2008

Profiling, extracting, and analyzing dynamic software metrics

Jeffrey T. Zemerick
West Virginia University

Profiling, Extracting, and Analyzing Dynamic Software Metrics

Jeffrey T. Zemerick

Thesis submitted to the

College of Engineering and Mineral Resources

at West Virginia University

in partial fulfillment of the requirements

for the degree of

Master of Science

in

Computer Science

Katerina Goseva-Popstojanova, Ph.D., Chair

Bojan Cukic, Ph.D.

Powsiri Klinkhachorn, Ph.D.

Department of Computer Science

Morgantown, West Virginia

2008

Abstract

Profiling, Extracting, and Analysis of Dynamic Software Metrics

Jeffrey T. Zemerick

This thesis presents a methodology for the analysis of software executions aimed at profiling software, extracting dynamic software metrics, and then analyzing those metrics with the goal of assisting software quality researchers. The methodology is implemented in a toolkit which consists of an event-based profiler which collects more accurate data than existing profilers, and a program called MetricView that derives and extracts dynamic metrics from the generated profiles. The toolkit was designed to be modular and flexible, allowing analysts and developers to easily extend its functionality to derive new or custom dynamic software metrics. We demonstrate the effectiveness and usefulness of DynaMEAT by applying it to several open-source projects of varying sizes.

Acknowledgements

I would like to thank Dr. Katerina Goseva-Popstajanova for her support, assistance, and for presenting me the opportunity to complete my research under her guidance. I would also like to thank my committee members, Dr. Bojan Cukic and Dr. Powsiri Klinkachorn. I would also like to express gratitude to the other professors in the Lane Department of Computer Science and Electrical Engineering for assisting me with my course work that enabled me to pursue this advanced degree. Special thanks goes out to my fellow researchers Maggie Hamill and Arin Zahalka for their willingness to help me, and to provide comedic relief when necessary. I would like to thank the NASA Office of Safety and Mission Assurance (OSMA) Software Assurance Research Program (SARP) managed through the NASA IV&V Facility, Fairmont, West Virginia, for funding my research. And of course, special thanks my family for their encouragement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction and Motivation

When software executes a lot of different things happen. Functions get called, execute, and terminate often an unknown number of times. In the case of software analysis these unknowns must be known. We need to know exactly which functions get executed, in which order, and how many times. We also must know which functions call which functions, where the most time is spent executing, and where the execution terminates. The type of tool that provides this information is called a profiler, and the data it provides is called a profile. A profile provides details on how the execution of the software is performed. There are many different profilers for every programming language, and all provide very different information. However, when unable to locate a profiler to give a specific set of data or perform certain crucial tasks, it may be necessary to create a new profiler.

The construction of a profiler is not a simple task. Some questions that must be addressed are:

- What type of profile do we need?

- Will a sampling or event-based profiler be best for our situation?
- Is there already an available profiler that meets our needs?
- If not, how should our profiler be implemented?
- How will our profiler gather and store data?

There are no easy, straight-forward answers to these questions. We can begin making progress by determining what type of information we would like to know about a program's execution. But how can information from a profile be useful?

As part of a larger research initiative, we are tasked with analyzing software to determine relationships between software faults and failures and their effect on software reliability assessment. To accomplish this we needed to profile several applications of varying size and complexity. Once we have the profiles, we extract the metrics of the execution, and use the metrics to predict the failure prone parts of the software.

We began profiling using `gprof`, a common call-graph profiler for C. As our research progressed, we became aware that more profiling data, such as a function trace and function timing, would be helpful. Therefore, we decided to construct a profiler, called `jzprof`, to meet our needs.

Once the profiler became operable, we took it one step farther. Our research required extracting metrics from the execution of the studied applications. Previously, these metrics were extracted from the `gprof` profiles and we were limited to function visit counts and caller/callee data. Function timing could not be done due to the sampling nature of `gprof`, as described later in this thesis. But now with `jzprof`, we could extract function times as another metric. `jzprof` continued to be expanded

and improved by allowing for extraction of a function trace and creation of a visual call-graph.

Now, `jzprof` is a component of a larger set of tools and scripts called DynaMEAT, the Dynamic Metric Extraction and Analysis Toolkit. With DynaMEAT we can profile applications, extract metrics, and prepare them for analysis much faster than previously possible.

This thesis is organized as follows. Chapter 2 describes the types of profilers and the profiles that can be produced. Chapter 3 discusses various dynamic metrics and Chapter 4 presents a profiling toolset we created to instrument and analyze C source code to collect these dynamic metrics. Chapter 5 presents the case studies in which DynaMEAT was utilized, and Chapter 6 presents the related work and how it affects our work. The final chapter presents our conclusions and possibilities for future research and development.

Chapter 2

Background

2.1 Profilers

Analyzing software reliability and performance often requires a method of determining what events take place during the execution of software. The type of tool that provides this information is called a profiler. Profiling can be described as the process of analyzing a program's execution to determine statistics of the execution. These statistics often include the functions called, the number of times each function was called, and which functions call which functions. Possessing this information allows developers and analysts to measure performance, optimize the source code, and follow the program's execution.

There are two key types of profiles - sampling-based profilers and event-based profilers. These profilers are also referred to as statistical profilers and exhaustive profilers, respectively. Both types of profilers rely on instrumenting the program's source code to profile.

