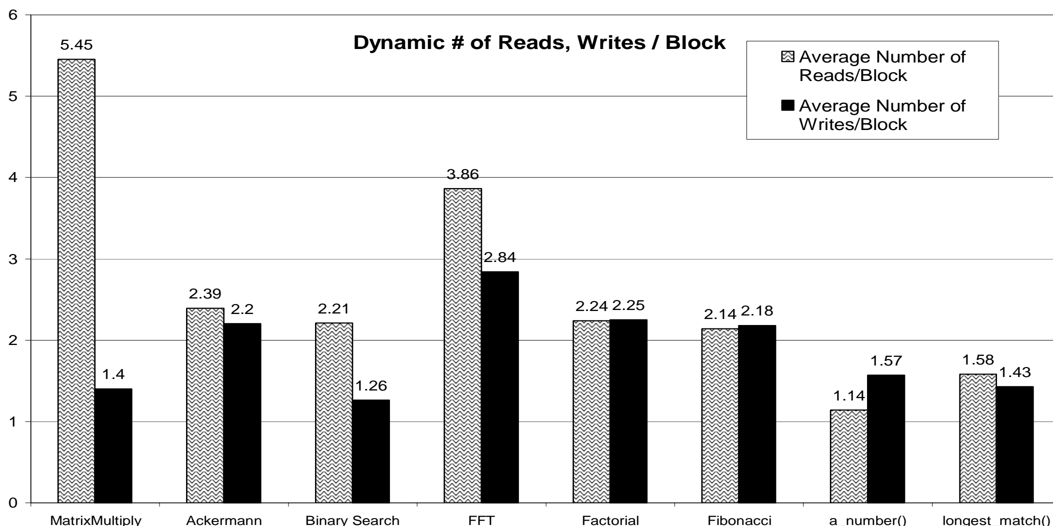
3.2 Reads and Writes

The next group of metrics deal with the static and dynamic number of reads and writes instructions observed in the TRIPS blocks. The reads and writes are the way of accessing and updating values in the register file. TRIPS has a GPA model with a grid of executing nodes moving values between them, hence one expects fewer access to the register file and more move instructions in the TRIPS model when compared to a RISC machine like Alpha. Thus, observing the frequency of accesses to the read and write queues is an important measure in validating the correctness of the TRIPS model.

Figure 5 below represents the static average of the number of reads and writes scheduled per TRIPS block. Figure 6 represents the same information over a dynamic execution scale where the TRIPS blocks are weighted according to the number of times they are actually executed. Hence, Figure 6 describes a more accurate picture of the frequency of accesses to the read and write queues for a given program.

Figure 5 and 6. Static and Dynamic average reads and writes per block





3.3 Critical Path and Fan-Out

Another interesting metric to study is that of the length of critical path of dependent instructions in a TRIPS block. The next metric whose results are given below in Figure 7, represents the length of the longest chain of dependent instructions that pass on data to each other in a TRIPS block. Each TRIPS block can be represented in the form of a data dependency graph. This metric calculates the longest path from a root of such a tree to any of its leaves. By comparing the length of the critical path instructions to the size of the block, it becomes possible to measure the amount of instruction level parallelism in the given block.

For example in the matrixmultiply program, the average size of the TRIPS block is 18.8 while the average length of critical path instructions is about 8. Hence on the average we can approximate the instruction level parallelism (ILP) to be about: 18.8/8 ~ 3. The ILP approximation is based on the assumption that on average all the non-critical instructions in the block also group into a dependency chain with the same length as the critical path. However, in reality many non-critical instructions may not lie in such long dependent chains. Hence, the ILP approximation gives us a good lower bound into the ILP available within a TRIPS block.

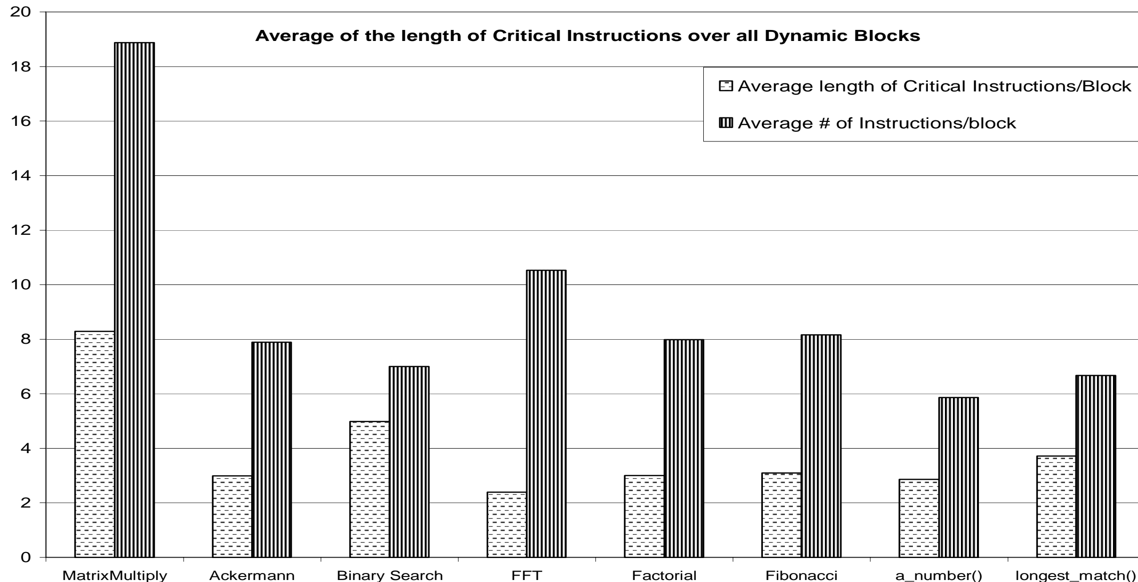


Figure 7. Length of Critical Path Instructions

This graph shows the average length of the critical chain of dependent instructions per TRIPS block.

Measuring the overhead of moving data between the execution nodes in a GPA model like that of TRIPS is another important issue. TRIPS uses the move instruction to transfer data between the nodes in a block. Thus, one expects frequent use of the move instruction and fewer accesses to the register file. The next metric measures the maximum number of nodes (fan-out) to which a single datum is transferred in a given block. The average of the maximum fan-out per block is taken over all the dynamic blocks and has been shown in the graph in Figure 9. The caveat however, is that this result does not cover the entire fan-out effect of all data that are moved in a block but only calculates the fan-out of the datum that is transferred to the most number of target nodes in a given

block. This metric only analyses the move instructions in a block and traces the target nodes of the moves. It returns the number of internal and leaf nodes a datum is transferred to, for only the datum that is transferred to the most of number of nodes in a block. Figure 8 below shows the mechanism of how this metric computes its results.

A. TRIPS Block

```
.bbegin main$4
;;;;;;;;;; Begin read preamble
R[1] read G[1] N[1,0] N[4,0]
;;;;;;;;;; End read preamble
N[1] <0> addi 88 N[5,0]
N[5] <1> mov N[9,0] N[12,0]
N[0] <2> genu 488 N[4,1]
N[4] <3> add N[16,0]
N[9] <4> mov W[3]
N[12] <5> mov W[0]
N[16] <6> mov W[1]
N[32] <7> genu %lo(main$5) N[8,0]
N[8] <8> app %bottom(main$5) W[2]
N[2] <9> callo I[0] MatrixMultiply
;;;;;;;;;; Begin write epilogue
W[2] write G[2]
W[0] write G[4]
W[3] write G[3]
W[1] write G[5]
;;;;;;;;;; End write epilogue
.bend
```

B. Move Tree

N[5]
N[16]
N[9]
N[12]
W[1]
W[3]
W[0]

C. Metric Output for Block main$4

No. of Instructions : 10
No. of reads : 1
No. of writes : 4
Max. Fan-Out : **4**

Figure 8. An Example of the Maximum Fan-out Metric Calculation

Average of the maximum Fan-Out length over all the dynamic blocks

Legend: Average of the max. fan-out length/block; Average # of Instructions/Block; Worst block with maximum fan-out

	Average of the max. fan-out length/block	Average # of Instructions/Block	Worst block with maximum fan-out
MatrixMultiply	1.72	18.88	8
Ackermann	2.1	7.89	6
Binary Search	1.04	7	6
FFT	0.67	10.53	22
Factorial	2.25	7.98	6
Fibonacci	2.62	8.16	6
a_number()	1.86	5.86	6
longest_match()	1.43	6.67	6

Figure 9. Length of maximum fan-out

By looking at the above result, we observe that on average over all the dynamic blocks, the largest amount of transfer of a datum is up to 2 target nodes per block. A better measure of the total fan-out of data in a block is the number of move instructions executed in a block. The move instructions are used to transfer data between nodes and hence correspond to the fan-out of all the data in a block. The graph in Figure 10 generated by the next metric represents average number of move instructions per dynamic block.

