Characterizing Function Inlining

with Genetic Programming

by

Chris Yu

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical [Computer] Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

August 17, 2004 [September 2004]

Author_____
Department of Electrical Engineering and Computer Science
August 17, 2004

Certified by_____
Saman P. Amarasinghe
Thesis Supervisor

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

Characterizing Function Inlining with Genetic Programming
by
Chris Yu

Submitted to the
Department of Electrical Engineering and Computer Science

August 17, 2004

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

ABSTRACT

Function inlining is a compiler optimization where the function call is replaced by the code from the function itself. Using a form of machine learning called genetic programming, this thesis examines which factors are important in determining which function calls to inline to maximize performance. A number of different heuristics are generated for inlining decisions in the Trimaran compiler, which improve on performance from the current default inlining heuristic. Also, trends in function inlining are examined over the thousands of compilation runs that are completed.

Thesis Supervisor: Saman P. Amarasinghe
Title: Associate Professor, Computer Science and Artificial Intelligence Laboratory

Acknowledgments

I would like to thank my supervisor, Saman P. Amarasinghe, for taking me into his team and entrusting me with this task. His dedication and affability are contagious, as it is evident to see they are transmitted to all the other members of the research group.

A special thanks to Rodric Rabbah for offering his guidance, his knowledge, and his patience to me the whole way. Without his constant motivation to do more, this thesis would have been a shell of its current self.

Another special thanks to Mark Stephenson for the countless hours he spent helping answer all of my countless questions. This thesis would not have been possible without the groundwork he provided.

Finally, countless others throughout the lab have helped me at one point or another. Though too numerous to name, I would like to thank all the members of the Computer Science and Artificial Intelligence Laboratory.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Function inlining is a compiler optimization technique where the code from the callee (the function being called) is inserted into the caller function thereby replacing the function call. This allows the program to avoid two jumps – one into the callee function, and one out of it – and hence avoid the overhead involved in making function calls. Furthermore, inlining functions can also create larger basic blocks – units of execution without branches or jumps – enabling other optimizations in modern compilers such as register allocation and instruction scheduling.

However, there are also negative performance factors involved with function inlining. If a function has too many other functions inlined in it, its size could expand so that it clobbers the instruction cache too often, taking a performance hit larger than the benefit gained from inlining. Most compilers have quotas on the amount of codesize expansion the inlining module can produce. Therefore the opportunity cost of inlining a function that produces only a small gain, versus inlining a function that produces a much better gain must be considered as well. Choosing the right mix of functions to inline is an intractable problem, in general.

Deciding which function callsites to inline is normally done by a heuristic created by the compiler engineer, who attempts to balance which functions are inlined so that

the benefits of inlining are maximally achieved.

This thesis will (1) examine the important characteristics that a heuristic for inlining should consider and (2) automatically derive an inline heuristic for optimizing compilers. In pursuit of these goals, we will be using a form of machine learning called genetic programming on an open-source C compiler.

1.1 Motivations

Optimal solutions for many problems, such as deciding which functions to inline, require the compiler to solve intractable problems. Since compiler running times must be reasonably bounded, compiler writers are forced to create heuristic algorithms which approximate the ideal solution. One technique used in developing heuristic algorithms is priority (or cost) functions, which combine the various factors of a problem into a single number. For instance, in the Trimaran [14] implementation of function inlining, the higher the priority of a function call, the more likely it is that function will be inlined.

The efficiency of an algorithm is determined by the efficiency of the priority function. And priority functions are ubiquitous in compiler optimizations, making them an ideal candidate for *meta optimization: the optimization of the compiler's optimizer.*

However, a significant amount of time goes into the development of these priority functions: a number of candidate functions are created by the programmer and tested. This tweaking continues until the developer decides a suitable solution has been achieved. The compiler engineer becomes entangled in the time-consuming process of guess-and-test.

As computer architectures evolve and increase in complexity, compilers designed to utilize those architectures also increase in complexity. The process of hand-tuning priority functions in these compilers quickly becomes infeasible as the number and

complexity of the interrelated optimizations increases. However, machine learning techniques can be used to substitute computing resources for human effort.

Genetic programming is one approach to using machine learning for finding suitable priority functions. Its operation is based loosely on Darwinian evolution: each generation the worst results are killed off, and the population is replenished by crossovers and mutations among the remaining results.

The results obtained from the experiments will be examined across a variety of different criteria. One goal is to develop further insight into function inlining performance, and its function space for priority functions.

1.2 Contents

The rest of the thesis is organized in the following manner. Chapter 2 describes the benefits of genetic programming, and the Finch framework used for machine learning. Chapter 3 describes function inlining, and how it is implemented in the Trimaran compiler. Chapter 4 describes the implementation of the compiler system used to evolve the inlining priority function. Chapter 5 describes the benchmark testing methodology, and examines the results gathered. Chapter 6 describes related work. And Chapter 7 concludes the thesis, and discusses future work.

Chapter 2

Finch framework

This chapter gives some background on genetic programming and describes the framework used for applying machine learning techniques in the context of meta optimization.

2.1 Genetic Programming

Genetic programming can be used in an unsupervised manner to search large expression spaces. Its operation is loosely based on Darwinian evolution: random function expression trees are created and tested, with crossover combinations occurring between the fittest expressions forming new expressions to test.

Figure 2-1 shows the genetic programming flow. Genetic programming first randomly creates an initial population of expressions. Each expression is executed and assigned a fitness. In our case, the fitness is going to be a measure of runtime (either in seconds or in cycles) so smaller is better. A subset of the expressions are 'killed' off, and replaced with either new random expressions, crossover expressions, or mutated expressions. Then the cycle repeats until the desired number of generations has elapsed, and a winner is chosen.

Finch [11] is a simple genetic programming framework developed by Stevenson et.

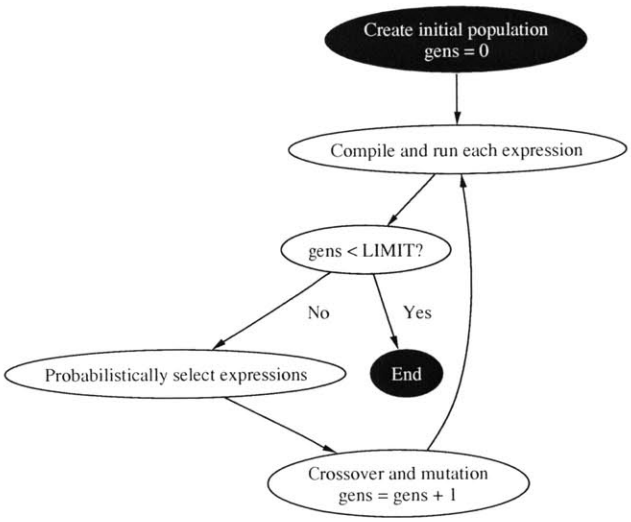

Figure 2-1: Genetic programming flowchart.

al. for compiler meta-optimization. Its design allows it to be easily integrated with a number of existing compilers, including Trimaran. Finch has an array of features very well suited for the purposes of this research, including the creation, testing, and ranking of expressions for optimization.

Genetic programming was chosen as the machine learning technique used in Finch for a multitude of reasons. Compiler optimization often involves a large number of factors, leading to large high-dimensional search spaces. Genetic programming is capable of searching these spaces, whereas many other machine learning algorithms are not as capable of scaling. Genetic programming is especially applicable when the relations between the large number of factors is relatively unknown [9].

Another important property of genetic programming is that the resulting functions generated are human-readable expression trees, which are output by the program using a Lisp-like syntax. Figure 2-2 illustrates some priority functions, and the crossover and mutation operations which create new priority functions. Post-experimental analysis of genetic programming can provide more insight into why the best priority functions perform well.

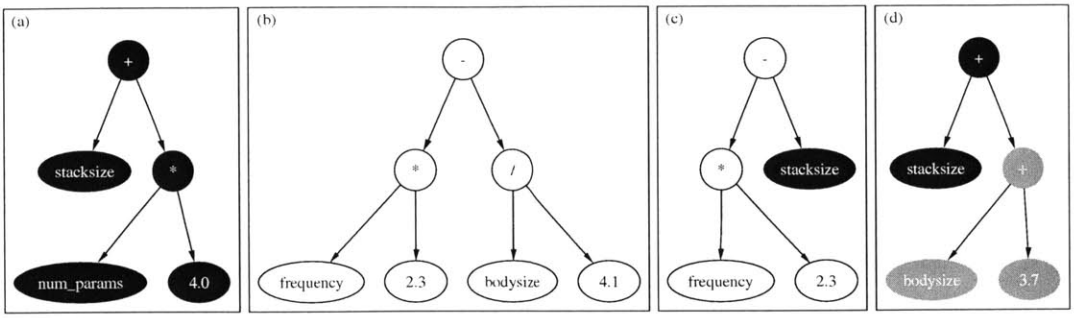

Figure 2-2: Example Finch expressions. Expressions (a) and (b) are examples of expressions produced by Finch. Expression (c) shows an example of a random crossover created from (a) and (b). Expression (d) is the resulting priority function from a mutation of (a).

Genetic programming is also well-suited to parallelization – an important factor in performing large experiments in a reasonable amount of time. At the beginning of every generation, the harness (the central controlling logic) creates all the expressions for the upcoming generation. A combination of lingering expressions from the last generation, new random expressions, crossover combinations, and mutations are chosen. All the expressions are then tested and assigned fitnesses. On a single system, the testing would be run serially and possibly take a very long time to complete. However, since each test has no dependencies on the other tests, the tests can be parallelized by running them on separate machines. With normal configuration parameters, the amount of time required for the centralized work is proportionally tiny, so large savings in runtime are obtained through parallelization. Given that a large cluster of machines was readily available to run experiments, choosing an algorithm which could leverage the resources available was essential.

2.2 Priority Functions

Priority functions are the expressions inside a compiler that Finch targets when optimizing. Also called cost functions, they are heuristic functions used to distill a

multitude of factors into a numerical result, or a cost. They allow for approximate numerical analysis in areas that are difficult to quantify exactly.

Priority functions are used in compilers to quantitatively compare different compilation paths to favor certain directions. For example, deciding which register to spill in a register allocation module, deciding which instruction to schedule next, and the one pertinent to this thesis, deciding which of the candidate functions to inline within the quota restrictions.

2.3 Finch Operation

Finch [11] consists of two related components: a testing harness which drives the genetic programming process by generating the priority functions, launching the compiler trials, and gathering fitness results; and a library which links into the target compiler to enable expression-tree priority function evaluation.

The harness is the component in charge of driving the entire genetic programming process, from generating expressions, to testing expressions, and to collecting and ranking expressions.

At the beginning of every generation, the harness decides what the population for the generation is going to be. The first generation of each run is populated with random expressions and an option to seed the population with initial baseline expressions. Every subsequent generation is created by 'killing' off a percentage of the population, and replacing the voids created with any of the following: randomly created expressions, expressions created by crossovers, or mutated expressions. The population size stays fixed between generations.

Table 2.1 shows the primitives and syntax that Finch uses to build expressions. The top segment represents the real-valued functions, which all return a real value. Likewise, the functions in the bottom segment all return a Boolean value.

Real-Valued Function	Representation
$Real_1 + Real_2$	(add $Real_1$ $Real_2$)
$Real_1 - Real_2$	(sub $Real_1$ $Real_2$)
$Real_1 \cdot Real_2$	(mul $Real_1$ $Real_2$)
$\begin{cases} Real_1/Real_2 & : \text{ if } Real_2 \neq 0 \\ 0 & : \text{ if } Real_2 = 0 \end{cases}$	(div $Real_1$ $Real_2$)
$\sqrt{Real_1}$	(sqrt $Real_1$)
$\begin{cases} Real_1 & : \text{ if} Bool_1 \\ Real_2 & : \text{ if not} Bool_1 \end{cases}$	(tern $Bool_1$ $Real_1$ $Real_2$)
$\begin{cases} Real_1 \cdot Real_2 & : \text{ if} Bool_1 \\ Real_2 & : \text{ if not} Bool_1 \end{cases}$	(cmul $Bool_1$ $Real_1$ $Real_2$)
Returns real constant K	(dconst K)
Returns real value of arg from environment	(darg arg)

Boolean-Valued Function	Representation
$Bool_1$ and $Bool_2$	(and $Bool_1$ $Bool_2$)
$Bool_1$ or $Bool_2$	(or $Bool_1$ $Bool_2$)
not $Bool_1$	(not $Bool_1$)
$Real_1 < Real_2$	(lt $Real_1$ $Real_2$)
$Real_1 > Real_2$	(gt $Real_1$ $Real_2$)
$Real_1 = Real_2$	(eq $Real_1$ $Real_2$)
Returns Boolean constant	(bconst $\{true, false\}$)
Returns Boolean value of arg from environment	(barg arg)

Table 2.1: Genetic programming primitives.

Testing of expressions is done by compiling a set of benchmarks using the expression as a priority function, and measuring the fitness of the expression when it is run. The fitness is a value specific to that run of the benchmark that reveals how well the expression performed relative to your objective. A customary use for the fitness is to measure runtime of a benchmark (lower is better), when the objective is to optimize performance and throughput. The harness ensures that every expression in a generation is run against every benchmark in the set. It batches up all the jobs (expression and benchmark pairs) that must be run, and then farms out the jobs to the parallelizing system.

Parameter	Setting
Population size	400 expressions
Number of generations	25 generations
Generational Mortality Rate	40%
Mutation rate	8%
Tournament size	7
Elitism	Best expression is guaranteed survival.
Fitness	Runtime in either seconds or cycles; lower is better

Table 2.2: Genetic programming parameters.

The Finch harness collects the results from the jobs as they finish, then the expressions are ranked in order of fitness. The rankings are used to determine which expressions make it into the next generation. For all tests in this paper, Finch used the parameters in Table 2.2.

The Finch library is called from the compiler in three places. The first two calls are for initializing the library when the compiler starts, and finalizing the library when the compiler terminates. The third function call is a replacement for the existing priority function. The function call parses the file that contains the candidate expression, then calculates and returns the correct priority. The features important to the priority function being optimized are sent to the Finch library to aid in this calculation. This way various priority functions can be tested in the compiler without changing the source code.