2.1.1 Sampling-Based Profilers

A sampling-based profiler periodically checks the status of the running program by examining the program's counter. `gprof` [16] is a popular sampling-based profiler. Sampling-based profilers typically introduce less overhead than event-based profilers, but sampling-based profilers are susceptible to inaccuracies. As illustrated in Figure 2.1, if a function executes completely within the profiler's checking interval then the profiler will not know that function was executed. There are steps that can be taken to help eliminate this problem, such as lowering the sampling interval (for `gprof` this requires rebuilding the Linux kernel), providing more input to the program to make it run longer, or profiling the program multiple times to increase the chances that all functions will be detected. However, none of these are complete and reliable solutions to the problem.

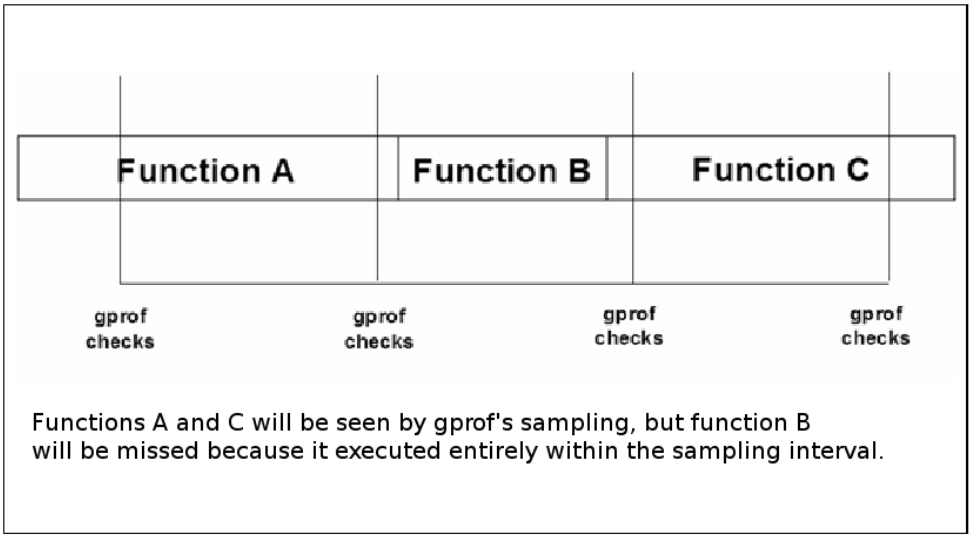

Figure 2.1: Demonstration of Possible Inaccuracies in `gprof`'s Output

2.1.2 Event-Based Profilers

Event-based profilers are triggered by events in the execution of the source code. Whenever a certain event occurs, the profiler will be activated causing it to perform its action, such as logging the event. The trigger event can be practically any event, from an execution of a line of code to a function entrance or exit. Event-based profilers typically introduce more overhead into the program than sampling-based profilers. The increase in overhead is due to the context-switching that occurs each time the trigger is activated. An event-based profiler has the potential to be triggered and execute much more frequently than a sampling-based profiler, which executes in predetermined intervals. However, event-based profilers do not suffer from the potential inaccuracies of sampling-based profilers.

2.1.3 Profiler Implementation

There are multiple ways to instrument code for profiling. Some profilers such as `gprof` instrument the source code at compile time. Other profiling tools instrument binary files [7, 40, 35, 29]. These tools are useful if the source code for the program being analyzed is not available. This thesis focuses on profilers that instrument the source code at compile time.

Profilers that instrument the source code at compile time insert the necessary code at locations called *instrumentation points* [28]. The developers of the profiler can choose where in the source code to place these instrumentation points but the most common locations are immediately before and after each function executes.

Each profiler is designed to meet specific requirements. However, a requirement

shared by all profiler development is to reduce the profiler's overhead as much as possible [12, 42]. Any execution attributed to the profiler is considered overhead, with most of the overhead being attributed to the context-switches resulting from when the profiler is triggered. Too much overhead can skew the profile rendering them useless.

The following is a list of requirements for creating an effective profiler. It is not intended as a comprehensive list, but rather to present the most important requirements for any profiler.

- First and foremost, the profiler must be as light-weight as possible. A profiler that slows down an executing program is worthless because its results will not be representative of the actual program.

- The profiler should be easy to incorporate into the build process of the application. Some applications have lengthy build procedures and introducing a significant change into this process could result in problems.

- The profiler should also be able to profile optimized code as well as unoptimized code [12]. This allows for the profiling of both test and production code.

2.2 Types of Profiles

Profiles can be obtained in different levels of granularity. For function-level profiles, there are two key types of profiles - a flat profile and a call-graph profile. The different types of profiles presented here are from executions of `indent` [38] on the same test case.

2.2.1 Flat Profile

A flat profile shows how much time was spent in each function and how many times that function was called. A flat profile will quickly show which functions in the code are visited more often and which functions are using the most time, which is especially useful for code optimization. Figure 2.2 is an example of a flat profile created by `gprof` when executing the program `indent`.

```
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00       78     0.00     0.00  compute_code_target
  0.00      0.00     0.00       62     0.00     0.00  lexi
  0.00      0.00     0.00       55     0.00     0.00  count_columns
  0.00      0.00     0.00       32     0.00     0.00  xmalloc
  0.00      0.00     0.00       23     0.00     0.00  output_line_length
  0.00      0.00     0.00       22     0.00     0.00  is_reserved
  0.00      0.00     0.00       19     0.00     0.00  better_break
  0.00      0.00     0.00       19     0.00     0.00  fill_buffer
  0.00      0.00     0.00       19     0.00     0.00  set_buf_break
  0.00      0.00     0.00       19     0.00     0.00  set_priority
  0.00      0.00     0.00       16     0.00     0.00  parse
  0.00      0.00     0.00       16     0.00     0.00  reduce
  0.00      0.00     0.00       15     0.00     0.00  clear_buf_break_list
```

Figure 2.2: Flat profile of `indent` execution produced by `gprof`.