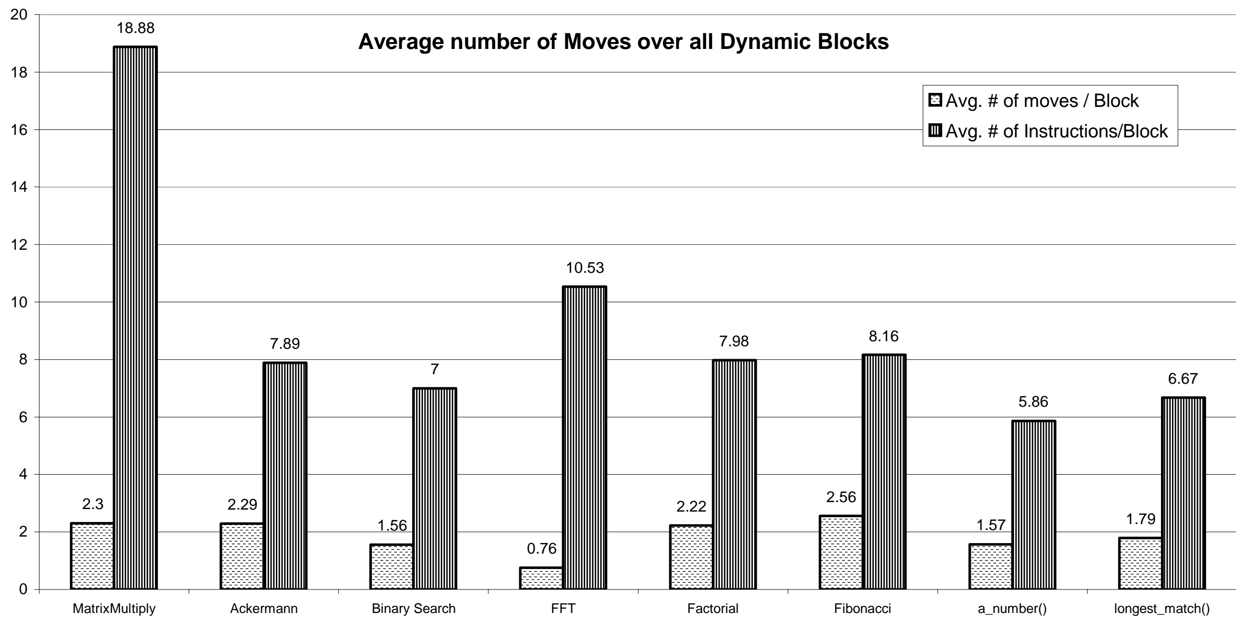


Figure 10. Dynamic average of moves per block

3.4 Useful Instructions and NOPs

Lastly, there is also a need to measure the overall quantity of useful instructions executed per TRIPS block. This metric measures the average number of non-NOP, useful instructions fetched in a TRIPS dynamic block. This result can be matched with the capacity of the TRIPS block to see the fraction of the execution nodes that are utilized and calculate the number of NOPs present in a block. This metric will also act as an important guideline in measuring the efficiency of the implementation of hyperblock formation in the future. One can expect to see a far greater usage of the TRIPS block's capacity with hyperblock formation. Figure 11 below shows the average number of useful instructions and NOPs present per dynamic block. Finally, the result of Figures 6-11 are summarized in table 5 below.

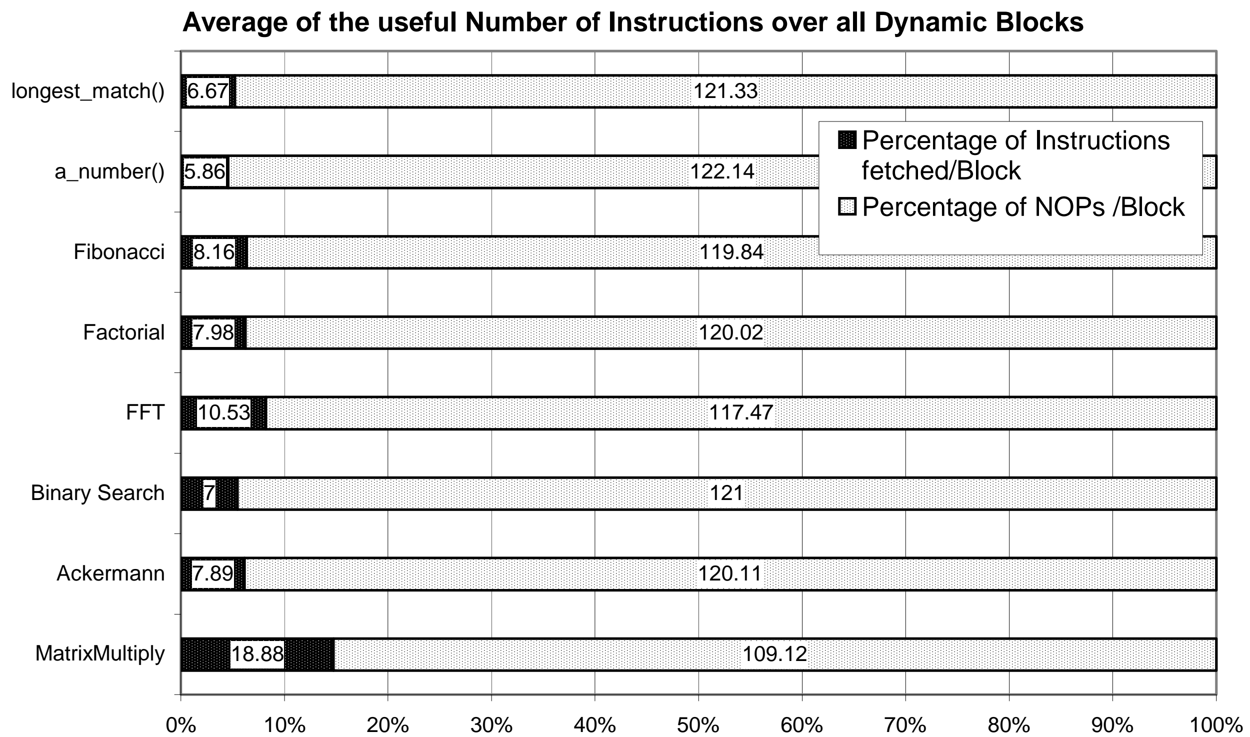


Figure 11. Dynamic average of useful instructions and NOPs per block

Table 5. Summary of the average results over all dynamic blocks

Programs	Averages (per Block)						
	Reads	**Writes**	**Critical path length**	**Max. fan-out**	**Moves**	**Instructions**	**NOPs**
Matrixmultiply	5.45	1.40	8.29	1.72	2.30	**18.88**	109.12
Ackermann	2.39	2.20	2.99	2.10	2.29	**7.89**	120.11
Binary search	2.21	1.26	4.99	1.04	1.56	**7.00**	121.00
FFT	3.86	2.84	2.40	0.67	0.76	**10.53**	117.47
Factorial	2.24	2.25	3.00	2.25	2.22	**7.98**	120.02
Fibonacci	2.14	2.18	3.10	2.62	2.56	**8.16**	119.84
a_number()	1.14	1.57	2.86	1.86	1.57	**5.86**	122.14
longest_match()	1.58	1.43	3.71	1.43	1.79	**6.67**	121.33

4 TRIPS versus Alpha

With the advent of every new architectural model comes the necessity to analyze and compare it with existing models. For a novel, architecture such as TRIPS, it becomes even more important to evaluate it effectiveness. This process of evaluation requires not only defining the advantages and disadvantages of the architecture but also comparing its efficiency with respect to an already existing architecture. Having already described the various features and specific configurations of the TRIPS model, this section will focus on comparing the outputs of the TRIPS architecture to that of an Alpha 21264 RISC architecture.

4.1 Why the Alpha 21264?

The Alpha 21264 is 64-bit, load/store RISC architecture machine that mainly increases performance via clock speed, multiple instruction issue and multiple processors [9]. The Alpha 21264 is a superscalar microprocessor with multiple fetch capability, out of order execution and speculative execution to maximize performance [10].

The Alpha machine has received widespread acceptance and recognition in the research community along with its commercial success. The bias towards the Alpha machine in this thesis can also be credited to the availability of tools and resources like the SimpleScalar toolset [6]. The SimpleScalar toolset is a system software infrastructure that can emulate the alpha platform and is used in building modeling applications for program performance analysis.

The results on the Alpha side of this thesis have been generated with the help of the SimpleScalar toolset, in particular the sim-profile program (which is a program profiling simulator) [6] and a parsing script. The script parses through the Alpha assembly language file to gather static information, just like the static.pl script on the TRIPS side. The source programs have been compiled using the default configurations of the same Scale compiler on the Alpha machine.

4.2 Expectations

The results of the comparisons of outputs on the two machines are discussed later. But before discussing the results, it becomes imperative to discuss the expected behavior of the source program on the two machines. Establishing the expected behavior will also gives us a better foresight while analyzing the results of the metrics.

Firstly, let's describe the metrics of comparison. I compare the efficiency of the TRIPS architectural model to Alpha model based on two factors, **(1) code size**, and **(2) instruction mixes**. Comparing the total number of instructions and the type of instructions executed on the two machines for a given program will give us a fair idea of the efficiency of the TRIPS model normalized to the Alpha machine.

Keeping the grid processor architecture and the block-atomic execution of the TRIPS model in mind, we can assume that the TRIPS output will result in having fewer accesses to the register files and more transfer of data between the execution nodes. We can certainly expect to see a greater number of move instructions on the TRIPS execution side. The transfer of data between the execution nodes in the blocks also introduces the cost of extra overhead instructions. These overhead instructions coupled with the room for necessary improvements in the Scale compiler and scheduler and the future implementation of hyperblocks will cause the TRIPS code size to be generally bigger than the Alpha code size. We can also expect the current version of the TRIPS code to have more conditional branches since each TRIPS block ends with a branch offset instruction that branches to the next executable block. Again, with the formation of hyperblocks, we can expect this effect to reduce as predication will eliminate theses conditional branches. We should also expect to see a better and fairer representation with the dynamic code size and instruction mix.

4.3 Static and Dynamic Code Sizes

The first metric of comparing the TRIPS and Alpha machine simply deals with looking at the code size generated for the source programs by the two machines. The code size can be measured both statically (code generated by the compiler) and dynamically (code executed by the simulator). The static code size gives us some insight into the overhead instructions and block-atomic execution model whereas the dynamic code size directly deals with the efficiency of the architectural model.

The graph comparing the static code size of the source programs has been shown in Figure 12 below. As expected, we see the TRIPS code size to be slightly bigger than the Alpha code size except in the case of the FFT program in which the TRIPS code size is about 70% of the Alpha code size.