Finch also includes the `fanalyzer` utility, which helps the user gain some numerical and visual insight into how a complex priority function works. When the Finch library generates the values for the priority function, it records the values of every subtree into a file. `fanalyzer` then reads the statistics from the file and calculates the Pearson linear correlation coefficient between the subtree and the entire expression. The data is output in the `dot` [4] format, which can be used to draw a graphical representation of the expression. The head of each subtree is assigned a grayscale color from white to black, with black nodes indicating a correlation of 1 or -1 with

the entire expression, and white nodes indicating a correlation of 0. The very top node of the expression is always colored black, since it represents the entire expression (and has perfect linear correlation with itself). Conversely, constant nodes are always colored white.

Chapter 3

Function Inlining

Deciding whether to expand function calls into the caller function is a difficult task. Like many compiler optimizations, function inlining performance is defined by the fitness of its priority function. The key decision is how much a function should be favored for inlining. Since there are often mitigating parameters such as maximum expansion quotas and minimum priority to inline, creating a priority function by hand can be relatively complex.

A function call arc is a data structure representing a function call from the caller function to the callee function, which contains metadata for both functions. Function inlining is often implemented at the single function call granularity, where each function call from a given function is considered a separate function call arc, even if the calls are to the same target function. Since functions can contain calls to other functions, inlining one function can create more possible function arcs to inline. Because of this, many programs have circular call graphs, and are impossible to inline completely. Stopping the inlining process has to be done by some type of quota heuristic, since eventually the memory footprint may be large enough to cause performance losses by effects such as deteriorating cache hit rates.

Different compilers handle function inlining quotas in different ways. Some com-

```
double square(double input) {
    return input * input;
}

void print-square(double input) {
    double out = square(input);
    cout << square(out);
}

void print-square-inlined(double input) {
    double out = input * input;
    cout << out * out;
}
```

Figure 3-1: Example of function inlining. A C language example. The `square` function is called from `print-square` but inlined into `print-square-inlined`.

pilers implement a quota for codesize growth as a ratio of the original codesize. Other compilers implement a quota for growth of each specific function. And some compilers only allow a certain depth of inlining to occur (for instance, you cannot inline a function call that is part of a previous function call that was inlined).

Existing heuristics for function inlining typically focus on two main factors: *(callee) size* and *frequency.*

Function size can be statically calculated at compile time. Small utility functions are often good targets for inline expansion, since inlining the calls results in little codesize expansion for the number of function calls avoided. A simple example of this would be the `square` function in Figure 3-1. Larger functions may increase the memory footprint of the program to the point where the benefits of inlining are negated.

Callsite frequency is obtainable by a couple different approaches. The most popular way is to use profiling, to determine how many times each function is called during a test run. However, this data is not available at compile time and requires a full test compile and run to obtain the data. Another approach which can be performed at

compile time is static analysis, which can calculate heuristics like how many callsites in the code call the function and how often it is called inside a loop. However, knowing the callsite frequency is extremely valuable, since inlining the functions that are called more often will avoid a larger number of function calls.

Other higher-order effects that are more difficult to qualify include things such as interaction with other phases of compilation (whether the increased freedom for register allocation and instruction scheduling allows for better decisions, or whether it exposes flaws in the compiler that degrade performance or make the completion time of the compile unacceptably long), interactions between the stacksizes of the caller and callee function (whether inlining the function pushes the stacksize past any critical size barriers), the number of parameters to the function, the instruction mix of the function, and so on.

3.1 Compiler Infrastructure

`Pinline` [3] is the profile-driven function inlining module of Trimaran.

The Trimaran inlining algorithm works by sequentially scanning over the source files, and calculating a priority for every function arc encountered. The arc is added to a sorted list keyed by its priority. Once the list contains all the function arcs in the program, then each arc in the list is processed in descending order of priority.

The algorithm processes one arc at a time as follows. First the arc's priority is recomputed since some of the parameters – such as callee function size – may have changed due to the inlining that has occurred since the file scanning stage. If the newly computed priority is less than the old priority by a significant amount, the arc is retagged with the new priority, and reinserted to the sorted list to be processed later. Otherwise, the arc is inlined if there are not mitigating properties that prevent it from being inlined:

Parameter	Setting
Max expansion ratio	1.5
Max function size	1,000,000,000
Max stackframe size	10,000,000
Min expansion weight	1.0
Inline function pointers	yes

Table 3.1: Trimaran Pinline parameters.

1. The code expansion quota has been exceeded (parameter based)
2. The function takes a varying number of arguments like the C `printf` function
3. The function's priority is less than the minimum required for expansion (parameter based)
4. The function's size is greater than the maximum function size that can be inlined (parameter based)

If the inlined function itself contains function calls, then new function arcs are created and added to the list. The current arc is removed from the top of the list, and the algorithm continues until no more arcs are left to be inlined.

The parameters in Table 3.1 were the ones used throughout the experiments. The expansion size is set to 1.5 to give enough freedom for the function inlining to operate. However, it is not so large that everything is inlined and the selection of arcs is inconsequential. The other parameters are set to extreme values so as to try not to impose more restrictions that Finch has to optimize around.

Chapter 4

Implementation

Trimaran was coupled with Finch to enable us to discover priority functions for function inlining automatically.

Trimaran is composed of many separate executables held together by a number of scripts. `Pinline` is the process in Trimaran that applies the function inlining optimization. Initialization and cleanup calls to the Finch library were inserted in the `main` function. The existing priority function for function inlining is the `keyOf` function, which takes in a function call arc that contains all the information needed for a priority function. It was replaced by a call to the Finch library, which takes in the features, parses the priority function from a file on disk (placed there by the harness), and returns the correct value accordingly.

Figure 4-1 shows a simplified excerpt of the Trimaran priority function. For the large majority of cases, the priority function is simply $\frac{frequency}{\sqrt{size}}$, where *size* is the bodysize of the callee, and *frequency* is the frequency the arc is called during profiling. However, it does lower the priority by a factor of 0.8 if the arc is both recursive and indirect. It also has a condition where it will return a huge key if the arc meets certain very favorable parameters. The final return statement is included for when the callee function has not been processed yet. The complexity of this function helps

```
// Note that weight is the profiled frequency
if (RECURSIVE(arc) || INDIRECT(arc))
    weight *= 0.8;
if (weight > 0.0 && !RECURSIVE(arc) && !INDIRECT(arc) &&
    size <= SMALL_FUNCTION && favor_small_functions)
    return weight > HUGE_KEY ? weight : HUGE_KEY;

// size == 0 when callee function's file not yet scanned in
if (size != 0)
    return weight/sqrt((double)size);
// optimistic estimate to be corrected later, size = 1
else
    return weight;
```

Figure 4-1: Excerpt of the original Trimaran baseline function.

to illustrate the difficulty in hand-tuning priority functions.

4.1 Sorted List Issues

Trimaran sequentially scans over the program source files in the scanning phase, and adds all function arcs to a sorted list keyed by their priorities. Once the list contains all the function arcs from the entire program, then each arc in the list is processed in descending order of priority in the processing phase. During the scanning phase, if the current file being scanned contains calls to functions which have not yet been scanned in, the callee portion of the function arc data passed into the priority function will be incomplete. Originally, Trimaran dealt with this with an optimistic approach: it assumed the size of the callee function was the absolute minimum of 1, and proceeded accordingly. Note that since the priority function used by Trimaran was $\frac{frequency}{\sqrt{size}}$, the arc would have its highest possible priority for a given frequency.

Then when the arc is encountered in the processing phase for the first time, the callee function information will be present since all of the functions will have been scanned at that point. The priority will be reduced and reinserted in the correct place

in the list, maintaining the list's invariant. Also note that during the processing phase, the priority of each item in the list only decreases, since frequency stays constant, and size monotonically increases as functions are inlined.

However, this method does not work with Finch, since for some function arcs, there will be a variety of callee features that are unavailable during the scanning phase. Since nothing can be assumed about the priority function, the trick used by Trimaran is not applicable. Instead, the arc is just assigned a priority of ∞ (i.e. using the Linux system *HUGE* double), so that all the infinite priority arcs are processed at the beginning of the processing stage. Their priorities are reduced to the correct values since all the the functions are scanned in at that point.

Unfortunately, since the function sizes will increase as functions are inlined, this could cause the list to have misordered arcs during processing. Size may be positively correlated to priority in a random priority function, so it is possible that some arcs in the list may increase in priority as processing occurs. But the arcs will not rise in the list, since the ordering of the list is only determined by the priority of the arcs at the time of insertion. Modifying the algorithm to recompute priorities for all arcs after each arc is processed would cause the entire process to grow quadratically in runtime relative to the number of arcs, which would be undesirable for scaling the compiler to larger programs. Therefore, as an artifact of the Trimaran implementation, function arcs will only decrease in priority in the list – never increase.

4.2 Platform

Genetic programming was chosen for Finch partly because of its ability to distribute the workload to a machine farm. The Portable Batch System (PBS) was used to parallelize across the workload across multiple identical machines, each equipped with dual 2.2 GHz Intel Xeon processors and 2 GB of RAM running the 2.4.24 Linux

kernel. Each job dispatched consisted of testing a single priority function against a single benchmark. PBS handled the tasks of starting jobs on idle machines, queuing jobs in reserve, and spreading the workload among all eligible machines. Only one job was allowed to run at one time on each machine, even though they were dual-processor machines. The jobs reserve both processors in the machine during runtime to eliminate the effects on runtime that the other processor running would have. About a dozen machines were available to process jobs in the PBS queue, though often the queue was shared with other users running unrelated jobs.

4.3 Features Considered

Extracting features from the function inlining module to create a search space for Finch is an important step. The following features were the ones provided to the Finch priority function:

- Boolean: Recursive – Recursive functions arcs are those that have the same function as caller and callee. `Pinline` can inline recursive arcs, but the inlined copy still makes a function call to itself. Recursive arcs may be less desirable to inline than normal arcs since the function call is not eliminated. This still helps performance since the number of times the arc will be called is significantly reduced. Also, the extra instructions help to create larger basic blocks, exposing more instruction-level parallelism.

- Boolean: Indirect – Indirect arcs are ones that the profiler determines were called using a function pointer. The function call can be inlined by wrapping the inlined code around a check to see if the function pointer is equal to the function inlined. Indirect arcs may be less desirable to inline than normal arcs since the inlined function may not always be the function called from that function pointer.

- Real: Frequency – This is the profiled frequency of the function arc. More frequently called function arcs may be better candidates for inlining, since more function calls will be eliminated.

- Real: Callee bodysize – Functions with a smaller bodysize may be better targets for inlining, since they increase the size of the code by less. Smaller functions allow more functions to be inlined with respect to the quota.

- Real: Callee stacksize – The callee stack may contain parameters passed to the function and local variables used by the function. A larger stacksize may indicate the function does more computational work. This parameter is included since its relationship with function inlining may be useful, but the relationship is not understood well.

- Real: Caller bodysize – A larger caller bodysize may make the arc less attractive for inlining, since the resulting function may have too large of a memory footprint.

- Real: Caller stacksize – The caller stack contains the local variables of the caller function. This parameter is included since its relationship with function inlining may be useful, but the relationship is not understood well.

- Real: Number of Parameters – The more parameters a function has, the more register and stack manipulation goes into the overhead of the function call. Functions with a very large number of parameters may benefit more from being inlined to avoid the extra overhead.

4.4 Workflow

Trimaran consists of many decoupled executables held together by bash shell scripts. This decoupling makes replacing or skipping compile stages very easy by writing

```
_EM_cb_6: /* inflate cb 6 */

    /* op 20 mov [(r 2 i)] [(r 5 i)] */
    _EM_r_2_i = _EM_r_5_i;

    /* op 65 jump [] [(cb 6)] */
    goto _EM_cb_6;
```

Figure 4-2: Example of C-code from gen_probed_lcode. The output is basically a translation of each intermediate representation assembly statement to a C statement.

scripts based on the default Trimaran ones. A number of modified shell scripts and python scripts were used for executing programs, gathering metrics, and monitoring execution.

Although Trimaran provides a cycle-level simulator to execute the compiled code on, its runtime was found to be prohibitively slow with larger benchmark inputs. Finch runs would have taken weeks instead of days given the computing power we had available.

To reduce the length of the compile process, Trimaran's facility that outputs low-level C-language code from its intermediate representation was used. First the benchmark C source code is compiled by Trimaran into its intermediate representation using the `compile_bench` script in basic block formation mode. The resulting intermediate (`.O`) files are normally then processed by the backend. However, the `gen_probed_lcode` script converts the `.O` files back into rudimentary C-code. The resulting C-code output mirrors the low-level instructions of the intermediate representation. An example of the type of code it outputs is shown in Figure 4-2. The `gen_probed_lcode` script then uses the host compiler (e.g. `gcc`) to compile the code into an executable binary.

`gcc` version 3.1.1 was used as the host compiler. It was run in the `-O2` optimization mode, which enables most of optimizations except function inlining. Furthermore, just to ensure that `gcc` was not inlining functions, the `-fno-inline` flag was specified

as well. Using this method, the runtime of the compiler was reduced significantly.

Benchmark inputs were chosen so that the runtime would be large compared to the time difference between runs. This meant about 15-30 seconds of runtime was preferable. To minimize the effects of deviations in time between runs, each benchmark is run 5 times, and the median time is reported as the fitness. Scripts are wrapped around all of the above for logging, monitoring, and convenience.

4.5 Fanalyzer

The existing `fanalyzer` was extended with a Java version so the statistical computation would not be hindered by C++ `double` type precision issues. Java was chosen so the BigDecimal class could be used for easier management of precision. The new program works in exactly the same way `fanalyzer` does.

The only problem encountered in writing the new class was that the Java BigDecimal class has no method for taking a square root in the standard library. However, the BigSquareRoot [5] class found online helped fill that need.

Chapter 5

Experimental Results

5.1 Trimaran Default versus Finch Baseline

Since the default Trimaran priority function is $priority = \frac{frequency}{\sqrt{size}}$ in most cases, this section explores how much performance difference exists between the default priority function and a Finch model of it as only $\frac{frequency}{\sqrt{size}}$. From here on out, the function $priority = \frac{frequency}{\sqrt{size}}$ is referred to as the baseline priority function.