From this flat profile we can see that 78 calls were made to the *compute_code_target* function. If our task is to optimize the performance of `indent`, *compute_code_target* should be the first function we optimize because it is called most often. It is easy to see from the zeroes in Figure 2.2 that `gprof` often may not provide accurate timing due to its sampling-based nature.

2.2.2 Call-Graph Profile

A call-graph profile shows which functions (callers) called which functions (callees). A call-graph profile allows for tracing the execution of a program. Figure 2.3 shows the beginning of the call-graph produced by `gprof` while executing `indent`.

```
                     Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) no time propagated

index % time    self  children    called     name
                0.00    0.00      13/78          dump_line [14]
                0.00    0.00      19/78          set_buf_break [9]
                0.00    0.00      46/78          output_line_length [5]
[1]      0.0    0.00    0.00      78         compute_code_target [1]
-----------------------------------------------
                0.00    0.00      62/62          indent [22]
[2]      0.0    0.00    0.00      62         lexi [2]
                0.00    0.00      22/22          is_reserved [6]
                0.00    0.00      18/19          fill_buffer [8]
-----------------------------------------------
                0.00    0.00      13/55          dump_line [14]
                0.00    0.00      19/55          set_buf_break [9]
                0.00    0.00      23/55          output_line_length [5]
[3]      0.0    0.00    0.00      55         count_columns [3]
```

Figure 2.3: Call-graph of `indent` execution produced by `gprof`.

From the call-graph in Figure 2.3, we can see that the function *compute_code_target* was called a total of 78 times. Of those 78 times it was called 13 times by *dump_line*, 19 times by *set_buf_break*, and 46 times by *output_line_length*. Because of the absence of any functions listed under it, we can tell that *compute_code_target* did not call any functions. The function *lexi* called *is_reserved* 22 times and *lexi* called *fill_buffer* 18 times.

2.2.3 Function Trace

A function trace is a listing, often formatted hierarchically, that shows the order and depth of function calls.

```
\--ENTERING reset_parser at 1297109194
\--EXITING  reset_parser at 1297111786
\--ENTERING indent at 1297121134
\--\--ENTERING clear_buf_break_list at 1297122666
\--\--EXITING  clear_buf_break_list at 1297123086
\--\--ENTERING fill_buffer at 1297124706
\--\--EXITING  fill_buffer at 1297125906
\--\--ENTERING lexi at 1297127074
\--\--\--ENTERING fill_buffer at 1297129046
\--\--\--EXITING  fill_buffer at 1297132390
\--\--EXITING  lexi at 1297133710
\--\--ENTERING dump_line at 1297148786
\--\--EXITING  dump_line at 1297151886
\--\--ENTERING lexi at 1297153630
\--\--EXITING  lexi at 1297177126
\--\--ENTERING print_comment at 1297179730
\--\--\--ENTERING inc_pstack at 1297180138
\--\--\--EXITING  inc_pstack at 1297180550
\--\--\--ENTERING current_column at 1297182070
\--\--\--EXITING  current_column at 1297183358
\--\--\--ENTERING dump_line at 1297185034
\--\--\--\--ENTERING pad_output at 1297226782
```

Figure 2.4: Function trace of `indent` execution produced by `jzprof`.

Some function traces also indicate function returns, either implicity through a decrease of depth, or explicitly in the trace. Function traces are useful to follow the execution of a program. They allow for easy identification of cycles and recursion.

Figure 2.4 shows a function trace of an execution of `indent` that implicity and explicitly defines function returns. The number after a function entrance or exit is

a timestamp produced by accessing the Pentium Timestamp Counter. By explicitly stating returns and associating them with a timestamp we can determine how much time was spent executing a particular function.

Execution traces are also useful to compare executions. In [31], several uses of trace comparison are identified, such as determining the effectiveness of system testing compared to the use of the system in the field. Comparing traces is also useful to minimize the size of test cases [20]. By comparing traces generated by test cases we can eliminate duplicate tests and find areas of the software that are not being addressed by the test cases.

Work has also been done to reverse-engineer UML sequence diagrams from execution traces. In [18] and [33], an execution trace is used to create a sequence diagram. It is important to note that an execution trace represents only one possible execution of the software. A complete sequence diagram would additionally require static analysis.

The function trace is the most detailed of the profiles. The flat profile and call-graph can be derived from a function trace. The drawback of a function trace is its potential size. An execution in which many function calls are made can produce a function trace too large to easily analyze and study.

Chapter 3

Dynamic Metrics

3.1 Overview of Dynamic Metrics

Two types of metrics used for software analysis are static metrics and dynamic metrics. Static metrics are determined and calculated by analyzing the software's source code. Examples of static metrics include the number of lines of source code (SLOC) and McCabe's measure of the complexity of software, called cyclomatic complexity [30]. Unlike static metrics, dynamic metrics are collected during the execution of the software. Static metrics are often used to estimate development effort, testing, and management of the software [1, 27].