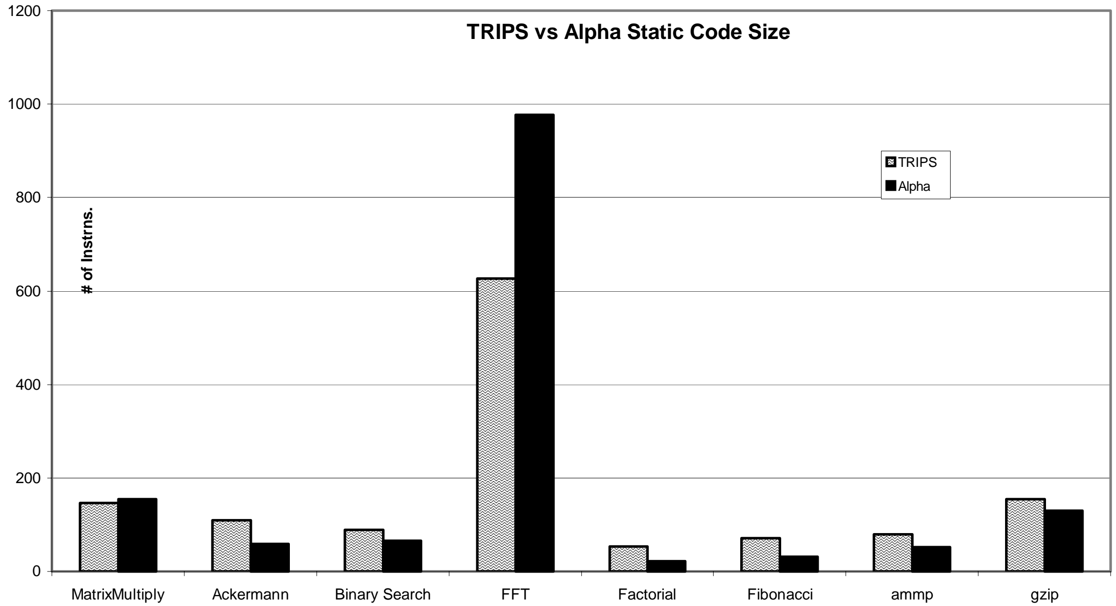
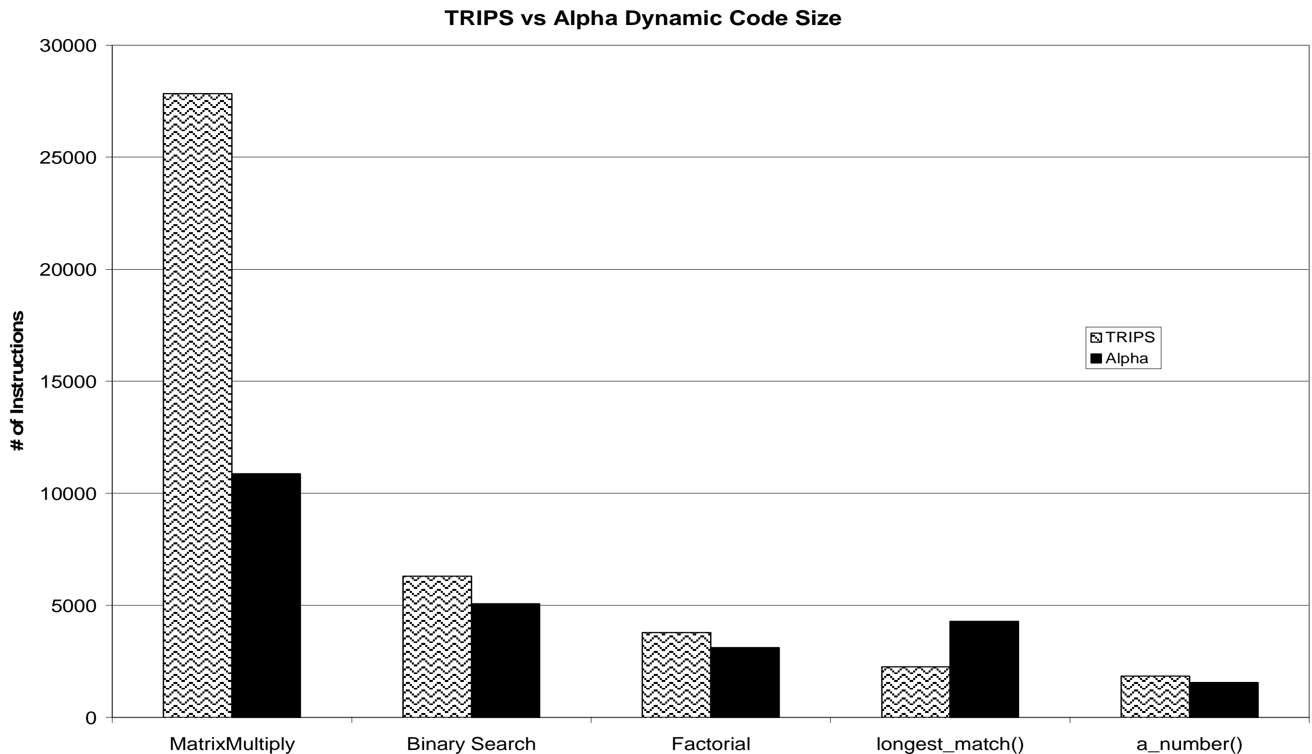


Figure 12. Static code sizes on the TRIPS and Alpha machines

Figure 13 below represents the TRIPS versus Alpha dynamic code sizes for the given source programs. The results would indicate that the Alpha machine seems to be more efficient than the TRIPS model. However, it must be kept in mind that the TRIPS model is still in development and with progress in the future its efficiency is bound to increase.



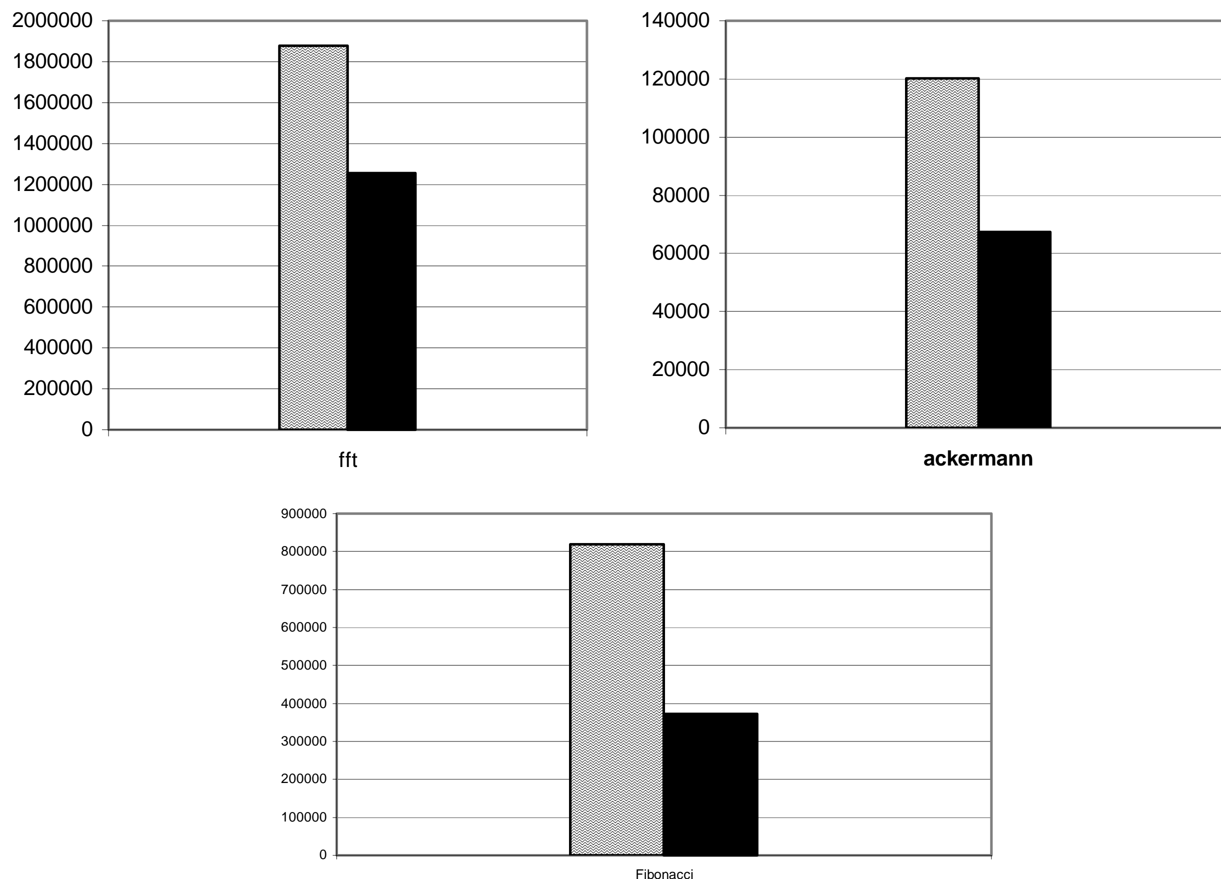


Figure 13. Dynamic code sizes on the TRIPS and Alpha machines

Looking at the dynamic code size in Figure 13, we again observe the TRIPS code to be bigger than the Alpha code size. Running the complete program on the simulator gives us a fairer idea of the execution code size. The results seem to match our expectation except for the case of the longest_match () function from gzip, where the TRIPS dynamic code size is bigger than the Alpha dynamic code size. I have so far been unable to explain the reason for this anomaly. Considering all the other programs, the TRIPS code size seems to vary between 20% and 150% bigger than the size of the Alpha code. Table 6 below lists out the percentage by which the TRIPS code is bigger than the Alpha code for the given programs.

Program	% Bigger than Alpha
Matrixmultiply	155.91%
Ackermann	78.62%
Binary search	24.27%
FFT	49.61%
Factorial	21.14%
Fibonacci	120.20%
a_number()	18.60%
longest_match()	-16.5%

Table 6. Percentage by which TRIPS dynamic code is bigger than the Alpha code

4.4 Instruction Mixes

The second metric used for comparing the two architectural models measures the various types of instructions that are executed on the two machines. Comparing the dynamic instruction mixes of the two platforms seems more logical and relevant; hence, the data for the static instruction mixes have been skipped. The dynamic instruction mix classifies the instructions executed by the simulator into six different categories depending on their function.

For the TRIPS side, the instruction mix is divided into the following categories:
(a) Arithmetic/Logical, (b) Load, (c) Store, (d) Conditional branch, (e) Move and (f) Others.

For the Alpha machine, the instruction mix is generated for the following categories by sim-profile:
(a) Arithmetic/Logical, (b) Load, (c) Store, (d) Conditional branch, (e) Unconditional branch and (f) Others.