The purpose of this experiment was twofold: (1) to determine whether the modifications made to the Trimaran inlining algorithm affected performance, and (2) to determine whether modeling the default priority function with the simpler baseline priority function in Finch affected performance. Ideally neither one of those modifications should have a statistically large affect on performance, so that the baseline priority function can be used in Finch to represent the default Trimaran function.

Using the Trimaran cycle-level simulator, two versions of Trimaran (the unmodified Trimaran, and the Finch-integrated version with the baseline priority function) were compared across a suite of SPEC CPU2000 Benchmarks [6] and inputs. The results are shown in Table 5.1. In all tests performed, the two are virtually indistinguishable in terms of performance. Hence the Finch approximation for the Trimaran

Benchmark	Input	Trimaran	Baseline	Ratio
164.gzip	lgred.log	724436564	724435649	1.000
164.gzip	lgred.random	1354963908	1355070192	1.000
164.gzip	lgred.source	1694078740	1693487362	1.000
164.gzip	lgred.graphic	1704649971	1704642310	1.000
164.gzip	lgred.program	3183465513	3183475545	1.000
181.mcf	smred	161412531	161456524	0.999
181.mcf	mdred	254597094	254597094	1.000
181.mcf	lgred	1085535946	1085515627	1.000
197.parser	mdred	861090004	873322404	1.014
197.parser	lgred	4660703110	4668080108	1.002
197.parser	test	5406421526	5407536876	1.000
300.twolf	test	452887018	452589756	0.999
300.twolf	lgred	1597538417	1600256998	1.002

Table 5.1: Trimaran default versus Finch baseline results.

priority function makes a good model of the existing priority function.

In all runs of Finch, the baseline priority function is used as the initial seed population which all generated priority functions will be measured against. The baseline priority function will represent the default Trimaran implementation, and speedups will be measured relative to its performance.

5.2 Individual Benchmark Evolution

Finch was used to evolve the Trimaran priority function on each of the four benchmarks (`164.gzip`, `181.mcf`, `197.parser`, and `300.twolf`) individually, using the parameters in Table 2.2. Each of the runs took between 2-4 days. The baseline function, $priority = \frac{frequency}{\sqrt{size}}$, was provided as a seed in each initial population to use as a metric. The resulting best priority functions for each of the benchmark runs will be referred to as best164, best181, best197, and best300, respectively, from here on. A full listing of the priority functions, and their `fanalyzer` outputs can be found in Appendix A.

Benchmark	Starting Arcs	Inlined Arcs	Ending Arcs
164.gzip	120	0-93	120-272
181.mcf	26	0-22	26-60
197.parser	929	0-864	929-3069
300.twolf	458	0-533	458-1256

Table 5.2: Benchmark function arcs statistics.

5.3 Post-Experimental Log Analysis

During each of the four individual benchmark Finch runs, about 3,500 priority functions are created and tested. For each of those priority functions, the compiler output is logged and archived for later examination. Python scripts were used to retrieve data from the logs and format it into statistics that could be analyzed for trends.

Table 5.2 shows how many function arcs each of the benchmarks begins compilation with, and the ranges for the number of arcs inlined, and the number of total arcs when inlining completes.

5.3.1 Fitness Histograms

Figures C-1 through C-4 show the histograms of all priority functions that tested successfully on the benchmarks. Each bar represents how many distinct priority functions fell into the quarter-second range of fitnesses.

It is interesting to note that Figures C-1 through C-3 display a bimodal distribution. These figures correspond to the benchmarks `164.gzip`, `181.mcf`, and `197.parser` respectively.

The larger of the two modes in the figures is the one on the left. It appears there was an abundance of priority functions that performed well. The smaller mode on the right side of the figures indicates there was also a group of priority functions that performed poorly. In all three graphs, there seems to be a clear separation between the group of priority functions that performed well versus the priority functions that

Benchmark	Training Fitness	Percentile
164.gzip	24.63	35.8
181.mcf	36.42	43.6
197.parser	23.3	60.2
300.twolf	14.29	66.9

Table 5.3: Baseline function percentile.

performed poorly, and there seem to be more priority functions that perform well than poorly.

Figure C-4, corresponding to the `300.twolf` benchmark, does not show the same trend as the other three. Its histogram only contains one mode, and it looks very much like a standard normal curve. Table B.4 shows that this benchmark has a very high standard deviation for fitnesses between runs, as a percentage of the fitnesses. An explanation for the normal curve is that the deviation between runs of this benchmark is large enough that it obscures any differences in the priority functions, creating a normal distribution. The large deviation between runs makes analyzing data from this benchmark difficult in subsequent sections, as many of the results obtained are the result of expected statistical outliers.

Table 5.3 shows the performance of the baseline priority functions relative to all the other expressions. The baseline priority functions all fell into the better performing group of the three benchmarks with the bimodal distributions, though their percentile rankings are mediocre. In `300.twolf`, the baseline priority function performed better than the median, with a percentile ranking of 66.9.

5.3.2 Inlined Arcs versus Fitness Scatter Plots

Figures C-5 through C-8 show scatter plots, where each data point depicts a priority function with a specific number of function arcs inlined and its fitness. There is one graph for each of the four benchmarks, and every expression that completed testing with errors is represented in the graphs.

Lower is better for fitnesses in these graphs, since fitness is a measurement of runtime. The best fitness for a given number of arcs inlined in a benchmark will be referred to as the best arc-obtainable fitness.

Figure C-5 and Figure C-7 exhibit a trend where the best arc-obtainable fitness decreases as the number of inlined arcs increases from 0. However, the best arc-obtainable fitnesses reach a minimum, then start to increase with the number of inlined arcs increasing. These figures correspond to the benchmarks `164.gzip` and `197.parser` respectively.

In the case of `164.gzip`, the peak fitness occurs at about 30 function arcs inlined; and for `197.parser`, the peak fitness occurs at about 100 function arcs inlined. After those points, there is a clear dropoff in the best arc-obtainable functions as the number of inlined arcs increases. There seems to be evidence that an "optimal" number of arcs to inline exists, and beyond that point each additional function inlined is detrimental to the fitness of the priority function.

Figure C-6 for the `181.mcf` benchmark does not exhibit this trend as clearly, since the best arc-obtainable fitnesses at 15 and 20 arcs inlined are both local minima. One explanation for this may be that there are just too few function arcs (only 26 at the start of compilation) in this benchmark. There is, however, a slight dropoff in fitness beyond 20 functions inlined, and perhaps the effect would be more pronounced if there were more arcs to inline.

`300.twolf` has a scatter graph that is completely different from the other 3. Near-peak fitnesses occur at many different points for the number of arcs inlined. One observation is that each mode – the distribution for a given number of arcs inlined – appears to be a normal distribution itself; this agrees with the results from the previous section that the overall distribution is normally distributed. There are a number of modes for which the number of arcs inlined contain many data points. Note that the extreme fitnesses (very high and very low) tend to appear in these

modes. These are just the statistical outliers expected from a normal distribution.

5.3.3 Compilation Time versus Fitness Scatter Plots

Although the priority function training used runtime as the fitness, another aspect of fitness could measure the amount of time it takes the compiler to produce an output. This section examines that aspect of the results.

Figures C-9 through C-13 show scatter plots where each data point depicts a priority function and its corresponding fitness and time to compile. Figure C-12 is just a closer view of the left portion of Figure C-11. The best fitness for a given time of compilation in a benchmark will be referred to as the best time-obtainable fitness.

Figure C-9, Figure C-10, and Figure C-12 all show a similar trend where a larger time of compilation results in better time-obtainable fitnesses only up until a certain point; then the best time-obtainable fitnesses get worse as compilation time increases. These figures correspond to the benchmarks `164.gzip`, `181.mcf`, and `197.parser` respectively. One explanation for this is that the larger amount of runtime is being used to inline more functions; and from the previous section's results, there is a dropoff in priority function performance after a threshold is exceeded.

These results seem to indicate that expressions that take much longer to compile do not result in good fitnesses. If this is the case, then in future Finch runs, the abnormally long compilation runs should just be killed after exceeding a time quota by a sniper script. This would speed up the genetic programming process since those expressions take up a disproportionate amount of runtime for their poor fitnesses.

Once again `300.twolf` is an unusual case. Its scatter plot in Figure C-13 does not show the same correlation between fitness and compilation time as clearly. The lowest fitnesses do come from priority functions that have relatively low compilation times, but then there are functions with high compilation times that have fitnesses that are almost as good. Again, the large deviation between runs of the benchmark

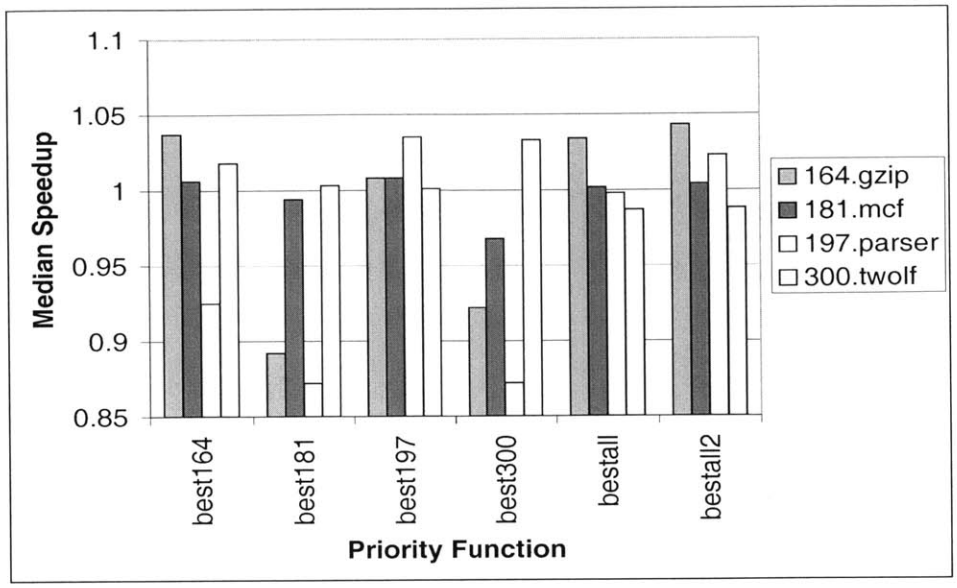

Figure 5-1: Median speedups

probably masks any real performance differences between priority functions.

5.4 Performance Results

The speedups achieved by Finch during training were as follows: `164.gzip` 1.042; `181.mcf` 1.043; `197.parser` 1.043; `300.twolf` 1.12. Speedup is computed $speedup = (bestfitness)/(baselinefitness)$. These were the actual speedups obtained during the running of Finch when choosing priority functions.

After the Finch runs had completed, the baseline function and the newly found best functions for each benchmark were run 25 times each on all four benchmarks to gather statistical data. Tables B.1 through B.4 show detailed information about the performance of the various priority functions on the different benchmarks. Figure 5-1 shows the median speedups obtained by the various best priority functions on all four benchmarks.

Using the data obtained from the statistical runs, the following median speedups were achieved by the best function over the baseline in the benchmarks on the training sets: `164.gzip` 1.037; `181.mcf` .994; `197.parser` 1.035; `300.twolf` 1.033.

Then the best priority functions for each benchmark were run with another set of data, the testing data, for cross-verification of results. The following speedups were achieved by the best function on the testing sets: `164.gzip` 1.037; `181.mcf` .984; `197.parser` 1.042; `300.twolf` 1.014. These were run only once each.

The results show that the improvements from the training set carry through to the testing set, with a similar looking set of figures obtained.

The best181 priority function did not show any speedup on the `181.mcf` training set; this will be examined further in an upcoming section. However, the other three benchmarks all had statistically significant gains.

And to quantify how much function inlining produces speedups in program execution, the following speedups were obtained from running the best priority functions over disabling the inlining module: `164.gzip` 1.153; `181.mcf` 1.084; `197.parser` 1.192; `300.twolf` 1.068.

5.4.1 164.gzip

`164.gzip` was trained with the `train.combined` input set, and tested against the `ref.program input` set, to produce the best164 priority function in Figure A-1.

This benchmark is a modified version of the gzip compression program designed work entirely in memory, so the effects of disk loading are isolated. The inputs vary from logs to programs to binaries to random bits; and the output is a losslessly compressed chunk of data.

The best164 priority function obtained the following median speedups on the four benchmarks: `164.gzip` 1.037; `181.mcf` 1.006; `197.parser` 0.925; `300.twolf` 1.018. The only real speedup obtained by this priority function was in the benchmark it

trained on, so there may be benchmark-specific properties in the priority function.

A simplified version of best164 is: $(callsite\_stacksize)^{3/2} * \sqrt{frequency} * (frequency + 1.5812)/9.5964$.

This *frequency* has a positive influence on the priority of this function, which makes sense as arcs with higher frequencies of execution are better candidates for inlining. The priority is also positively influenced to *callsite_stacksize*, which is interesting since it rewards callsites with larger stacksizes. This may be the benchmark-specific part of the priority function.

5.4.2 181.mcf

`181.mcf` was trained with the `train` input set, and tested against the `ref` input set, to produce the best181 priority function in Figure A-2.

This benchmark solves a problem similar to creating public transit schedules, where a vehicle fleet with a single depot is analyzed to determine a schedule. The inputs are the parameters for the problem, which is a listing of the type of trip and the time length required. The output is an optimal schedule from the benchmark.

The best181 priority function obtained the following median speedups on the four benchmarks: `181.mcf` 0.994; `164.gzip` 0.892; `197.parser` 0.872; `300.twolf` 1.003.

Since the median speedup for best181 on `181.mcf` was actually a slight slowdown, the next two best expressions were tested 25 times to see if they performed better. None of them could produce even a 1% speedup. The priority function created by this run did not create any real improvement in any of the benchmarks, including the one it was trained on. In fact, significant slowdowns in `164.gzip` and `197.parser` indicate this function has little research value.

One reason why this benchmark could have been difficult to improve upon is that there was a magnitude fewer function arcs than in the other 3 benchmarks. Fewer than 25 arcs were inlined in every single trial, in contrast to the 90-900 maximum

inlined arcs in the other benchmarks. With only about 30 decisions to make, there does not appear to be a lot of room to improve over the baseline function. Perhaps the benchmark is just not well suited for inlining research, as many priority functions will produce the same subset of arcs to inline.