On a high level, a dynamic metric can be a measure of the time the software executes before completion or a measure of the resources used by the software while executing. On lower levels, dynamic metrics could include which functions were called and when. An even lower level of measurement could include which lines of code were executed.

For a dynamic metric to be effective, it has been proposed that the metric should be unambiguous, dynamic, robust and discriminating, and independent of the machine collecting the metrics [9].

The following dynamic metrics are the target of the tools described in this thesis. These metrics are function-level metrics and were proposed in [15].

To help illustrate these metrics, a part of the visual call-graph for an execution of `indent` is given in Figure 3.1. In the figure, the numbers on the arcs indicate the number of times that function was called by its parent.

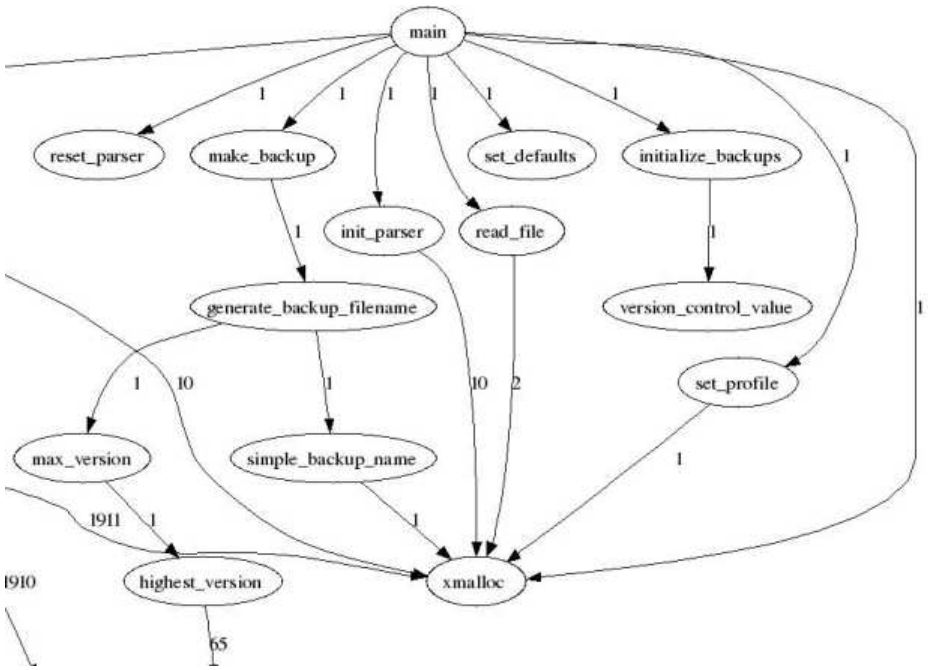

Figure 3.1: Part of an `indent` call-graph illustrating visit counts, fan-in, and fan-out.

- Visit count, VC_j, is a scalar value representing how many times function j was visited during one execution. Visit counts are often displayed in a matrix (Figure 3.2), where the rows of the matrix are the executions (testcases) and

the columns of the matrix are the functions in the program. The aggregate visit count for a function is the sum of the values in that function's column of the matrix, $VC_j^* = \sum_{n=1}^{k} VC_{n,j}$ where k is the number of executions (rows of the matrix).

testcase	addkey	better_break	clear_buf_break_list	compute_code_target
already-starred.c.txt	0	12	7	42
args.c.txt	0	1920	919	7325
backup.c.txt	0	391	189	1339
backup.h.txt	0	17	19	60
bad-break.c.txt	0	12	8	36
bad-comment.c.txt	0	5	7	17
bbb-test.c.txt	0	10	9	35
bbreak.c.txt	0	19	8	36
box-comm.c.txt	0	4	7	19
boxed.c-0.txt	0	17	25	87
boxed.c-1.txt	0	17	25	87
boxed.c-2.txt	0	17	25	87

Figure 3.2: Portion of matrix of `indent` visit counts.

If we are only interested in which functions were visited, we can replace all non-zero values in the matrix with 1, as in Figure 3.3.

testcase	addkey	better_break	clear_buf_break_list	compute_code_target
already-starred.c.txt	0	1	1	1
args.c.txt	0	1	1	1
backup.c.txt	0	1	1	1
backup.h.txt	0	1	1	1
bad-break.c.txt	0	1	1	1
bad-comment.c.txt	0	1	1	1
bbb-test.c.txt	0	1	1	1
bbreak.c.txt	0	1	1	1
box-comm.c.txt	0	1	1	1
boxed.c-0.txt	0	1	1	1
boxed.c-1.txt	0	1	1	1
boxed.c-2.txt	0	1	1	1

Figure 3.3: Binary visit count matrix of `indent` visit counts.

- Fan-in, FI_j, is a vector where the values are the names of the functions that call function j. A fan-in due to recursion is counted. For example, from Figure

3.1, $FI_{next\_state} = \{get\_token, next\_state\}$.

- Fan-out, FO_j, is a vector where the values are the names of the functions that function j calls. A fan-out due to recursion is counted. For example, from Figure 3.1, $FO_{generate\_backup\_filename} = \{max\_version, simple\_backup\_name\}$.