Thus, the only difference in the categories is the move (TRIPS) and unconditional branches (Alpha). Since the move instruction plays an important role in the TRIPS architecture, it seems logical to display it as a separate category for the TRIPS instruction mix.

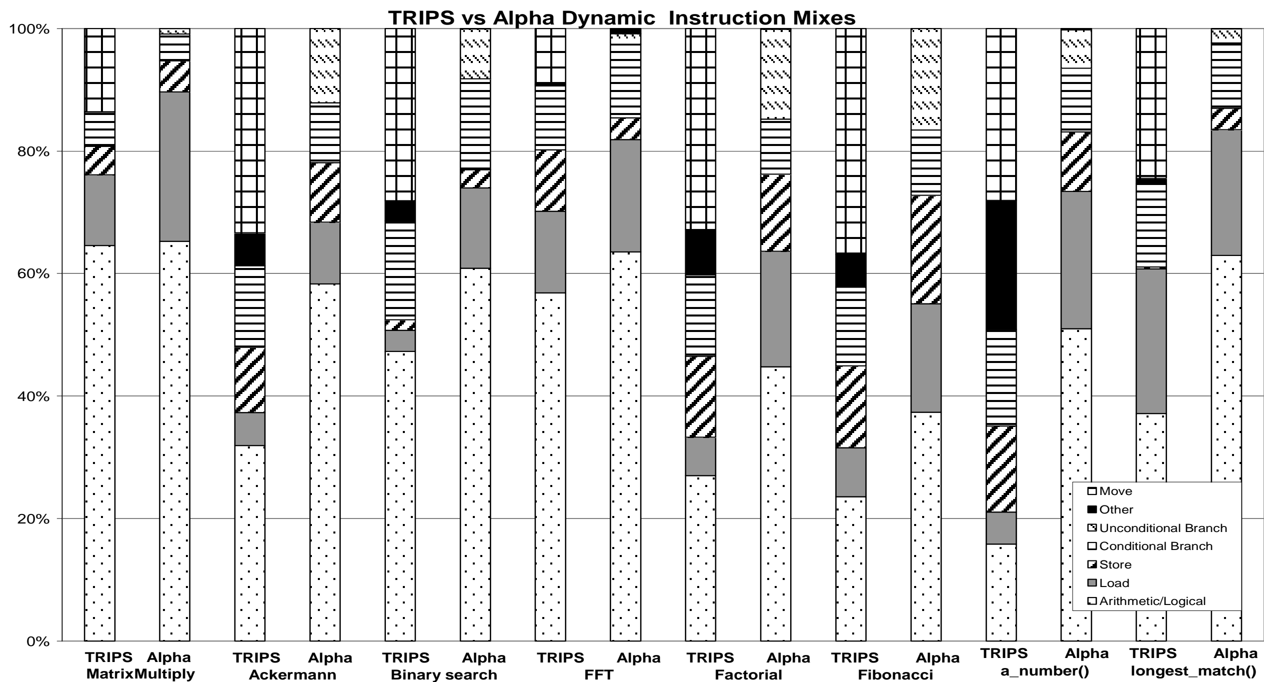


Figure 14. Dynamic instruction mixes for the TRIPS and Alpha machines

By looking at the results from Figure 14, it becomes noticeable that the results are as we expected. For most of the programs, the percentage of moves in the TRIPS instruction mix is nearly twice that of the unconditional branches in the Alpha side. The percentage of arithmetic, logical instructions, loads and stores seems to be about the same for both the models whereas the TRIPS

side seems to have a little bigger percentage of conditional branches than the Alpha instruction mix. The TRIPS code also has a higher percentage of other instructions like (genu, app) constants, which are used to generate unsigned constants etc. that are not present in the Alpha ISA. With hyperblock formation and predication, we should also expect the number of branches to decrease in the TRIPS instruction mixes.

Furthermore, Figure 14 does not include the relative difference between the total code size of the TRIPS and Alpha code. Hence, a category in the above graph having the same percentage for the TRIPS and Alpha side does not necessarily guarantee that the total number of instructions of that type would be the same. This calls for combining the size of the dynamic code with the instruction mix. By adding the total code size information, the actual number of instructions in a category can be observed. Figure 15 below represents the instruction mixes along with the total code size for four of the biggest programs that were run on the TRIPS and Alpha side.

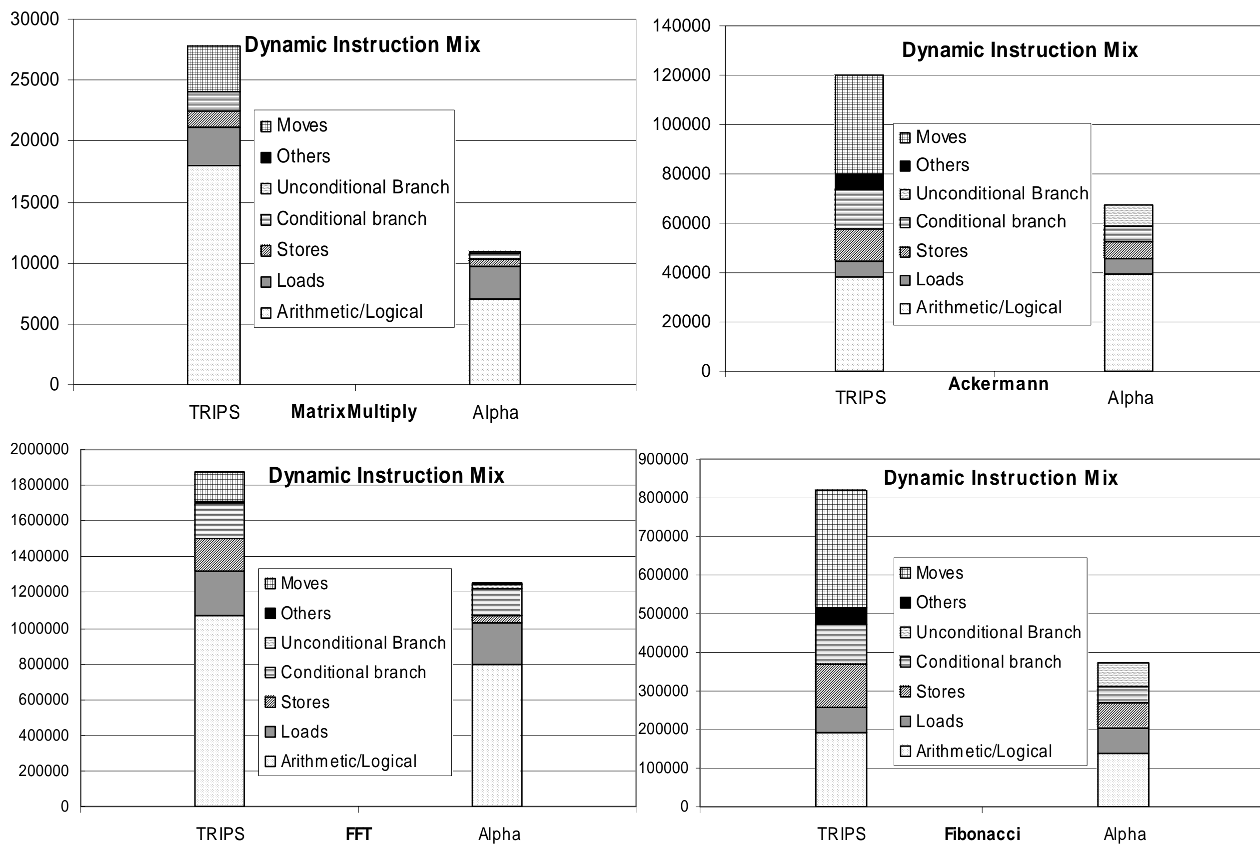


Figure 15. Dynamic instruction mixes with total code size

It becomes clearer from the above graphs that the number of loads, arithmetic and logical instructions executed, are about the same in both the machines. The TRIPS side executes marginally more conditional branches and stores. And, the move instructions on the TRIPS side are at least twice the number of unconditional branches executed in the Alpha side.

5 Conclusions

5.1 Conclusion

This thesis introduces the methodology of the tools and scripts (static.pl, dynamic.pl) that were developed to analyze the TRIPS architecture. It then reports the results of the metrics that were specific to TRIPS features (such as the average length of critical path instructions, fan-out effect and average size of a TRIPS block, etc). Finally, the thesis compares the output of the TRIPS model with the Alpha 21264 for a set of common source code. The comparisons of the size of the code (code produced by the compiler as well the code executed) and the instruction mixes for the source programs are reported for the two machines.

The results in this thesis for the TRIPS-specific features, and the comparisons of the two machines are shown below. The specific numbers reported in the thesis should not be used as a source to determine the final efficiency of the TRIPS architecture. The results are shown only because they will help create a foundation based on which future analysis of the TRIPS architecture would be possible. The scripts and the results also act as the initial setup and values that will guide future researchers in the TRIPS group.

The table below summarizes the TRIPS-specific metrics:

Filename	Averages (per Dynamic Block)					
	Read	**Write**	**Critical path length**	**Max. fan-out**	**move**	**Instruction**
Matrixmultiply	5.45	1.40	8.29	1.72	2.30	**18.88**
Ackermann	2.39	2.20	2.99	2.10	2.29	**7.89**
Binary search	2.21	1.26	4.99	1.04	1.56	**7.00**
Fft	3.86	2.84	2.40	0.67	0.76	**10.53**
Factorial	2.24	2.25	3.00	2.25	2.22	**7.98**
Fibonacci	2.14	2.18	3.10	2.62	2.56	**8.16**
A_number()	1.14	1.57	2.86	1.86	1.57	**5.86**
longest_match()	1.58	1.43	3.71	1.43	1.79	**6.67**

The tables lists the average numbers of read and write accesses in a TRIPS block, average length of the critical path of instructions in a TRIPS block, average number of nodes to which fan-out of a value occurs in a TRIPS block as well as the average number of moves and useful instructions in a TRIPS block. These results are not meant to criticize or praise the TRIPS work in progress but simply represent the output of the framework for analysis that I have created. These results will become more important in the future when the TRIPS project reaches its completion. The thesis must only be considered as an intermediate checkpoint in the analysis of the TRIPS architecture. The tools introduced in the thesis provide an infrastructure to the architecture analysis process whereas the results shown in the thesis are references for future researchers to compare and calculate the effectiveness of their implementations and optimizations to the TRIPS architecture.