5.4.3 197.parser

`197.parser` was trained with a shortened version of the `ref` input set, since the second-longest input still had too short a runtime (about 6 seconds) to train on. The testing was done on the entire `ref` input set. Shortening of the `ref` input set was done via taking a "`head -n 600`" of the normal `ref` input set. 25 generations of Finch produced the best197 priority function in Figure A-3.

This benchmark is a syntactic parser of English. The input is a list of sentences, each of which is analyzed by the parser. The parser outputs its interpretations and annotations of each sentence, such as whether certain words are unneeded. Expectations for this benchmark were that it would have large gains in speedup, since function inlining is especially beneficial to programs like parsers which spend most of their time running in loops and dispatching requests to small functions.

The best197 priority function obtained the following median speedups on the four benchmarks: `197.parser` 1.035; `164.gzip` 1.008; `181.mcf` 1.008; `300.twolf` 1.001. Although the only appreciable gain in performance is on the benchmark it trained on, this was the only priority function of the four that produced speedups of at least 1 on all four benchmarks. This is also easily the best performing function of the four.

A simplified version of the best197 priority function is: $(Recursive\ ?\ .8\ :\ 1) * (frequency^3) * (num\_params) * (\frac{1}{5+callee\_bodysize})$

This priority function is similar to the baseline function in that *frequency* contributes positively to the priority, and that *callee_bodysize* contributes negatively. The power to which both contribute is different though. An additional factor, *num_params*,

also contributes positively to the priority, which makes sense since more parameters requires more function call overhead in moving the parameters into the correct registers and stack positions. Inlining allows the register allocator more freedom, since the extra register activity would create more dependencies. Much like the default Trimaran implementation, there is a multiplicative penalty for the function arc being recursive; possibly since inlining a recursive arc does not fully remove the function call.

5.4.4 300.twolf

`300.twolf` was trained with the `train` input set, and tested against the `ref` input set, to produce the best300 priority function in Figure A-4.

This benchmark is a transistor placement and global routing algorithm that is similar to one which would be used in the production of microchips. According to the benchmark's description, much of the workload from doing inner loop calculations and traversing large enough amounts of memory to force cache misses. The inputs are a series of files which describe the cells to be placed, and the outputs are a routing plan and a placement plan.

As discussed earlier, this benchmark had a large variance in fitnesses between identical runs. The standard deviation was between 3-5% of the median for the baseline and the best300 functions. Since the median speedup obtained was only 3.3%, the variation was extremely significant. Perhaps the fact that most of the work done by this benchmark is done in inner loop calculations, means that the function call performance is comparatively insignificant. The normal distribution of the histogram in Figure C-4 seems to support this theory.

The best300 priority function obtained the following median speedups on the four benchmarks: `300.twolf` 1.033; `164.gzip` 0.922; `181.mcf` 0.968; `197.parser` 0.872. Although the function achieved a speedup in the benchmark it trained on, in all other

benchmarks it recorded significant slowdowns. The large deviation in runtimes could indicate there were better functions in the population, but that this one happened to be an outlier. The fact this function performed so poorly on the other three benchmarks may indicate the presence of many benchmark-specific features in its priority function.

Since Finch only runs each expression once during training, the large deviation could have had an adverse effect on ranking the expressions during training. For instance, during training, both the baseline and best times were smaller than the median times obtained later in the statistical runs. The speedup found during training was a very high 1.12.

A simplified version of the best300 priority function is: $(\sqrt{num\_params} * 1.18) + num\_params$

This priority function is quite unusual in that neither $frequency$ nor $callee\_stacksize$, the two variables in the baseline priority function, appear in it. Instead, it is relies on the positive influence of the num_params variable, which could indicate something benchmark-specific in this expression. Perhaps this benchmark is not well suited to inlining research either.

5.5 All Benchmarks

Using the same setup as with the individual benchmarks, Finch was used to run 25 generations of priority functions on all four of the benchmarks at once. When running multiple benchmarks, Finch determines the fitness for each priority function by the arithmetic mean of the speedups over the baseline. The bestall priority function was produced by this process, and then run 25 times against all four benchmarks for statistical data. It is shown in Figure A-5.

The speedups achieved by Finch during training with the bestall priority function

were: 164.gzip 1.034; 181.mcf 1.021; 197.parser 1.000; 300.twolf 1.136.

From the statistical data, the following median speedups were achieved by the bestall priority function over the baseline on the training sets: 164.gzip 1.034; 181.mcf 1.002; 197.parser 0.998; 300.twolf 0.987. The bestall function does not perform as well as best197, except on the 164.gzip benchmark where it performs about on par with best164.

The large differential between the speedups during training and the median speedups indicates that the bestall priority function won partially because its individual benchmarks ran very favorably compared to their median runtimes. Also, the presence of the 300.twolf benchmark threw off results because of its large standard deviation in fitness between runs, as discussed earlier. Notice the speedup obtained by 300.twolf was a pretty extreme outlier of 13.6% over baseline, and undoubtedly contributed a large amount to the victory of the bestall priority function. Interestingly enough, during the statistical run a slowdown was found on 300.twolf for this function versus the baseline.

5.5.1 Retrospective Function Filtering

The bestall priority function won partly because of its abnormally high 300.twolf fitness. Plenty of evidence has suggested that 300.twolf is not a very useful benchmark in determining function inlining fitness. Perhaps better priority functions were discarded during the genetic programming process, because of the way the inclusion of the 300.twolf benchmark. Using the logs from the all benchmarks run of finch, a Python script was used to sort through all the priority functions and calculate a fitness score using only the other 3 benchmarks. The resulting priority function, bestall2, was then run 25 times for statistical data. It is shown in Figure A-6.

The speedups achieved by Finch during training with the bestall2 priority function were: 164.gzip 1.042; 181.mcf 1.017; 197.parser 1.033; 300.twolf 0.985.

From the statistical data, the following median speedups were achieved by the bestall2 priority function over the baseline on the training sets: `164.gzip` 1.043; `181.mcf` 1.004; `197.parser` 1.023; `300.twolf` 0.988. This is the best overall performing priority function found in these experiments. Specifically, it improved the `164.gzip` and `197.parser` benchmark fitnesses by the most, while staying near even with regards to `181.mcf` and `300.twolf`. It is interesting to note that removing the influence of `300.twolf` on selection did not promote a priority function which greatly sacrificed performance in `300.twolf`.

5.6 Fanalyzer Analysis

Figures A-7 through A-12 show the `fanalyzer` output of the six priority functions obtained from Finch. All leaf nodes with zero correlation have been pruned for brevity.

The head of each subtree is assigned a grayscale color from white to black, with black nodes indicating a correlation of 1 or -1 with the entire expression, and white nodes indicating a correlation of 0. These trees show a graphical representation of which parts of the expression dominate the final priority output, and hence reveal which parts of the expression are more important.

There are two separate important concepts found in the `fanalyzer` outputs. Individual features which correlate strongly (closer to +1 or -1 than 0) with the value of the expression indicate that the function moves with the value of that feature (either positively or negatively depending on the sign of the correlation). Features with low correlations are not very important in the calculating of the expression.

The other concept is that certain branches dominate the computation of the expression when they lie directly along the computation path of the function. Nodes which do not lie along the computational path of the function are ones whose values are used to compute Boolean values for branching (e.g. the first argument of the cmul

and tern primitives). Some features contribute directly to the value of the priority, and some only contribute indirectly via the Booleans.

Figure A-7 shows the best164 priority function. It has a large positive correlation with frequency feature, and a small positive correlation with caller_stacksize. Frequency is definitely the dominant factor in this expression, since the frequency leaves and the subtrees containing frequency are the highest correlated.

The best181 priority function in Figure A-8 shows that the whole expression is very highly correlated (rounded to 1.000 in fact) with the caller_stacksize parameter. There is also a slight negative correlation between the num_params feature and the expression.

Figure A-9, which shows the best197 priority function, has a moderate positive correlation with frequency, and a small positive correlation with num_params. The caller_stacksize and the callee_bodysize features both have small negative correlations. The dominant branch of the expression seems to come from the branch where frequency is divided by the callee_bodysize. This is all multiplied by another expression of the frequency. Another branch with a moderate correlation right subtree off the root, which is the result of the frequency multiplied by num_params.

The best300 function in Figure A-10 has a large positive correlation with num_params, and a moderate positive correlation with callee_stacksize. However, the lt (less than) subtree containing the callee_stacksize node is not used in the actual calculation of the priority. The various num_params branches dominate the expression.

Figure A-11 shows a high positive correlation with frequency, and slight negative correlations with caller_stacksize, callee_stacksize, callee_bodysize, and num_params. The expression is dominated by the right subtree off the root containing a frequency node.

The bestall2 priority function in Figure A-12 has a moderate positive correlation with frequency. It has a slight negative correlation with num_params, callee_bodysize,

and callee_stacksize. The expression is dominated by a chain of multiply operations which lead to many frequency nodes.

5.7 Feature Examination

An examination of the results obtained led us to discern which features mattered the most in function inlining. Two of the functions, best181 and best300, differed greatly in the `fanalyzer` breakdown from the other four. Not surprisingly, they corresponded to the two benchmarks which had results which typically differed from the other results. They were also the two worst performing functions across all the benchmarks (performing far worse than the baseline on many of the benchmarks). For the purposes of looking at trends, those two functions will be ignored.

The four remaining priority functions all have moderate to high positive correlations to frequency. All four of those expressions are dominated by subtrees that contain frequency values. Conversely, the other two priority functions that do not contain the frequency feature are the two worst performing by a large margin. No other feature was so heavily correlated with the best priority functions, so it is a reasonable conclusion that frequency is by far the most important feature of the ones tested in determining which function arcs to inline. Intuitively, it agrees with the idea that higher the frequency the function is called, the more times the function overhead will be saved when it is inlined.

The callee_bodysize parameter appears in three of the functions (best197, bestall, and bestall2) with slightly negative correlations in each. It contributes to the two best performing priority functions in the denominator of a divide expression (best197 and bestall2), but does not contribute to the value of bestall. Again, it is intuitive that the callee_bodysize should be negatively correlated with the priority since smaller functions should be better targets for inlining. The baseline priority function is tuned

to have callee_bodysize in the denominator as well.

The rest of the features do not seem to contain any conclusive trends. The dominance of the frequency and callee_bodysize parameters could indicate why the baseline priority function was tuned only to use those two parameters.

Three priority functions contain the caller_stacksize feature, but it only contributes to the priority directly in the best164 function. There it has a small positive correlation, but it has a small negative correlation in the other two functions it appears in (best197 and bestall2), which were the best performing priority functions. Therefore it is inconclusive what the effect this feature has, if any.

Only the bestall2 function contains the caller_bodysize parameter. It has a small negatively correlation with the value of the function, and does not seem to contribute to the value of the expression.

The callee_stacksize has a negative correlation in the bestall and bestall2 priority functions. It contributes directly to the values of both expressions, and appears to have an impact on the value of the expression in bestall2.

The num_params feature appears in two of the functions, but has a miniscule negative correlation in one of them (bestall2), and a small positive correlation in the other (best197). It does contribute to the overall priority in both expressions, but its effect is inconclusive.

Chapter 6

Related Work

Cheng [3] created the Trimaran Pinline module and its default priority function. Hwu et al. [15] created the predecessor inlining module for the Trimaran compiler, and used only the profiled frequency in determining which arcs to inline.

Way et al. [13] experimented with inlining heuristics on the Trimaran compiler using a demand-driven inliner in a region-based compiler. Two classes of inlining heuristics were examined: the first-order heuristics order functions in order of importance to inline, and the second-order heuristics determine whether a function is inlined.

Stephenson et al. [11] examined using machine learning to target the priority functions in a compiler. They created the Finch framework for unsupervised genetic programming searches for priority functions. They showed they could successfully tune the priority functions that covered region formation and register allocation in the Trimaran compiler. They also improved the priority function used for scheduling instruction prefetches in the Open Research Compiler for the Intel Itanium platform.

An abundance of other research has been conducted involving machine learning use in compilers and only the most relevant works are included.

Kulkarni et al. [12] developed techniques to speed up the searching speed of genetic

algorithms. Cavazos et al. [8] used supervised learning to teach a compiler when to skip instruction scheduling in a Just-In-Time compiler.

Calder et al. [2] examined supervised learning via neural networks and decision trees to search for effective static branch prediction heuristics. Monsifrot et al. [1] used supervised decision tree learning to determine which loops to unroll. Both of these supervised learning approaches involved matching training inputs with known outcomes (they called this process labeling). However, labeling is only possible when the optimal outcomes are known, and not applicable to many problems.

Cooper et al. [10] used genetic algorithms to solve compilation phase ordering problems on an application basis. Grewal, et al. [7] designed the COGEN(t) compiler to use genetic algorithms to map code to irregular DSPs. Both of these approaches involved evolving the application instead of the compiler, though Coopers work used the information learned in the application-specific optimizations to create a general-purpose sequence.

Chapter 7

Conclusion

An examination of the results obtained led us to discern which features mattered the most in function inlining.

Calling frequency was the single most important feature in a priority function: the higher the calling frequency, the higher the priority of the arc should be to inline. The four best performing priority functions (best164, best197, bestall, bestall2), and the baseline priority function, all had high correlations between the priority and the profiled calling frequency. The other two priority functions (best181, best300) did not use the calling frequency, and were the worst two performing priority functions (worse than baseline too) by a large margin. None of the priority functions had a negative or zero correlation between priority and calling frequency.

Callee bodysize was found to have a small negative correlation with priority in the priority functions it appeared in (best197, bestall, bestall2), as well as in the baseline priority function. The three priority functions it appears in are also the best performing three, which reinforces our conclusion that a smaller callee bodysize should make the priority of inlining higher.

The hand-tuned default $priority = \frac{frequency}{\sqrt{size}}$ in Trimaran performed quite well, and agrees with the conclusions about the two features above. The histograms of

the expression fitnesses showed that there is an abundance of good priority functions. One possible reason for this is that the frequency feature is so dominant, that any reasonable combination with other features will produce a good priority function.