- $FI_j^{Distinct}$ is the number of distinct functions from which a function receives control. This number is equal to the number of functions in the fan-in vector FI_j, illustrated in Figure 3.4.

testcase	addkey	better_break	clear_buf_break_list	compute_code_target
already-starred.c.txt	0	1	3	4
args.c.txt	0	2	4	4
backup.c.txt	0	2	3	4
backup.h.txt	0	1	3	4
bad-break.c.txt	0	2	4	3
bad-comment.c.txt	0	1	3	4
bbb-test.c.txt	0	1	3	3
bbreak.c.txt	0	2	3	3
box-comm.c.txt	0	1	3	3
boxed.c-0.txt	0	1	3	3
boxed.c-1.txt	0	1	3	3
boxed.c-2.txt	0	1	3	3

Figure 3.4: Portion of matrix of `indent` fan-ins showing $FI_j^{Distinct}$.

- $FO_j^{Distinct}$ is the number of distinct functions to which control is passed. This number is equal to the number of functions in the fan-out vector FO_j.

Chapter 4

DynaMEAT: Dynamic Metric Extraction and Analysis Toolkit

4.1 Overview and Objective of DynaMEAT

The objective of DynaMEAT is to provide a methodology for the extraction of dynamic metrics. To accomplish this, we must first profile the target code. The resulting profiles are then parsed and inserted into a database to allow for the extraction and derivation of dynamic metrics. The collected metrics can then be analyzed. A high-level diagram of DynaMEAT is given in Figure 4.1, and the components of each block are described below.

- Profiling of the Code
  - A C profiler called `jzprof` which was created to resolve `gprof`'s lack of accurate timing data, ensure that no functions are missed during an execution due to `gprof`'s sampling interval, and to provide a means for the

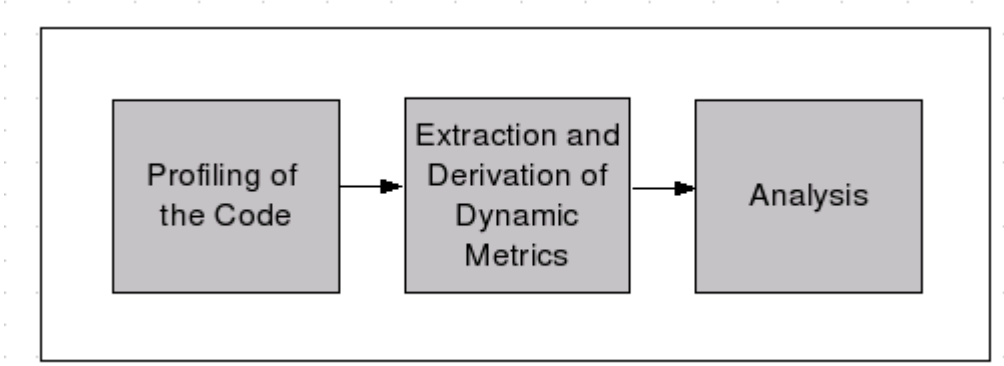

Figure 4.1: High-level Block Diagram of DynaMEAT

extraction of dynamic metrics.

  - A graphical user interface for viewing profiles called `jzprofgui`.

- Extraction and Derivation of Dynamic Metrics
  - AWK scripts to parse and insert the profiles into a database.
  - A tool called `MetricView` for extracting and deriving metrics from the profiles.

- Analysis
  - `MetricView` allows for analyzing skewness of executions.
  - Further analysis can be performed on the metrics extracted by `MetricView`.

As shown in the sequence diagram in Figure 4.2, `jzprof` is used first to profile the application being tested. The resulting profile can optionally be analyzed using `jzprofgui`, or the profile can be parsed and inserted into a database. Once the profile is in the database, `MetricView` can extract the required metrics.

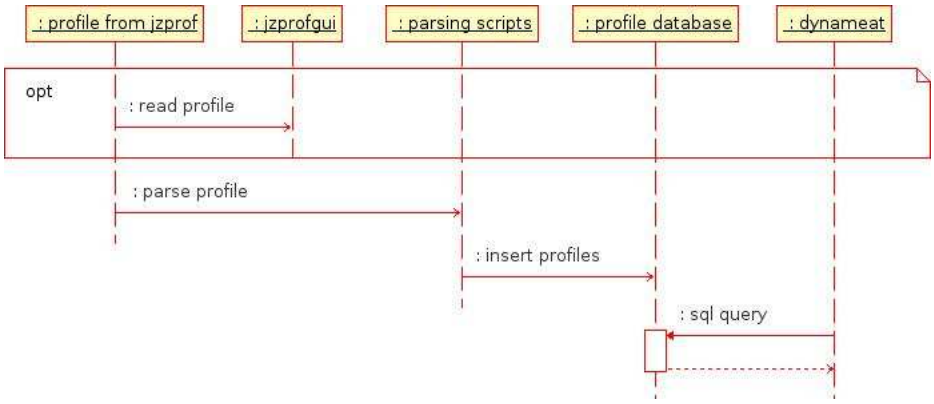

Figure 4.2: DynaMEAT Sequence Diagram

4.1.1 When DynaMEAT is Useful

The toolkit has the potential to be useful to a wide audience, as shown in Figure 4.3. On the lowest level, an analyst may use the function trace to follow the program's execution to verify and validate the source code. If the task at hand is to optimize the execution of the program then the software's developer could use the flat profile to identify which functions have the highest demand (product of visit counts and time spent in the functions), allowing the developer to focus their attention on those high-demand functions. Research groups such as ours can use the toolkit to further study software quality and reliability.