The comparison of code size and instruction mixes between TRIPS and Alpha, also follow the same principle. The initial results note that the TRIPS dynamic code size is expectedly larger than the Alpha code size and the TRIPS instruction mixes include nearly twice the number of moves as compared to the unconditional branches in the Alpha mix. The numbers of arithmetic, logical instructions, loads and stores are proportional and TRIPS has slightly greater number of conditional branches and other instructions (such as constant instructions [5]).

5.2 Future Work

Since this thesis was the first attempt at analyzing and marking the efficiency of the TRIPS architecture, a huge scope for future development is possible. With further progress in the implementation of the TRIPS architecture to the prototype, the scripts have to be developed to include more specific features of the TRIPS model. The output of the scripts has to be modified to return user-friendly and specific pieces of information. Most importantly, as the development of the TRIPS toolchain progresses, heavier workloads like the SPEC benchmarks should be used with the metrics to obtain new results that span a larger source domain. The scripts should also be periodically modified and maintained up to date with the changes in the TRIPS architecture, for e.g.: the scripts had to be modified to parse the new (tsim_arch) architectural simulator's output correctly when it replaced the earlier (tem) TRIPS toolchain emulator.

I hope that TRIPS researchers will find these tools and metrics useful and frequently use it to generate results to evaluate the progress of the TRIPS project and develop better analysis tools.

References

[1] TRIPS Project homepage: http://www.cs.utexas.edu/users/cart/trips.

[2] R. Nagarajan, K. Sankaralingam, D. Burger and S.W. Keckler. A Design Space Evaluation of Grid Processor Architectures. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 177-189, December 2001.

[3] D. Burger, S.W. Keckler, M.D. Dahlin, L.K. John, C. Lin, K.S. McKinley, C.R. Moore, J. Burrill, R.G. McDonald and W. Yoder. Scaling to the end of Silicon with EDGE Architecture. In *IEEE Computer*, in press, May 2004.

[4] K. Sankaralingam, R. Nagarajan, H. Liu, C.K. Kim, J. Huh, D. Burger, S.W. Keckler, and C.R. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 422-433, June 2003.

[5] R. McDonald. TRIPS Processor Architecture Manual, version 0.2. In *TRIPS Internal Documents*, pages 1-168, April 2004.

[6] D. Burger and T.M. Austin. The SimpleScalar Tool Set, Version 2.0. In *Computer Architecture News, 25 (3)*, pages 13-25, June 1997.

[7] L. Shustek. Analysis and performance of computer instruction sets. In *Stanford Linear Accelerator Center Report 205*, *Stanford University*, pages 56-82, May 1978.

[8] W.C. Alexander and D.B. Wortman. Static and Dynamic Characteristics of XPL Programs. In *Computer, Volume 8, Number 11,* pages 41-46, November 1975.

[9] Alpha Architecture Handbook, version 4. Compaq Computer Corporation, July 1999.

[10] R. E. Kessler. The Alpha 21264 Microprocessor. In *IEEE Micro 19(2)*, pages 24-36, March 1999.

[11] W. Yoder. TRIPS Assembly Language (TASL) Specification, version 2.1D. In *TRIPS Internal Documents*, pages 1-29, September 2003.

[12] D.A. Patterson and D.R. Ditzel. The Case for the Reduced Instruction Set Computer. In *Computer Architecture News, 8(6)*, pages 25-33, October 1980.

[13] R.P. Colwell, C.Y. Hitchcock III, E.D. Jensen, H.M. Brinkley Sprunt and C.P.Kollar, Instruction Sets and Beyond: Computers, Complexity, and Controversy. In *IEEE Computer, 18(9)*, pages 144-155, June 1985.

Appendix

A. Location of the code metrics

The scripts for the code metrics (static.pl, dynamic.pl) and all the other source files and tools used in this thesis report along with their outputs have been saved in the CVS repository. They are available for all TRIPS research group members to use by checking out from the repository. The files are stored under the following directory: **tsrc/CodeMetrics/tester** in the CVS repository.

Use the following command to check out the files from the CVS repository:

> *cvs –d /projects/trips/cvs checkout tsrc/CodeMetrics/tester*

Note: In order to run the Alpha side of the metrics, the user must also have access to the sim-profile tool under the SimpleScalar toolset.

B. Usage of the code metrics

The script for static code metrics, static.pl, requires a TASL assembly file (*.s) as an input file while the dynamic metrics script, dynamic.pl, requires the TASL assembly file and the TRIPS architectural simulator's trace file (tsim_arch.tt) as inputs. Given below are the steps to use the code metrics on a given C source file, once they have been checked out of the CVS repository. One can also apply similar steps to collect metrics for hand-generated TIL or TASL code.

- **Step 1: Run the C source on the TRIPS toolchain**

> *tcc –save-temps [compiler options] matrixmultiply.c –o mm.out*

- **Step 2: Simulate the executable and create a trace file**

> *tsim_arch –tt mm.out*

- **Step 3: Run the script**

> *perl dynamic.p l –f matrixmultiply.s –t tsim_arch.tt*

C. Sample Output

For a given C file, 10x10matrixmultiply.c , the output of the dynamic.pl script would look like the following:

$ perl dynamic.pl –f matrixmultiply.s –t tsim_arch.tt

**** Overall result in Verbose Mode ****

Block Name	Instructions	Frequency	Read	Write	Critical Path	Max. Fan-Out
MatrixMultiply$7	24	1000	7	1	11	2
Main	19	1	2	5	8	8
MatrixMultiply	12	1	4	5	4	4
MatrixMultiply$2	11	100	3	1	5	2
_start	10	1	1	1	3	0
main$4	10	1	1	4	3	4
main$5	10	1	1	3	4	2
main$3	7	100	4	3	2	0
main$2	7	1	1	4	2	0
main$1	7	100	4	3	2	0
MatrixMultiply$9	6	10	1	1	5	2
MatrixMultiply$3	6	10	1	1	5	2
MatrixMultiply$8	6	100	1	1	5	2
MatrixMultiply$6	5	100	2	3	3	2
MatrixMultiply$1	3	10	1	2	1	0
MatrixMultiply$10	2	1	2	1	1	0
MatrixMultiply$5	2	10	0	1	1	0
MatrixMultiply$4	2	1	0	1	1	0
_exit	2	1	0	1	1	0

```
  0-  9 instructions in   444 blocks
 10- 19 instructions in   105 blocks
 20- 29 instructions in  1000 blocks
 30- 39 instructions in     0 blocks
 40- 49 instructions in     0 blocks
 50- 59 instructions in     0 blocks
 60- 69 instructions in     0 blocks
 70- 79 instructions in     0 blocks
 80- 89 instructions in     0 blocks
 90- 99 instructions in     0 blocks
100-109 instructions in     0 blocks
110-119 instructions in     0 blocks
120-129 instructions in     0 blocks

Number of Distinct Blocks      : 19
Total No. of Blocks executed  : 1549
Total No. of Reads             : 8442
Total No. of Writes            : 2175
```

Total Dynam Instrctns fetched : 29252
Total Dynam Instrctns executd: 27844
Total no. of NOPs : 170428
Avg. Reads/Block : 5.45
Avg. Writes/Block : 1.40
Avg. Instructions/Block : 18.88
Avg. CriticalPath lngth/block : 8.29
Avg. Crit.fan-out lngth/blk : 1.72
Avg. NOPs/Block : 110.02

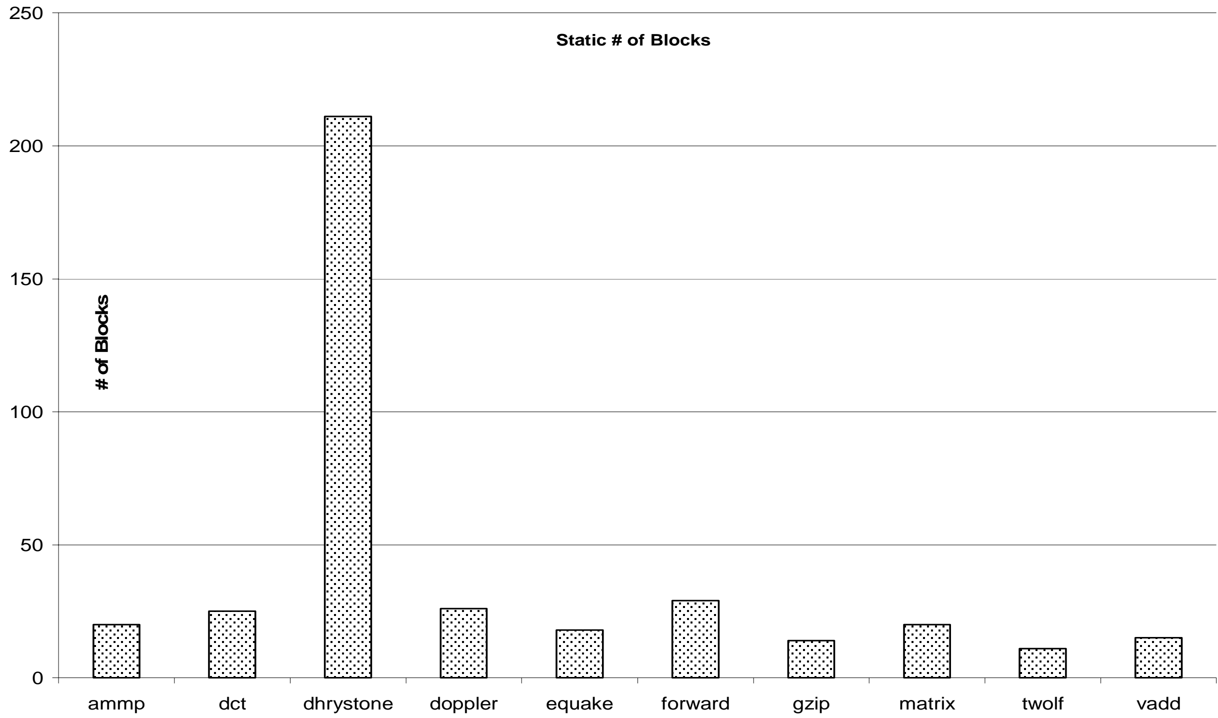
add	6302	22.63%
mov	3557	12.77%
extsw	3221	11.57%
slli	3100	11.13%
lws	3000	10.77%
addi	1832	6.58%
bro	1545	5.55%
sw	1300	4.67%
tlti	1220	4.38%
muli	1110	3.99%
mul	1000	3.59%
movi	229	0.82%
lw	200	0.72%
tltu	200	0.72%
genu	10	0.04%
sd	7	0.03%
app	6	0.02%
scall	3	0.01%
ret	2	0.01%
callo	1	0.00%
ld	1	0.00%