Other general trends found while examining logs indicate that there tends to be an optimal number of functions to inline on some benchmarks. Inlining fewer or more function arcs leads to worse performance. Another interesting trend indicates that the expressions that take the longest to compile do not perform well relative to the best expressions.

Genetic programming can be used to improve priority functions via unsupervised machine learning. The bestall2 priority function recorded non-trivial speedups over baseline on the `164.gzip` and `197.parser` benchmarks, which were the two harder benchmarks to record improvements on. It kept about even in the performance of the other two benchmarks.

A couple of benchmarks were not good choices for function inlining research. `300.twolf` had a highly varying runtime that made comparisons of runtimes only differing by a few percent impossible. And `181.mcf` had too few function arcs: on the order of 30-50 versus the hundreds in all the other benchmarks.

7.1 Future Work

Testing more features for the priority function could perhaps reveal important factors that were overlooked. The eight features which were extracted for genetic programming were chosen in part because they were the ones most readily available in the compiler code. Features such as instruction mix or number of memory operations seem like they could be insightful if implemented and tested.

The profiled frequency feature was almost too dominant in the expressions. Its overwhelmingly strong signal may have masked the smaller effects the other features

had. Removing it from the feature list could enable better conclusions to be drawn about the other features. Since it is a profile-obtained feature, it would be interesting to try to perform a static analysis to see if it could perform as well, but without the profiling requirement.

Genetic programming is extremely time-consuming, so the lengths of the various benchmark runs performed were not as long as we would have liked. Continuing the evolution of the functions for more generations would have been interesting, to see if the fitnesses would bottom out. Also, running each expression a greater number of times before taking the median could also alleviate more standard deviation in highly fitness-varying benchmarks like `300.twolf`.

Also, difficulties with the large variances between runs of benchmarks such as `300.twolf` could have had to do with the fact that the Xeon processors in the machines have a very dynamic architecture. Their deep pipelines and complex prediction units make obtaining consistent results difficult. A much longer Finch run could use the Trimaran simulator instead of the `gcc` compiled output approach we took.

It seemed that at an expansion ratio of 1.5, good priority functions were plentiful to come by. One possible explanation for this is that there is so much space to inline functions that it is easy to create a priority function which inlines all of the critical functions. Tightening up that parameter to something smaller could force a greater selection pressure, where the results would be less top-heavy, and reveal greater character about good priority functions. It would also be interesting to note if the same functions performed well as the expansion ratio grew.

Finally, it would be interesting to apply the same methodology to other compilers, since the Trimaran inlining module seems to perform very well as it is already.

Appendix A

Best Priority Functions

```
(cmul
 (not
  (and
   (barg indirect)
   (bconst false)))
 (cmul
  (bconst true)
  (sqrt
   (mul
    (darg frequency)
    (darg caller_stacksize)))
  (darg caller_stacksize))
 (div
  (add
   (darg frequency)
   (dconst 1.5812))
  (cmul
   (bconst false)
   (dconst 5.1384)
   (dconst 9.5964))))
```

Figure A-1: 164.gzip best priority function

```
(tern
 (lt
  (add
   (dconst 6.7389)
   (dconst 6.6538))
  (div
   (div
    (darg caller_bodysize)
    (darg caller_bodysize))
   (darg caller_stacksize)))
 (sqrt
  (darg caller_bodysize))
 (add
  (div
   (dconst 8.4569)
   (darg num_params))
  (add
   (darg caller_stacksize)
   (darg num_params))))
```

Figure A-2: 181.gzip best priority function

```
(mul
 (tern
  (and (bconst true) (bconst false))
  (add
   (tern
    (barg recursive)
    (mul
     (sqrt (dconst 2.3665))
     (add (darg num_params) (dconst 4.8979)))
    (darg frequency))
   (dconst 0.1998))
  (mul
   (tern
    (eq
     (sqrt
      (add (darg num_params) (dconst 4.8979)))
     (div
      (sqrt
       (dconst 9.5950))
      (cmul (bconst true)
    (darg caller_stacksize)
    (dconst 4.1024))))
    (dconst 6.3747)
    (tern
     (barg recursive)
     (mul (darg frequency)
  (dconst 0.8276))
     (darg frequency)))
   (div
    (mul
     (darg frequency)
     (dconst 0.8276))
    (add (dconst 5.0428) (darg callee_bodysize)))))
 (mul darg frequency) (darg num_params)))
```

Figure A-3: 197.parser best priority function

```
(add
 (dconst 7.3144)
 (sub
  (cmul
   (lt
    (cmul
     (barg recursive)
     (dconst 5.6407)
     (darg callee_stacksize))
    (dconst 1.4056))
   (div
    (dconst 5.4112)
    (dconst 9.1427))
   (sqrt
    (sqrt
     (cmul
      (not
       (barg indirect))
      (mul
       (darg num_params)
       (darg num_params))
      (dconst 1.9737)))))
  (sub
   (dconst 0.1324)
   (darg num_params))))
```

Figure A-4: 300.twolf best priority function

```
(cmul
 (not
  (and
   (not
    (lt
     (sqrt
      (div
       (darg frequency)
       (dconst 19.4660)))
     (add
      (dconst 0.8868)
      (darg frequency))))
   (or
    (or
     (bconst false)
     (bconst false))
    (lt
     (darg caller_stacksize)
     (darg callee_bodysize)))))
 (div
  (dconst 8.3499)
  (darg callee_stacksize))
 (div
  (darg frequency)
  (dconst 1.3566)))
```

Figure A-5: bestall priority function

```
(mul
 (div
  (dconst 8.2339)
  (darg callee_bodysize))
 (tern
  (or (bconst false) (barg indirect))
  (dconst 4.4401)
  (mul
   (darg frequency)
   (cmul
    (not
     (barg recursive))
    (sqrt
     (dconst 5.4444))
    (cmul
     (gt
      (tern
       (barg recursive)
       (add (darg frequency) (dconst 0.1998))
       (darg caller_bodysize))
      (tern
       (barg recursive)
       (dconst 3.9016)
       (dconst 4.0268)))
     (div
      (mul
       (mul (darg frequency) (darg frequency))
       (div
(cmul
 (barg indirect)
 (dconst 5.7431)
 (dconst 3.1674))
(cmul
 (bconst false)
 (div
  (darg frequency)
  (div
   (darg num_params)
   (dconst 4.9746)))
 (darg callee_bodysize))))
      (darg callee_stacksize))
     (add (darg frequency) (darg num_params)))))))
```