4.2 `jzprof`

4.2.1 Design Decisions and Implementation of `jzprof`

Our primary goal was to develop a functional and useful event-based profiler. We wanted to construct it in such a way that would make it easy to add additional

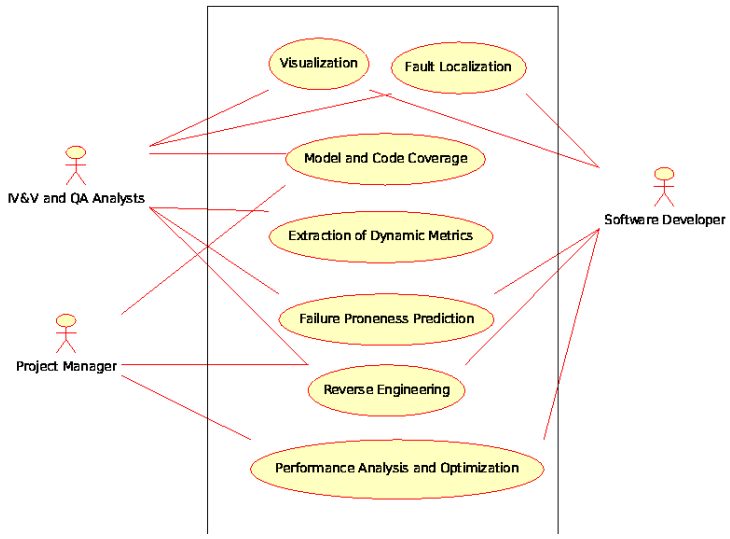

Figure 4.3: Possible Usages of DynaMEAT.

functionality should the need arise. To accomplish this, `jzprof` was built in a modular fashion. (For example, we refer to the visual call graph functionality and function trace functionality as modules.) The modules are independent of each other and developing a new module only requires writing the code for the new module. The rest of `jzprof` does not need to be altered in anyway. This ensures backward compatibility should we need to reanalyze past profiles and provides a means for new functionality and the extraction of other metrics.

`jzprof` is an event-based, or exhaustive [16], profiler because it gathers profiling data by instrumenting the code at every function entrance and exit. In `jzprof`, the profiling data is collected using `gcc`'s *__cyg_profile_func_enter* and *__cyg_profile_func_exit* functions. When compiled with the *-finstrument-functions* option, these two functions will be called each time a function is entered and exited, respectively. The address of the calling function and the address of the function called are passed as

arguments to each function.

`jzprof` utilizes the *rdtsc* instruction to access the Pentium Timestamp Counter to calculate the time spent executing in functions.

```
void __cyg_profile_func_exit(void *this_fn, void *call_site) {
        // Function exit
}

void __cyg_profile_func_enter(void *this_fn, void *call_site) {
        // Function entrance
}
```

Figure 4.4: Example implementation of instrumentation functions using `gcc`.

When a program compiled with `jzprof` executes, `jzprof` stores the function addresses and timestamps as a linked list of structures. When the executing program terminates, `jzprof` traverses the linked list writing the structures to a binary file (jzprof.out). The jzprof.out file then can be processed with the `jzprof` reader to obtain the human-readable profile. `jzprof` compiles to an object file which must be linked with the program to be profiled.

To facilitate easier use of `jzprof`, we created `jzprofgui`, a graphical application written in C# and compiled using Mono. This language was chosen to provide compatibility for both Windows and Linux operating systems. The GTK# runtime is required on both operating systems. To use the `jzprofgui`, the `jzprof` reader must be used to convert the jzprof.out file to an XML file which can then be opened and processed by `jzprofgui`. `jzprofgui` shows the call-graph, flat profile (Figure 4.5), extracted dynamic metrics, and the visual call-graph. The two major goals of

`jzprofgui` were to provide a graphical means of viewing the profiling data and to provide a solution which would be operating system independent.

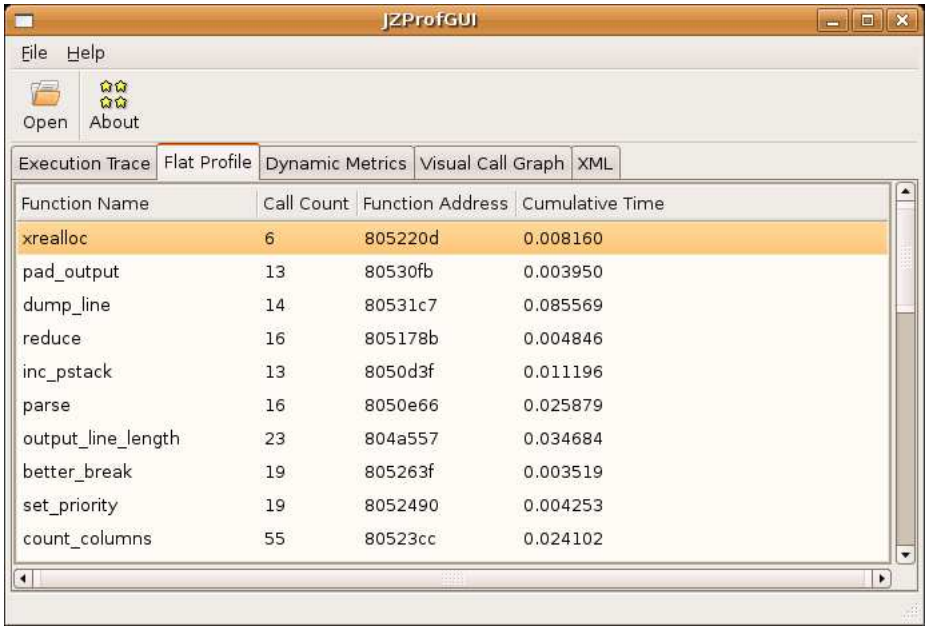

Figure 4.5: `jzprofgui` showing a flat profile. This is the same flat profile presented in Figure 2.2.