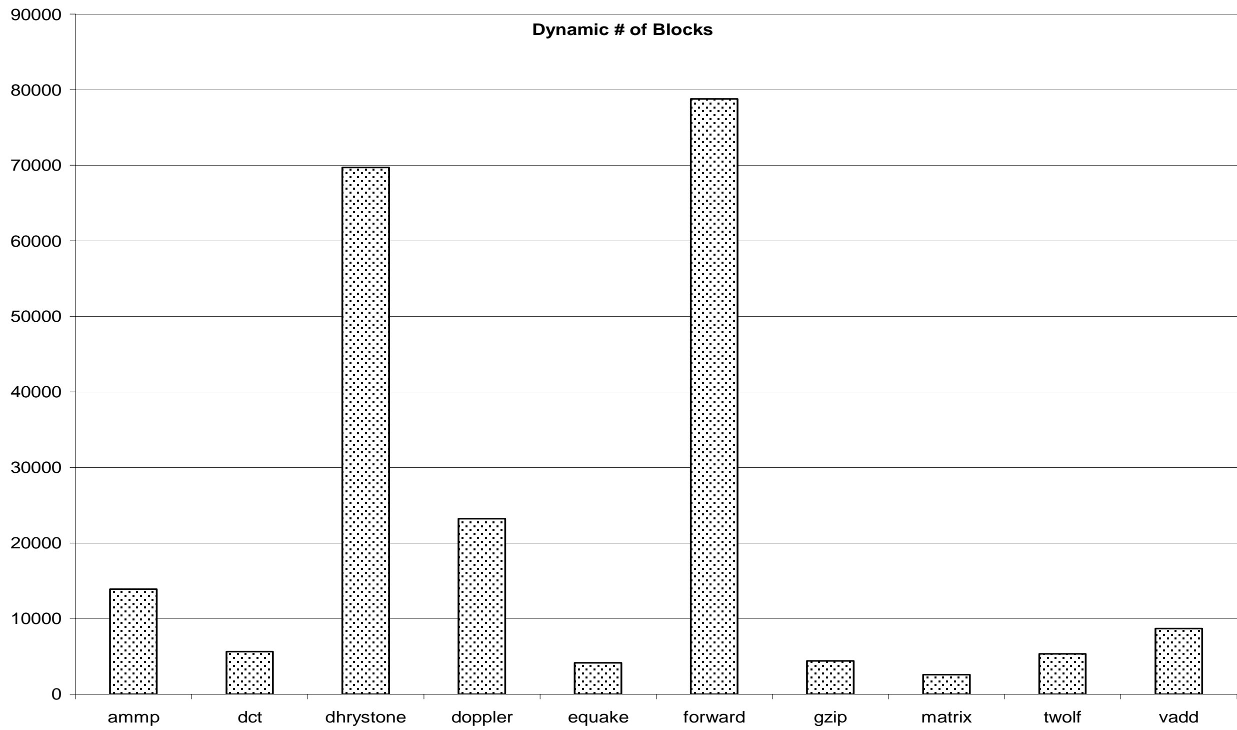
Total: 27844 Instructions

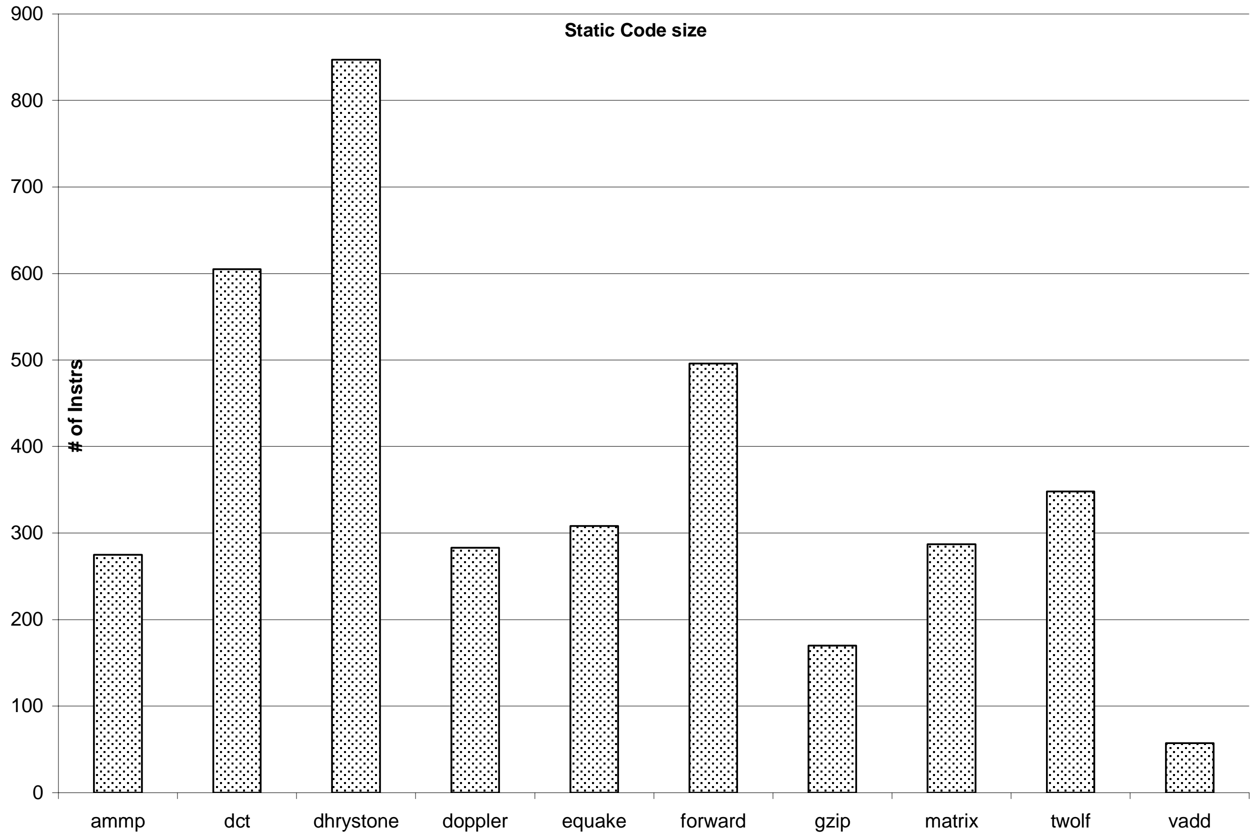
D. Micro-benchmarks Metrics

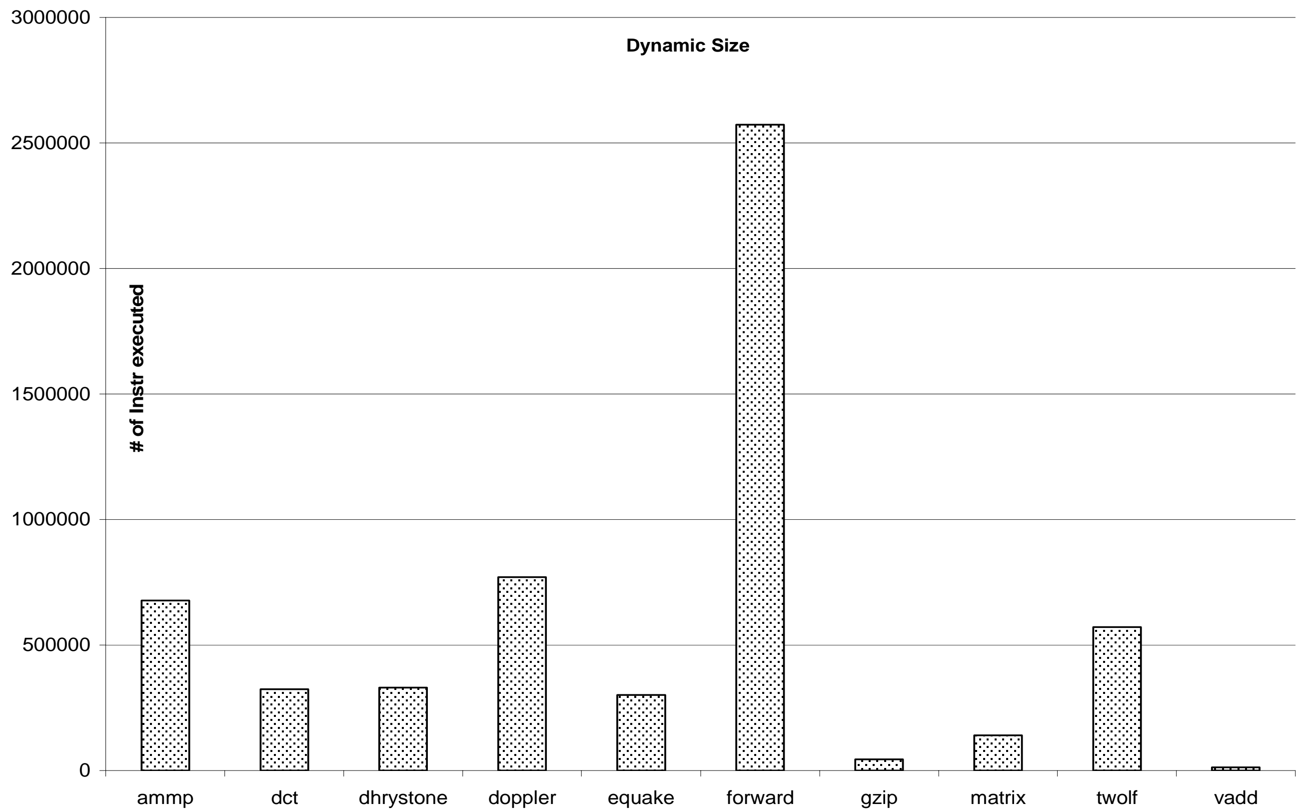
The TRIPS-specific metric scripts were also run on micro-benchmarks, most of which included hand-generated TIL or TASL code. The results below represent a summary of these metrics. Only the benchmarks that executed to completion with normal exit status were used for this experiment. They are as follows:

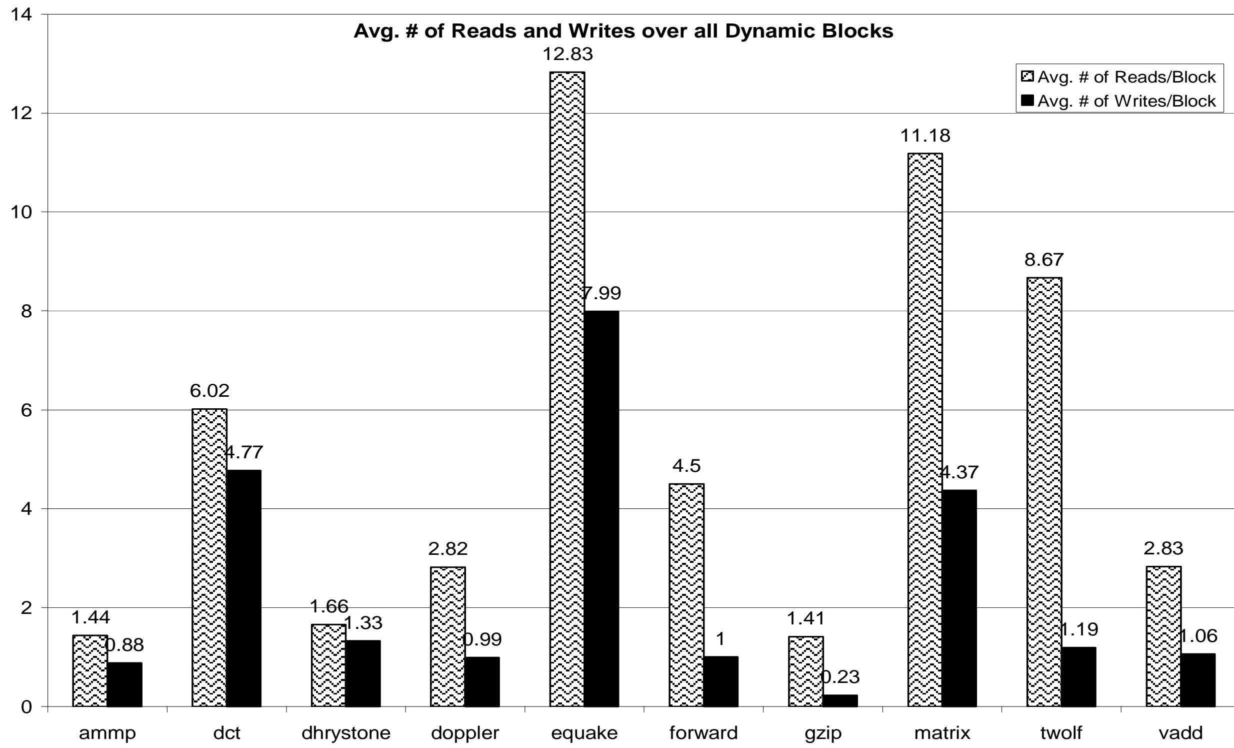
Benchmarks	Source Type
ammp	Hand-generated TIL code
dct	C source code
dhrystone	C source code
doppler	C source code
equake	Hand-generated TIL code
forward	C source code
gzip	C source code
matrix	Hand-generated TIL code
twolf	Hand-generated TIL code
vadd	Hand-generated TASL code

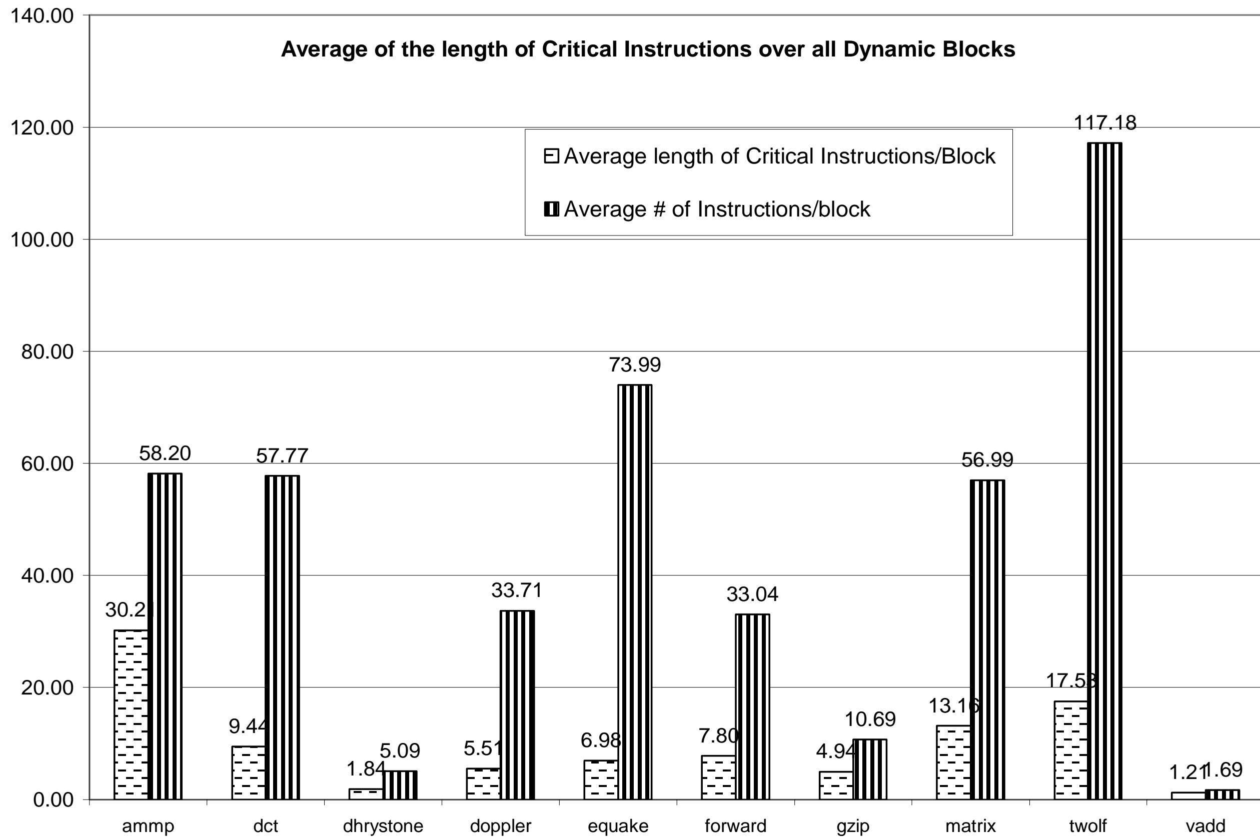

Static # of Blocks
# of Blocks
250
200
150
100
50
0
ammp
dct
dhrystone
doppler
equake
forward
gzip
matrix
twolf
vadd

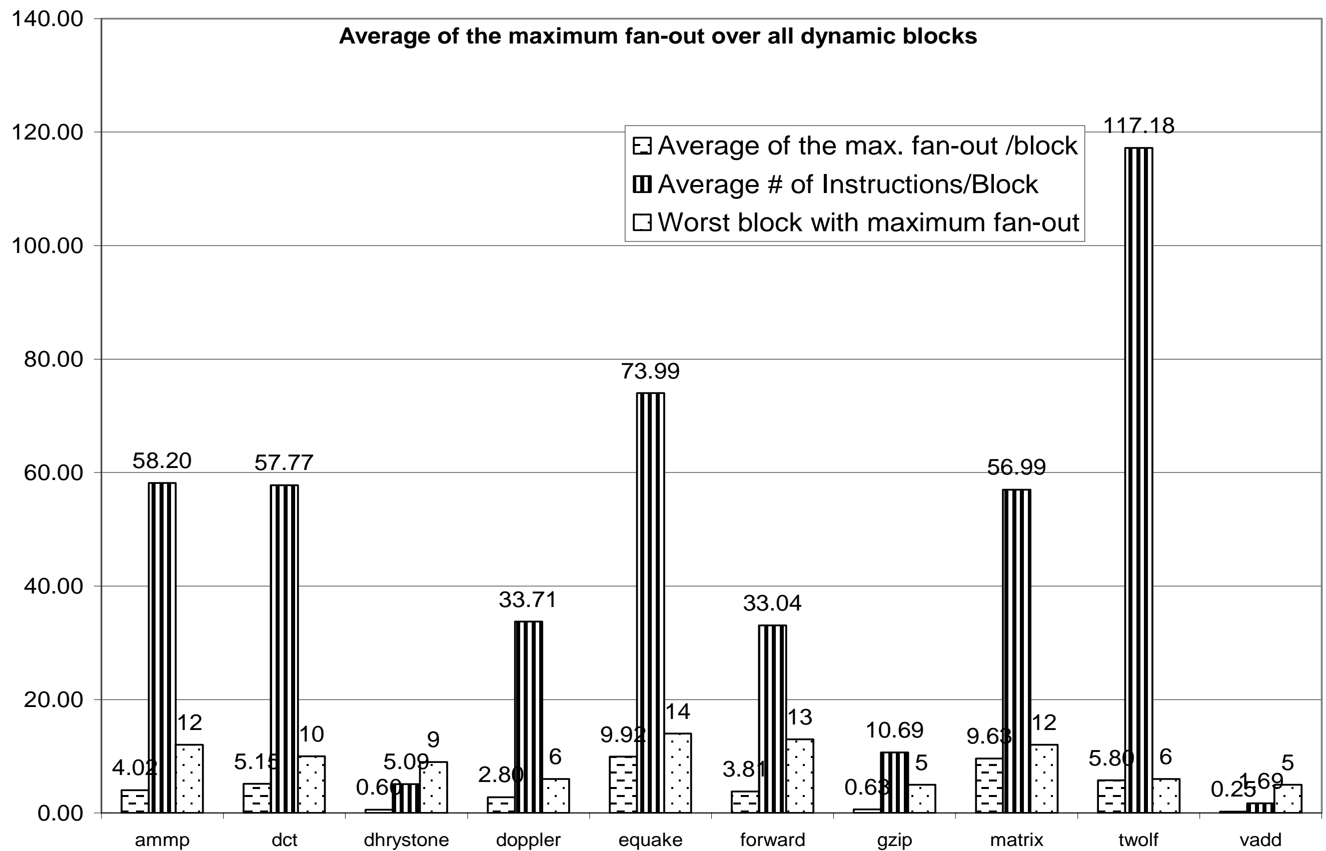

Dynamic # of Blocks
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
ammp
dct
dhrystone
doppler
equake
forward
gzip
matrix
twolf
vadd