Figure A-6: bestall2 priority function

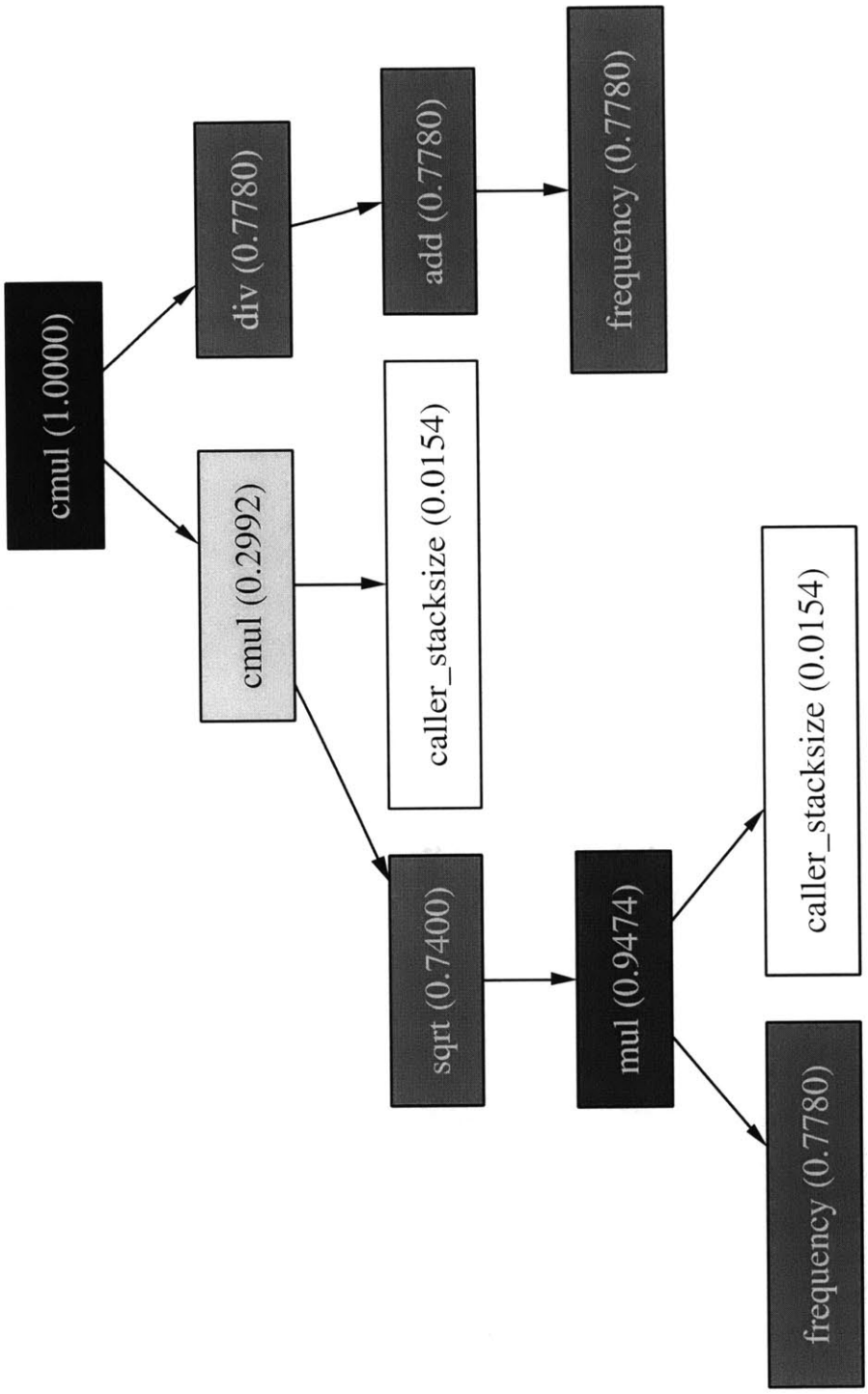

Figure A-7: Fanalyzer best164 Output

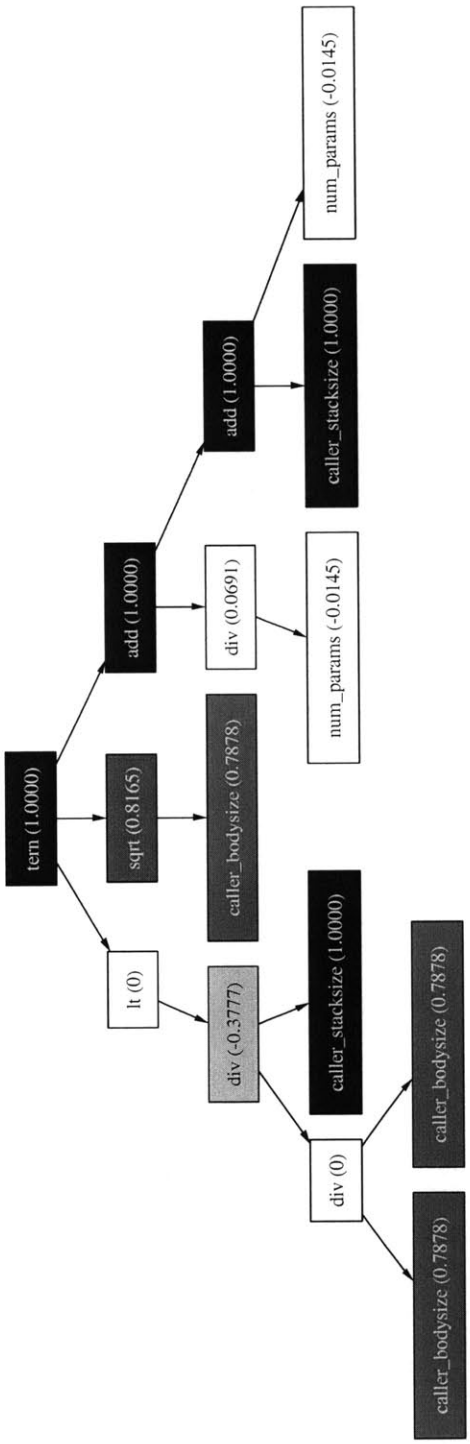

Figure A-8: Fanalyzer best181 Output

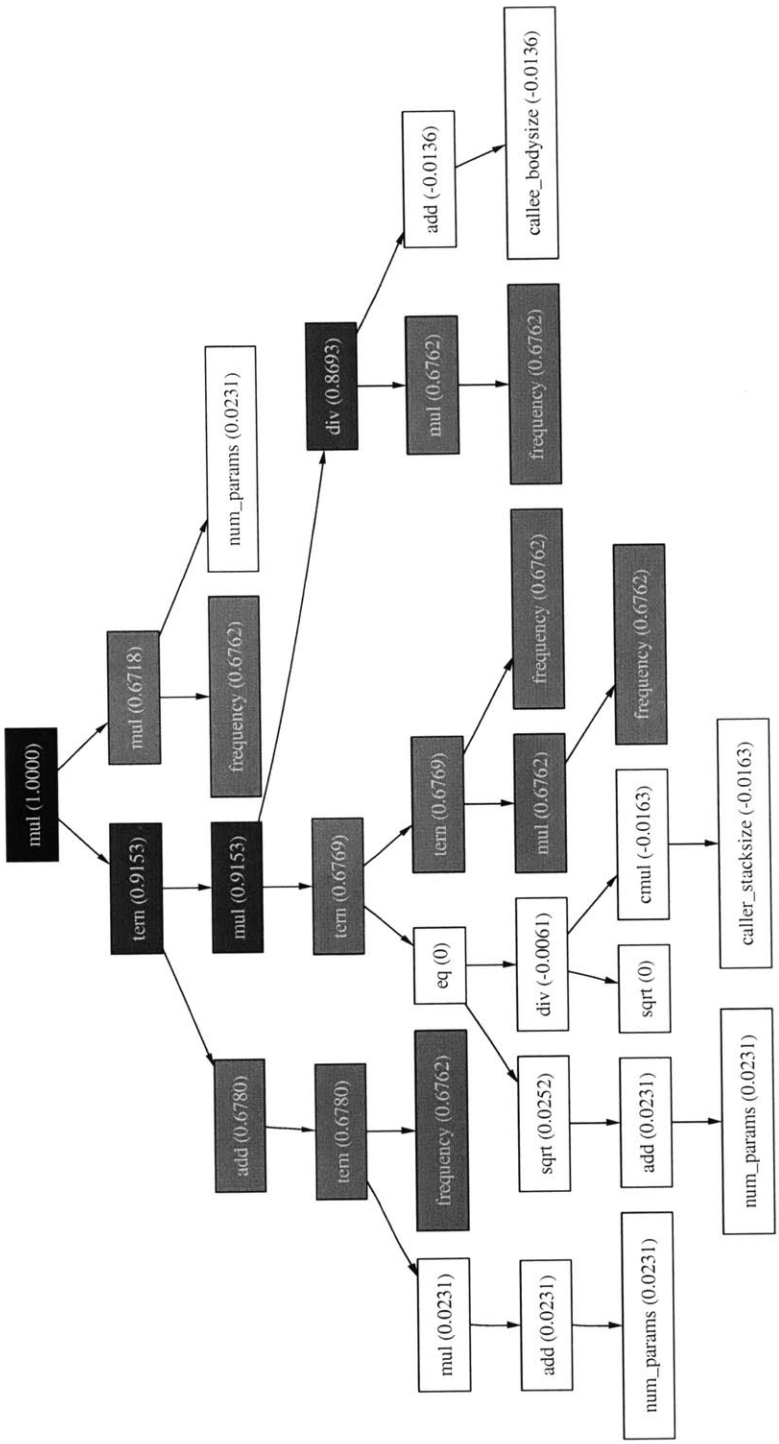

Figure A-9: Fanalyzer best197 Output

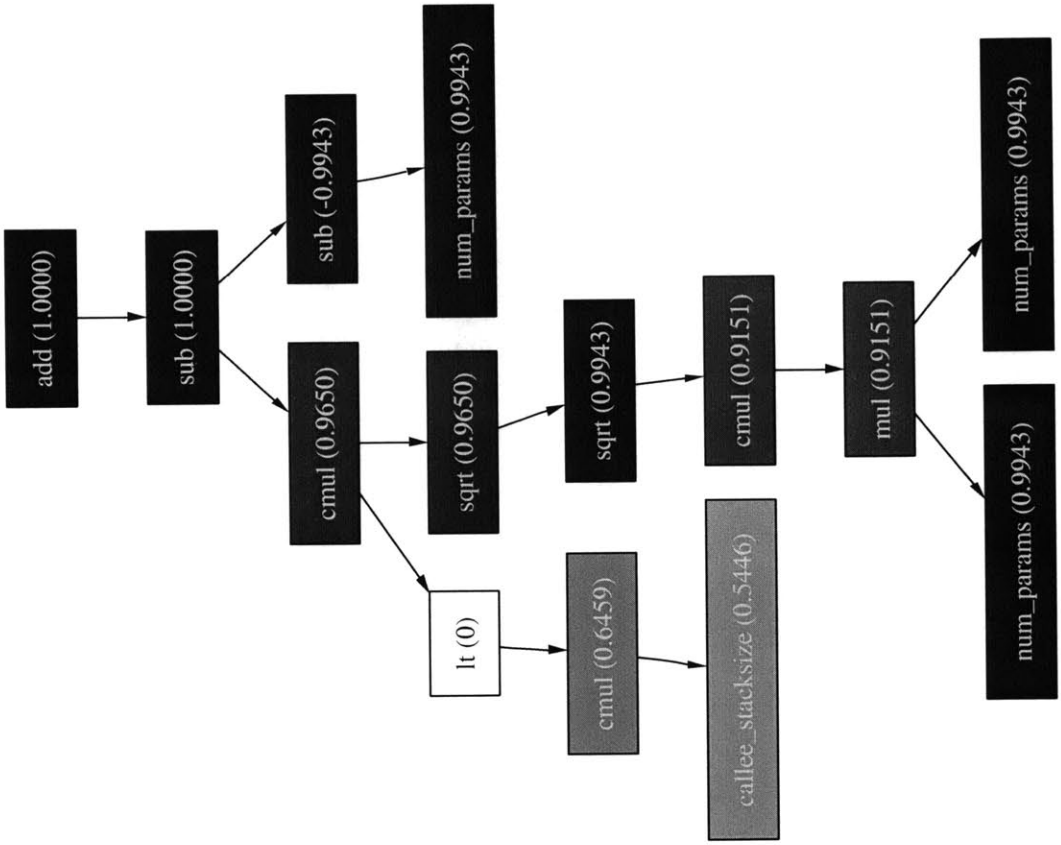

Figure A-10: Fanalyzer best300 Output

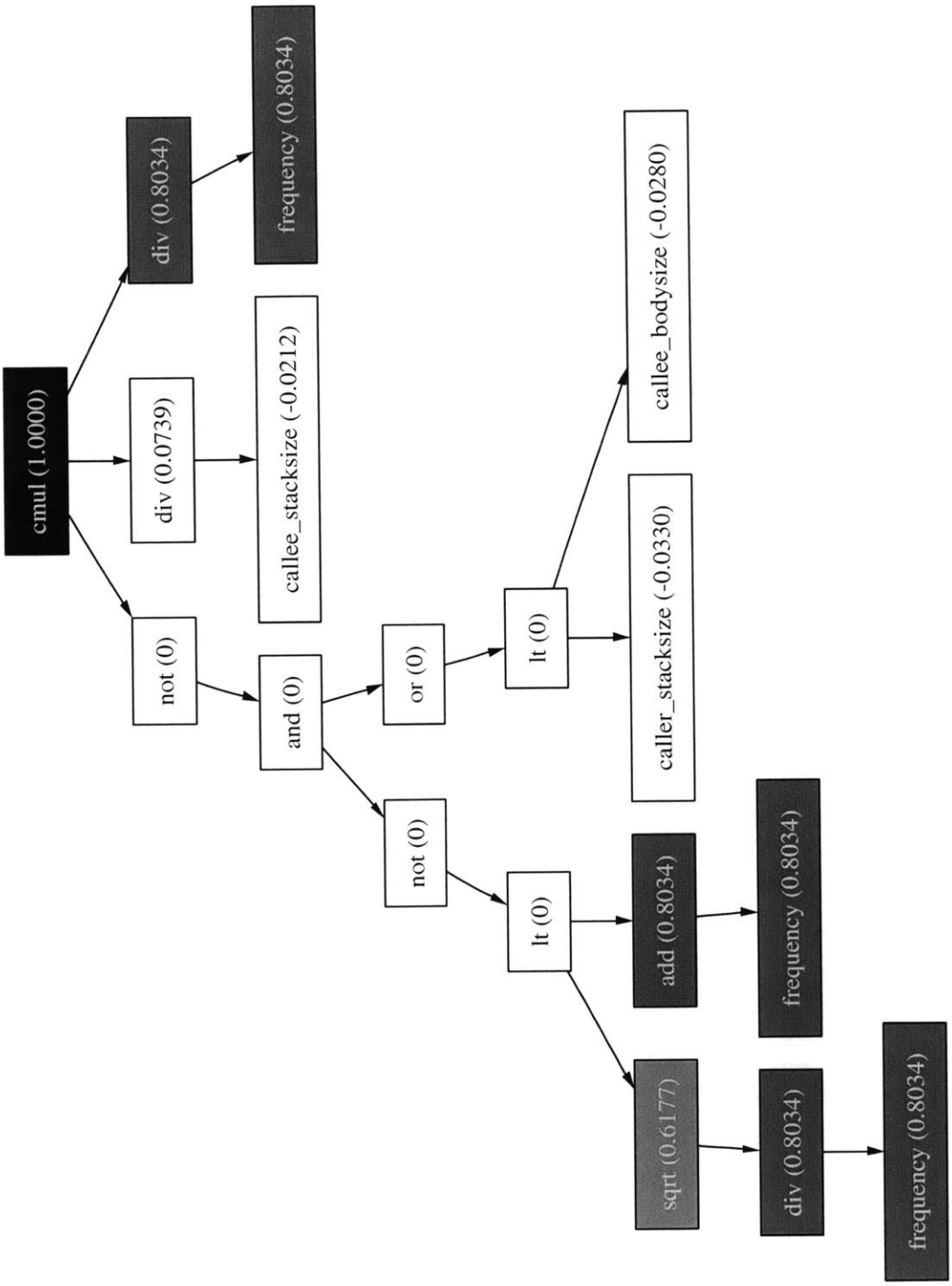

Figure A-11: Fanalyzer bestall Output

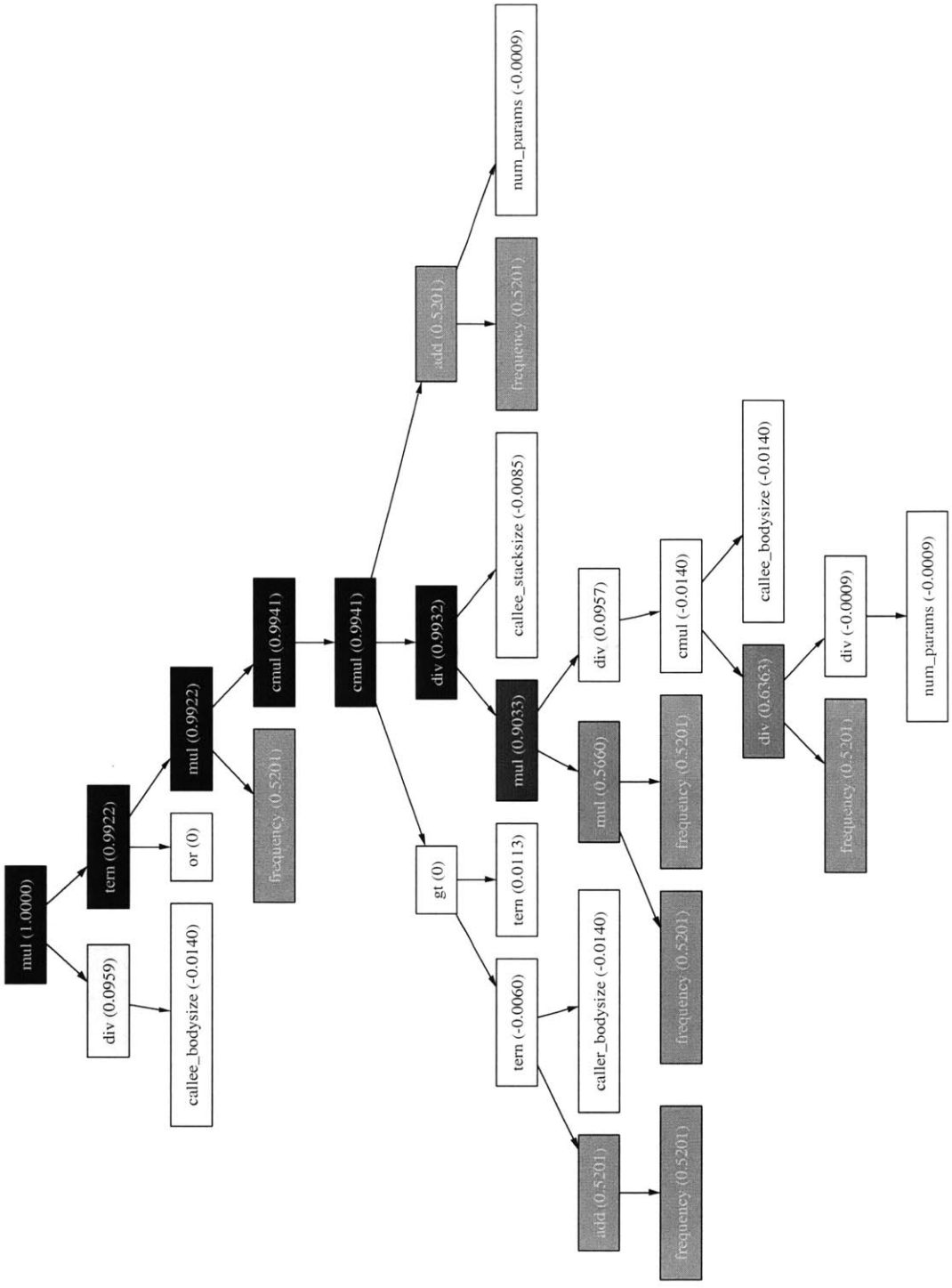

Figure A-12: Fanalyzer bestall2 Output

Appendix B

Tables

	Speedup	Median	Mean	Std. Dev.	Range	% Std. Dev.
baseline		25.07	25.08	0.053	0.21	0.21%
best164	1.037	24.18	24.19	0.038	0.18	0.16%
best181	0.892	28.12	28.15	0.155	0.6	0.55%
best197	1.008	24.88	24.94	0.141	0.54	0.57%
best300	0.922	27.20	27.24	0.136	0.53	0.50%
bestall	1.034	24.25	24.31	0.210	0.81	0.87%
bestall2	1.043	24.04	24.07	0.174	0.71	0.72%

Table B.1: 164.gzip Benchmark Statistics

	Speedup	Median	Mean	Std. Dev.	Range	% Std. Dev.
baseline		36.16	36.34	0.423	1.5	1.17%
best164	1.006	35.95	36.51	1.88	7.09	5.23%
best181	0.994	36.39	36.19	0.470	1.58	1.29%
best197	1.009	35.85	36.42	1.94	7.30	5.41%
best300	0.968	37.34	37.98	1.95	7.29	5.23%
bestall	1.002	36.07	36.39	1.36	7.02	3.76%
bestall2	1.004	36.00	36.57	1.96	7.34	5.43%

Table B.2: 181.mcf Benchmark Statistics

	Speedup	Median	Mean	Std. Dev.	Range	% Std. Dev.
baseline		23.36	23.34	0.111	0.38	4.74%
best164	0.925	25.26	25.37	0.497	1.99	1.97%
best181	0.872	26.78	26.92	0.435	1.52	1.62%
best197	1.035	22.56	22.54	0.108	0.37	0.48%
best300	0.872	26.78	26.95	0.431	1.66	1.61%
bestall	0.998	23.41	23.51	0.423	1.74	1.81%
bestall2	1.024	22.82	22.93	0.458	1.80	2.01%