`jzprofgui` also shows the execution trace, some metrics of the execution, and the visual call graph. The XML tab displays the profile formatted as XML.

4.2.2 Overhead of `jzprof`

Since all profilers need to cause as little overhead as possible to be effective, `jzprof` was designed to minimize overhead by separating the profiling and analysis events. The profiling work that takes place when the program executes is kept to an absolute minimum. Once profiled, analysis is non-critical and can not affect the results. The

overhead from the profiler is highest when the program being profiled contains many calls to short-running methods as this requires more context-switching and it results in the profiler saving more data.

To measure the overhead introduced by `jzprof` we compared it to `hrprof` [43], another event-based profiler. Fifty random test cases from each of the applications listed in Table 4.1 were executed. The results were calculated by averaging the times from each test case. The test machine was a Dell Optiplex GX260 with an Intel Pentium 4 3.06GHZ processor and 1GB of RAM running Ubuntu 6.06 LTS.

To time the execution, a very small C program was created that gets a timestamp, executes the testcase, and gets another timestamp. The time spent executing is the difference in two timestamps. The Unix `time` utility was not used because it only reports times to the thousandths place which may not allow us to adequately compare the execution times.

The increase in time for `jzprof` from `indent` to `gcc` is most likely due to the length of the linked list created by `jzprof` at runtime. For a larger program like `gcc`, the length of the linked-list becomes much larger than in a smaller program like `indent`.

Application	**With `hrprof`**	**With `jzprof`**	**With No Profiler**
`indent`	0.378948	0.062863	0.012004
`gcc`	0.334561	0.293975	0.115398

Table 4.1: Comparison of Execution Times in Seconds

In our tests, `jzprof` produced significantly less overhead than `hrprof` when tested

with `indent` and only slightly less overhead when tested with `gcc`. The difference in time can be attributed to the fact that `hrprof` produces a profile in the `gprof`-compatible format at the end of the execution, while `jzprof` reserves this functionality for a separate application, the `jzprof` reader. By only writing function data to the profile during execution and omitting other operations we can minimize the overhead introduced by a profiler.

4.2.3 Reading the Profiling Data

The `jzprof` reader is used to analyze the jzprof.out file. The reader opens the jzprof.out file and loads the data into a linked list of structures. Each structure represents either a function entrance or exit and contains the function addresses, a timestamp provided by *rdtscll()* of when the event occurred, and a flag indicating whether the function was entered or exited.

The `jzprof` reader translates the function addresses to function names using the Unix *nm* tool, which extracts symbols from object files.

The reader is controlled by command line arguments. Table 4.2.3 lists the available options. The *-t* option prints a function trace as a tree showing the functions called, the timestamp, and the depth of the calls. The *-f* option prints a flat profile showing the functions called, the number of times each function was called, and timing information for each function.

The *-o* produces a *gmon.out* file which contains the flat profile and call-graph in the `gprof` format. This allows us to be able to view the profile in a familiar layout.

A visual call-graph, or complete calling context tree [46], provides a different

Option	Operation
-t	Print the function trace.
-g	Make a graph.dot file for a visual call graph.
-f	Print the flat profile.
-o	Make `gprof` compatible output (gmon.out file).
-d	Extract and print various dynamic metrics.
-s	Create a SQL file to insert the profile into a database.
-xml	Format the profile as XML (jzprof.xml) for viewing with `jzprofgui`.

Table 4.2: `jzprof` Reader Command Line Options

perspective on the profile. Passing the option *-g* to the reader will create a *graph.dot* file in the *dot* language. Passing this file to the `dot` [13] utility will create a graph of the execution. The nodes of the graph represent the functions called and the arcs represent a transfer of control. Figure 4.6 shows a part of the visual call-graph for an execution of the program `indent`. The numbers alongside each arc indicate the number of times a function was called by another function. In the figure 4.6 the function *indent* calls the function *lexi* 62 times. Arcs that are possible but not taken during the execution will not be included on the visual call-graph.