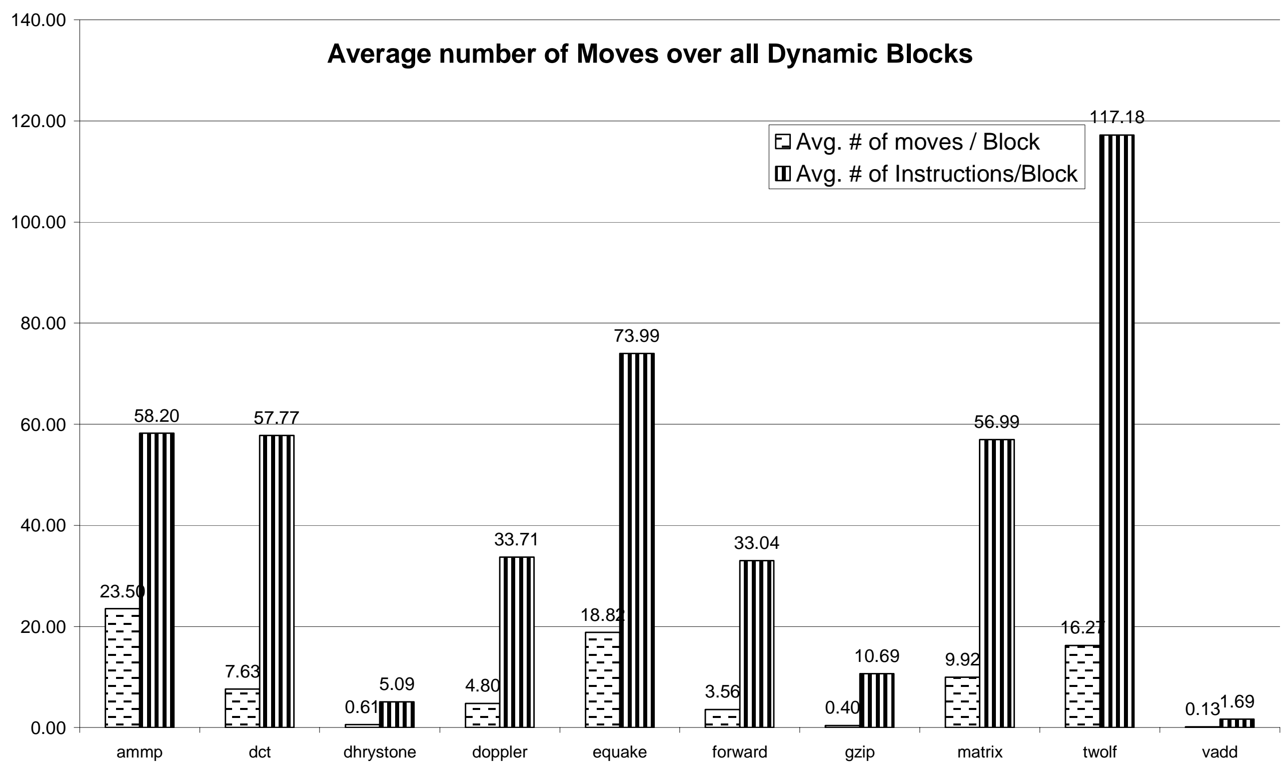

Static Code size
# of Instrs
900
800
700
600
500
400
300
200
100
0
ammp
dct
dhrystone
doppler
equake
forward
gzip
matrix
twolf
vadd

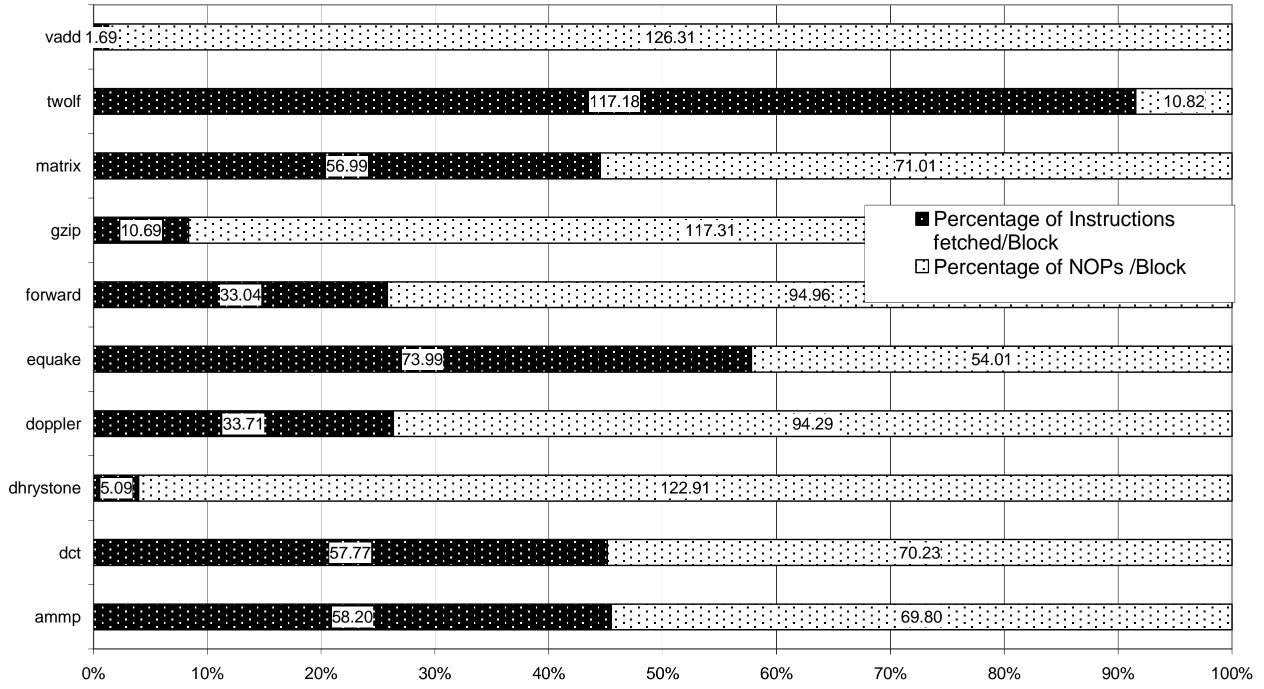

Dynamic Size
# of Instr executed
3000000
2500000
2000000
1500000
1000000
500000
0
ammp
dct
dhrystone
doppler
equake
forward
gzip
matrix
twolf
vadd


Avg. # of Reads and Writes over all Dynamic Blocks
Avg. # of Reads/Block
Avg. # of Writes/Block
14
12
10
8
6
4
2
0
1.44
0.88
6.02
4.77
1.66
1.33
2.82
0.99
12.83
7.99
4.5
1
1.41
0.23
11.18
4.37
8.67
1.19
2.83
1.06
ammp
dct
dhrystone
doppler
equake
forward
gzip
matrix
twolf
vadd


Average of the length of Critical Instructions over all Dynamic Blocks
Average length of Critical Instructions/Block
Average # of Instructions/block
140.00
120.00
100.00
80.00
60.00
40.00
20.00
0.00
30.2
58.20
9.44
57.77
1.84
5.09
5.51
33.71
6.98
73.99
7.80
33.04
4.94
10.69
13.16
56.99
17.58
117.18
1.21
1.69
ammp
dct
dhrystone
doppler
equake
forward
gzip
matrix
twolf
vadd


Average of the maximum fan-out over all dynamic blocks
Average of the max. fan-out /block
Average # of Instructions/Block
Worst block with maximum fan-out
140.00
120.00
100.00
80.00
60.00
40.00
20.00
0.00
58.20
57.77
33.71
73.99
33.04
56.99
117.18
4.02
12
5.15
10
0.60
5.09
9
2.80
6
9.92
14
3.81
13
0.63
10.69
5
9.63
12
5.80
6
0.25
1.69
5
ammp
dct
dhrystone
doppler
equake
forward
gzip
matrix
twolf
vadd


Average number of Moves over all Dynamic Blocks
Avg. # of moves / Block
Avg. # of Instructions/Block
140.00
120.00
100.00
80.00
60.00
40.00
20.00
0.00
23.50
58.20
7.63
57.77
0.61
5.09
4.80
33.71
18.82
73.99
3.56
33.04
0.40
10.69
9.92
56.99
16.27
117.18
0.13
1.69
ammp
dct
dhrystone
doppler
equake
forward
gzip
matrix
twolf
vadd


Average of the useful Number of Instructions over all Dynamic Blocks
vadd
1.69
126.31
twolf
117.18
10.82
matrix
56.99
71.01
gzip
10.69
117.31
forward
33.04
94.96
equake
73.99
54.01
doppler
33.71
94.29
dhrystone
5.09
122.91
dct
57.77
70.23
ammp
58.20
69.80
0%
10%
20%
30%
40%
50%
60%
70%
80%
90%
100%
Percentage of Instructions fetched/Block
Percentage of NOPs /Block