Table B.3: 197.parser Benchmark Statistics

	Speedup	Median	Mean	Std. Dev.	Range	% Std. Dev.
baseline		14.69	14.70	0.749	2.42	3.89%
best164	1.018	14.25	14.35	0.615	2.57	4.32%
best181	1.003	14.47	15.20	1.803	6.8	12.46%
best197	1.035	14.49	14.44	0.562	1.92	3.88%
best300	1.033	14.04	14.17	0.726	2.89	5.17%
bestall	0.987	14.70	14.76	0.608	2.28	4.13%
bestall2	0.988	14.69	14.70	0.749	2.49	5.10%

Table B.4: 300.twolf Benchmark Statistics

Appendix C

Figures

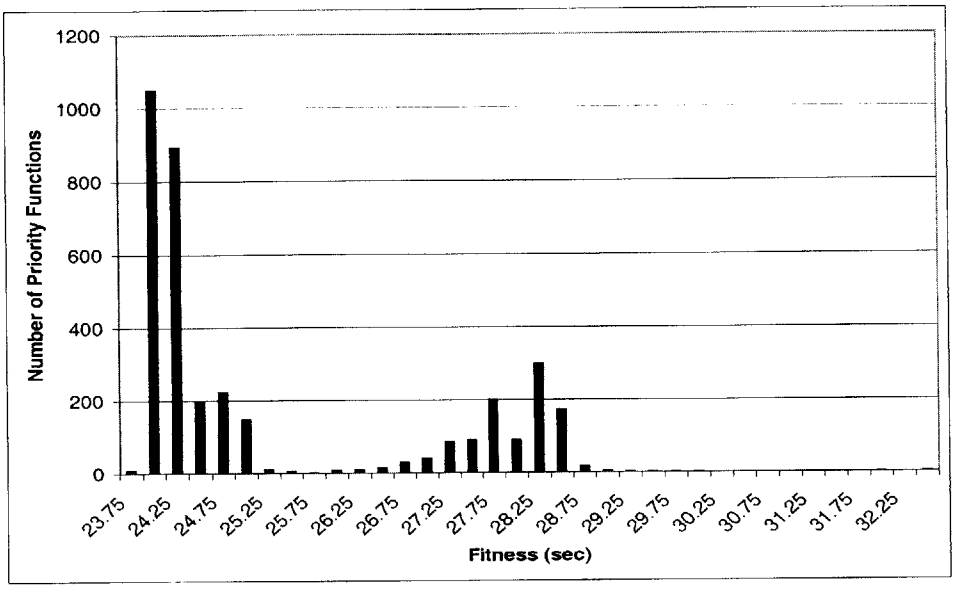

Figure C-1: 164.gzip Histogram of Fitnesses

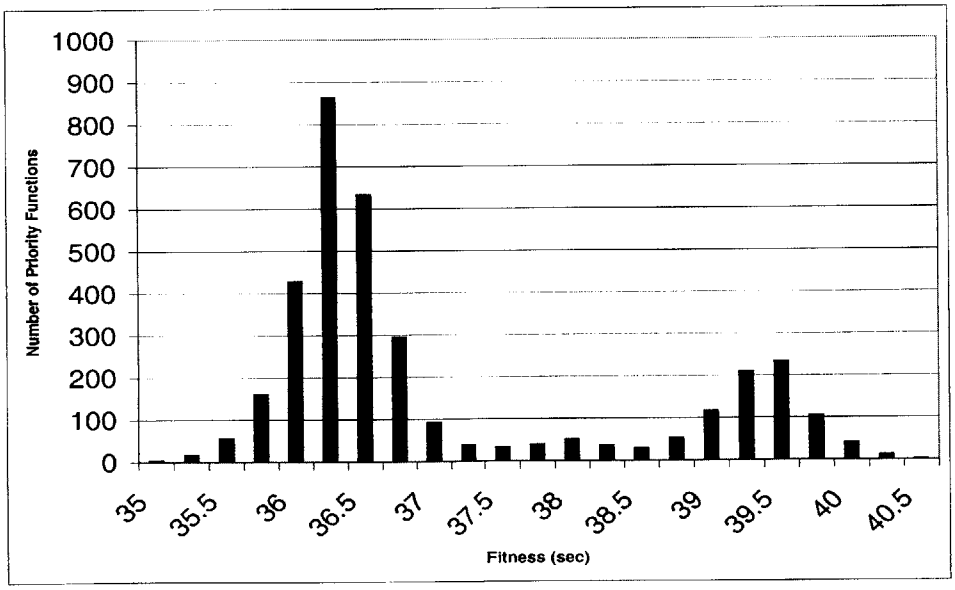

Figure C-2: 181.mcf Histogram of Fitnesses

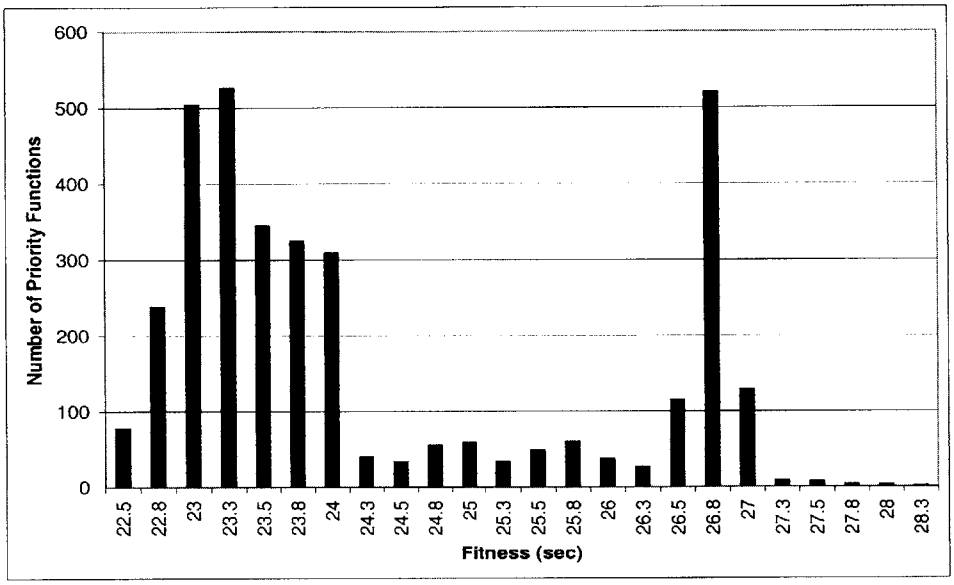

Figure C-3: 197.parser Histogram of Fitnesses

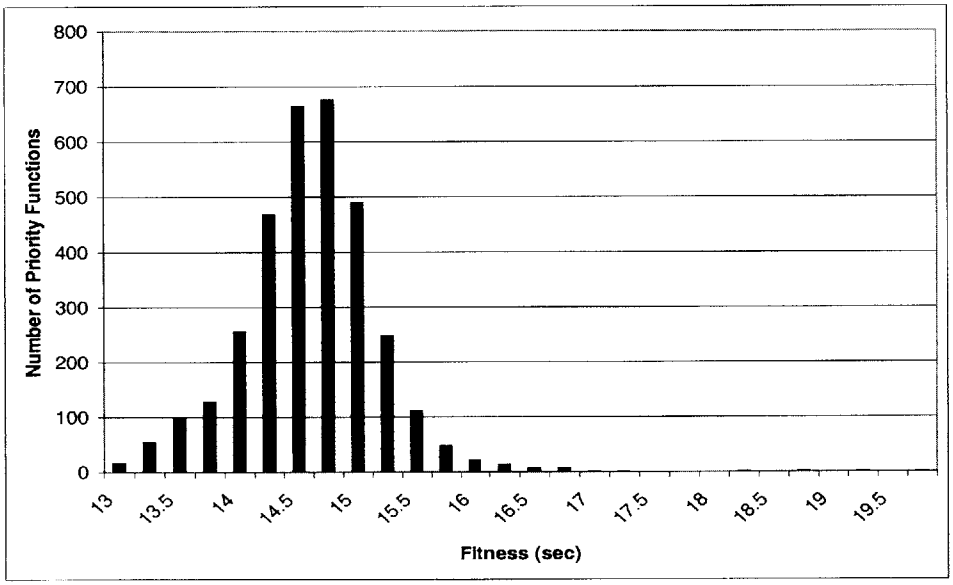

Figure C-4: 300.twolf Histogram of Fitnesses

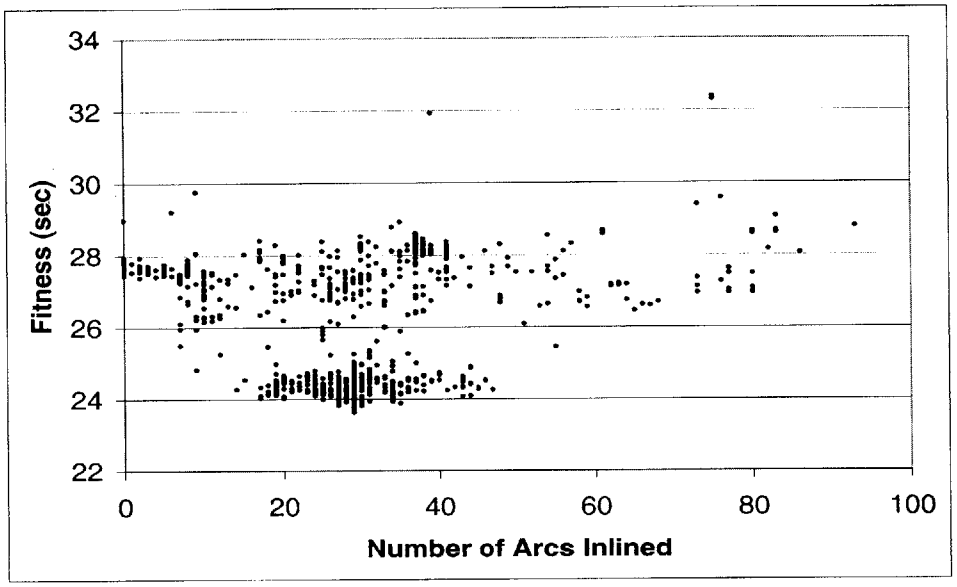

Figure C-5: 164.gzip Scatter Plot of Inlined Arcs versus Fitness

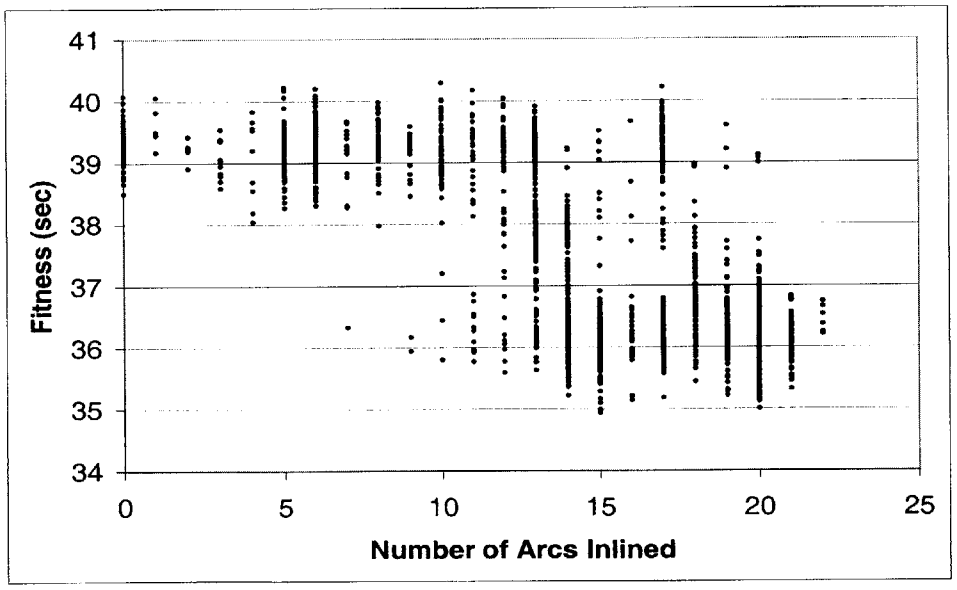

Figure C-6: 181.mcf Scatter Plot of Inlined Arcs versus Fitness

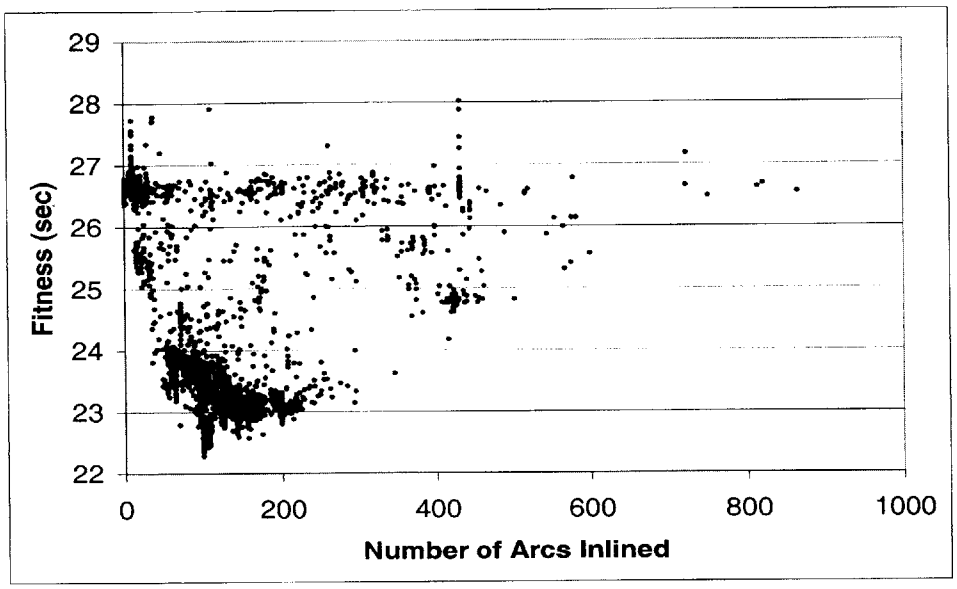

Figure C-7: 197.parser Scatter Plot of Inlined Arcs versus Fitness

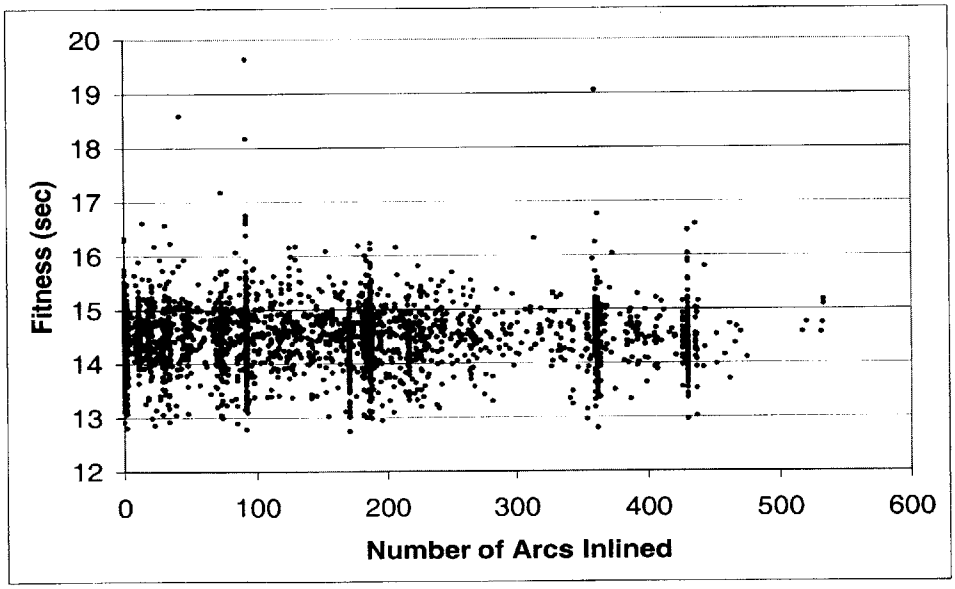

Figure C-8: 300.twolf Scatter Plot of Inlined Arcs versus Fitness

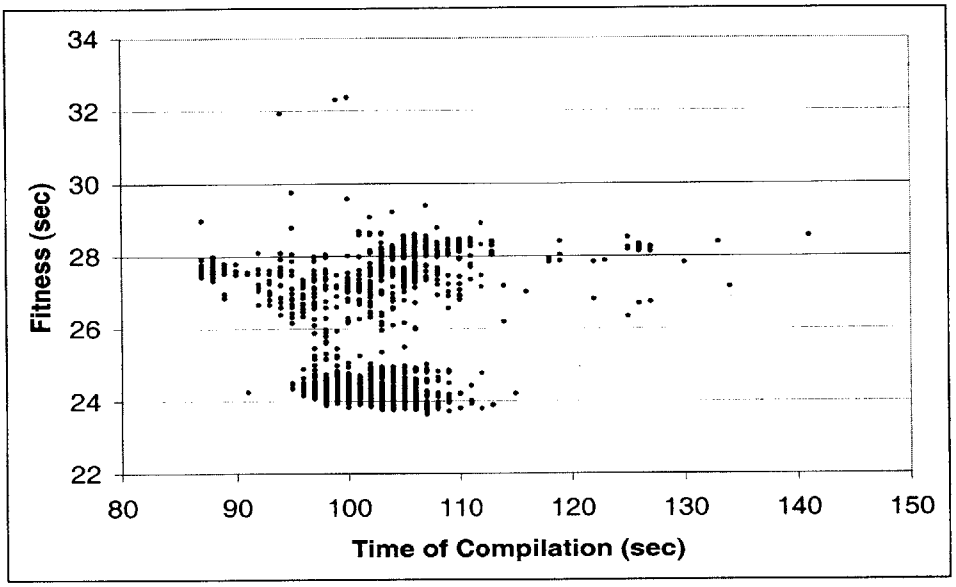

Figure C-9: 164.gzip Scatter Plot of Compilation Time versus Fitness

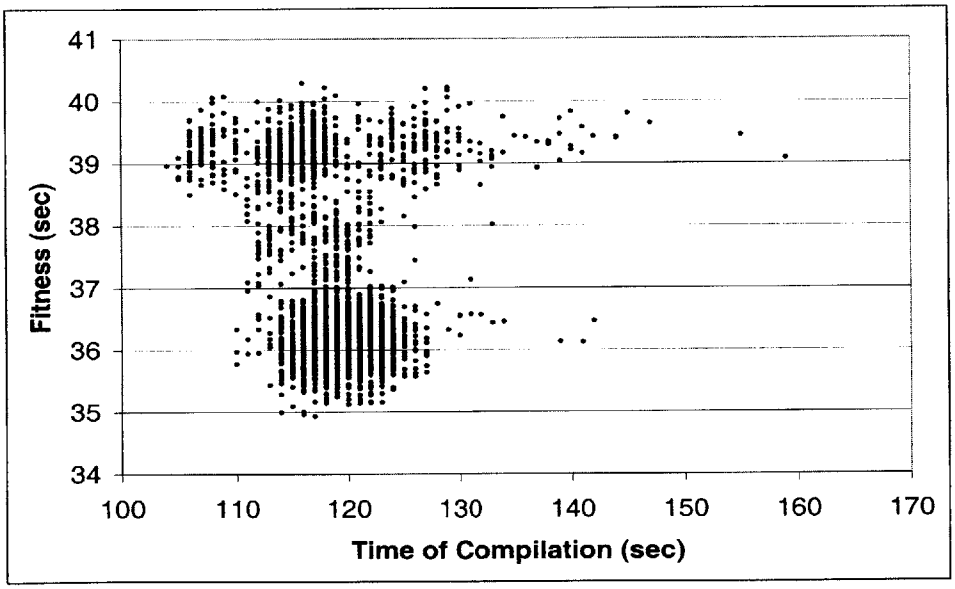

Figure C-10: 181.mcf Scatter Plot of Compilation Time versus Fitness

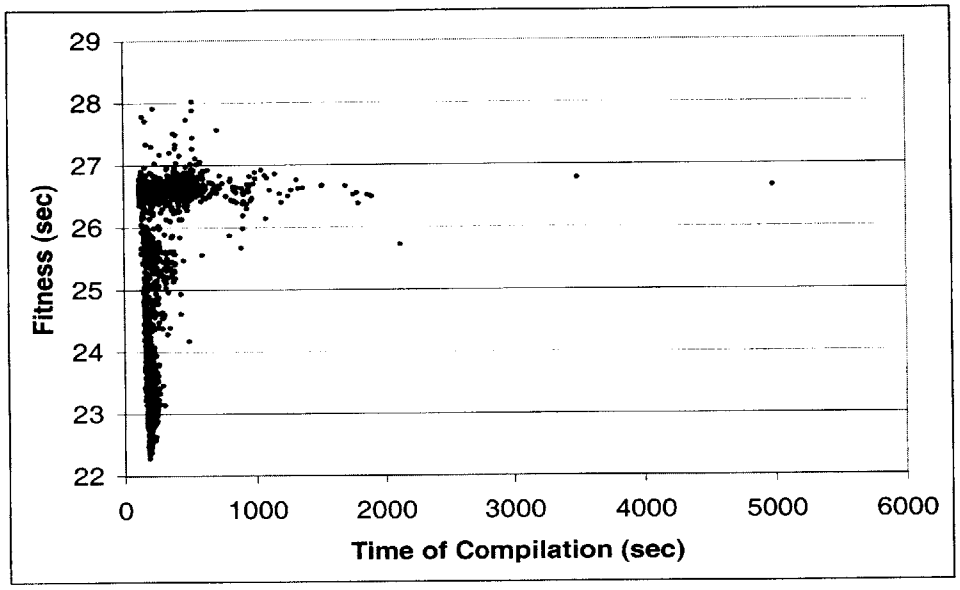

Figure C-11: 197.parser Scatter Plot of Compilation Time versus Fitness

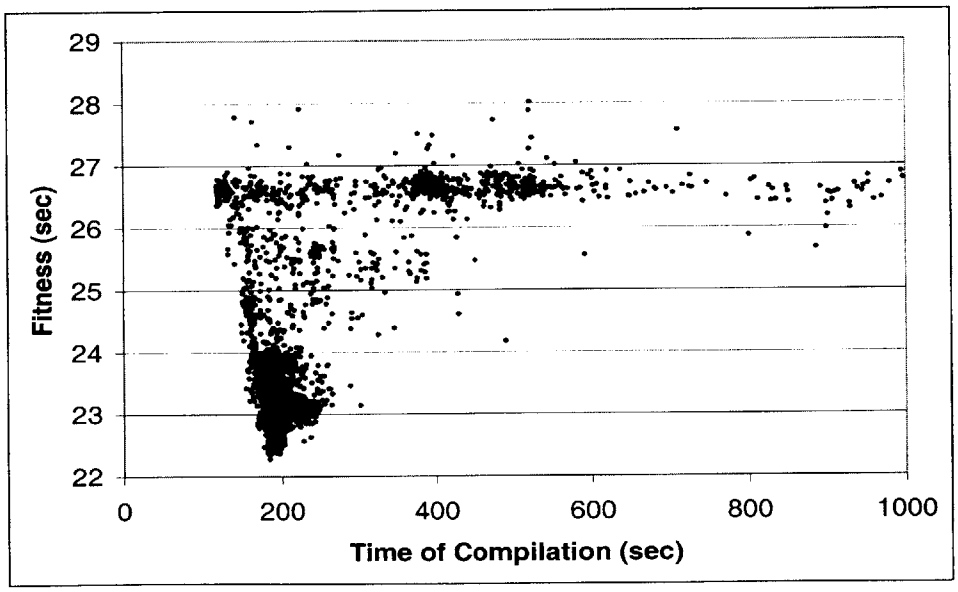

Figure C-12: 197.parser Zoomed View of Scatter Plot of Compilation Time versus Fitness

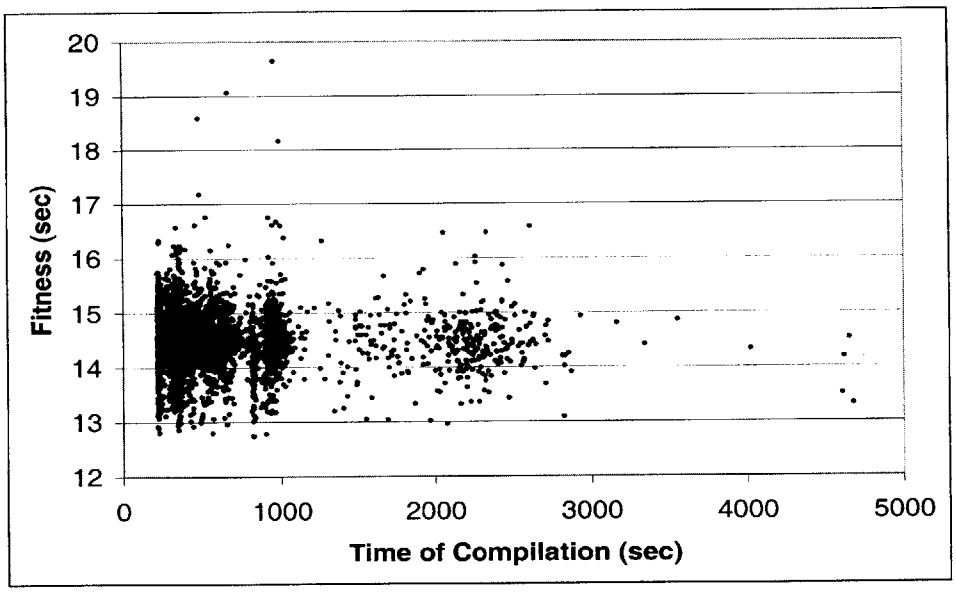

Figure C-13: 300.twolf Scatter Plot of Compilation Time versus Fitness

Bibliography

[1] A. Monsifrot, F. Bodin, and R. Quiniou. A Machine Learning Approach to Automatic Production of Compiler Heuristics. *Artificial Intelligence: Methodology, Systems, Applications*, pages 41–50, 2002.

[2] B. Calder, D. G. ad Michael Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-Based Static Branch Prediction Using Machine Learning. *ACM Transactions on Programming Languages and Systems (ToPLaS-19)*, 19, 1997.

[3] B. Cheng. Pinline: A Profile-Driven Automatic Inliner For the Impact Compiler. Master's thesis, University of Illinois at Urbana-Champaign, 1996.

[4] http://www.research.att.com/sw/tools/graphviz/.

[5] http://www.merriampark.com/bigsqrt.htm.

[6] http://www.spec.org/cpu2000/.

[7] G. W. Grewal and C. T. Wilson. Mapping Reference Code to Irregular DSPs with the Retargetable, Optimizing Compiler COGEN(T). *International Symposium on Microarchitecture*, 42:192–202, 2001.

[8] J. Cavazos and E. Moss. Inducing Heuristics To Decide Whether To Schedule. *Proceedings of the SIGPLAN '04 Conference on Programming Language Design and Implementation*, 2004.

[9] J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* The MIT Press, 1992.

[10] K. Cooper, P. Scheilke, and D. Subramanian. Optimizing for Reduced Code Space using Genetic Algorithms. *Languages, Compilers, Tools for Embedded Systems*, pages 1–9, 1999.

[11] M. Stephenson, S. Amarasinghe, M. Martin, and U. O'Reilly. Meta Optimization: Improving Compiler Heuristics with Machine Learning. *Proceedings of the SIGPLAN '03 Conference on Programming Language Design and Implementation*, June 2003.

[12] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast Searches for Effective Optimization Phase Sequences. *Proceedings of the SIGPLAN '04 Conference on Programming Language Design and Implementation*, 2004.

[13] T. Way, B. Breech, W. Du, and L. Pollock. Demand-driven Inlining Heuristics in a Region-based Optimizing Compiler for ILP Architectures. *IASTED International Conference on Parallel and Distributed Computing and Systems*, 2001.

[14] http://www.trimaran.org/.

[15] W. W. Hwu and P. P. Chang. Inline Function Expansion for Compiling C Programs. *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, 1989.