A visual representation of the call-graph is able to be constructed using the program's trace. Because `jzprof` is an event-based profiler and has instrumented the program's function entrances and exits, we already have the program's complete trace. A trace as a text file can be unmanageable due to the large number of function calls and exits. Formatting the trace as a visual representation allows the call-graph to be

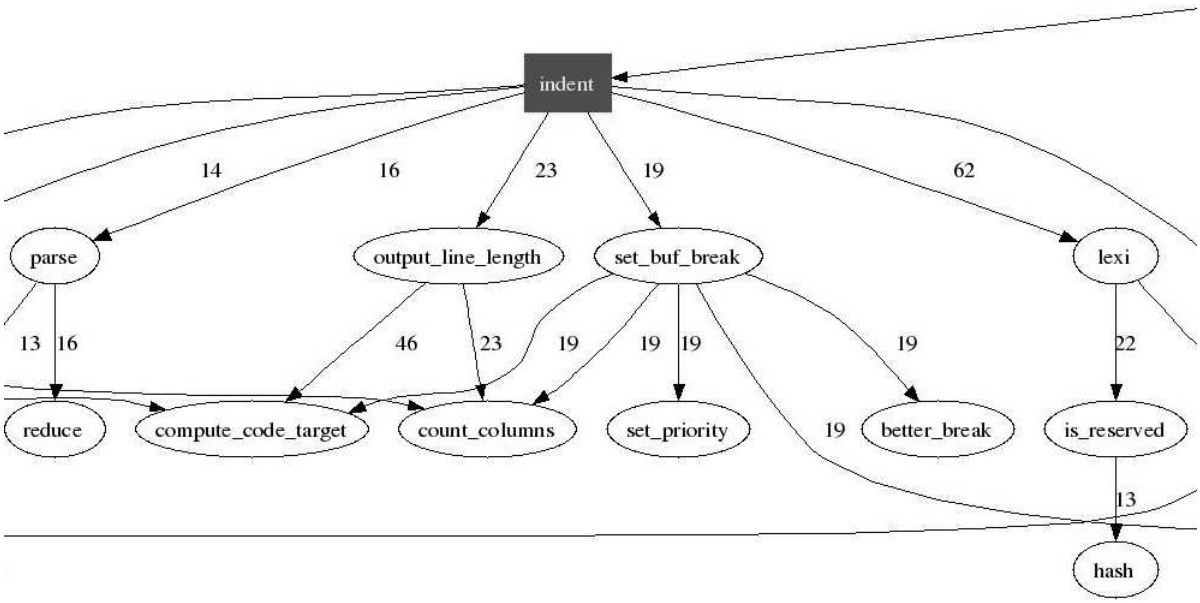

Figure 4.6: Part of a Visual Call-Graph Produced by `jzprof`

studied and analyzed for debugging and source code verification purposes. However, unlike a trace our visual call-graph as it stands now does not give any indication as to the order of functions called.

Constructing a complete calling context tree using a sampling-based profiler would be much more difficult. A calling context tree built using a sampling technique is referred to as an *Approximate Calling Context Tree* [3]. One can never be completely certain that an approximate calling context tree is correct, but we do know that our complete calling context tree is correct because of its event-based construction. Creating an approximate calling context tree using sampling is preferred in instances where time and space constraints do not permit using an event-based technique [3].

A lot of information can be gathered simply by studying the visual call-graph. It is

easy to determine which functions were executed (visit counts), and which functions call which functions (fan-in and fan-out).

4.3 Storing the Profiles in a Database

Some metrics cannot be extracted by `jzprof` due to their inherent nature. These metrics can be derived once the profiles have been inserted into a database. Using a database permits operating on large amounts of data and for deriving metrics over multiple testcases. We chose `MySQL` as our database. To construct a database of the profiles, we:

1. Get the profiles in the `gprof` format. This allows for parsing profiles from `jzprof`, `gprof`, `hrprof`, and any other profiler that utilizes the `gprof` format.

2. Parse the profiles with AWK scripts[1] that formats the function calls and visit counts into SQL insert statements.

3. Execute the SQL insert statements on a `MySQL` database.

When completed, the result will be a table that contains the call graph for each execution. The schema of this table is presented in Figure 4.7. The application table will be created for each application being tested. *id* is an auto-incrementing integer that uniquely identifies the *caller-callee* pair for the testcase in the database. *testname* is the name of the testcase in which *caller* invoked *callee* a total of *num_visits* times. *pass_fail* is a boolean value indicating whether the testcase passes (0) or fails (1). *profiler* is the name of the profiler that generated the profiles. The *percenttime*,

[1]The AWK scripts were written by Maggie Hamill.

selftime, and *childrentime* fields are not currently used but they allow for future examination of the time spent executing functions.

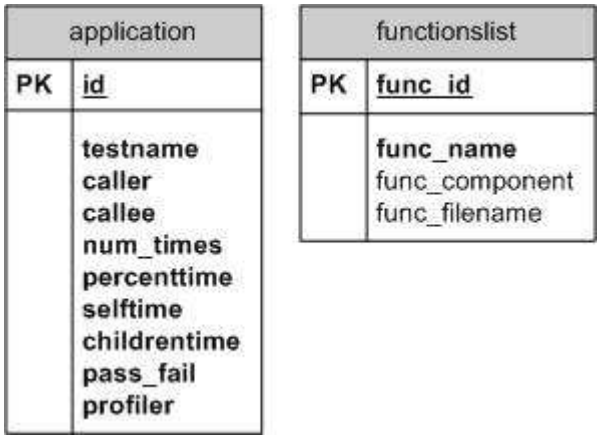

Figure 4.7: The schema of the database tables.

4.4 Extracting Dynamic Metrics from the Database

Once the profiles have been inserted into the database we need to extract the desired metrics. The simplest metrics, such as function visit counts, can be extracted using basic SQL queries. However, more complex metrics, such as fan-ins and fan-outs, can be extracted using `MetricView`.

4.4.1 `MetricView`

`MetricView` was created to help facilitate easy access to the database of profiles. The purpose of this program is to allow users to analyze and visualize the contents of the profiling database. `MetricView` is written in C#.

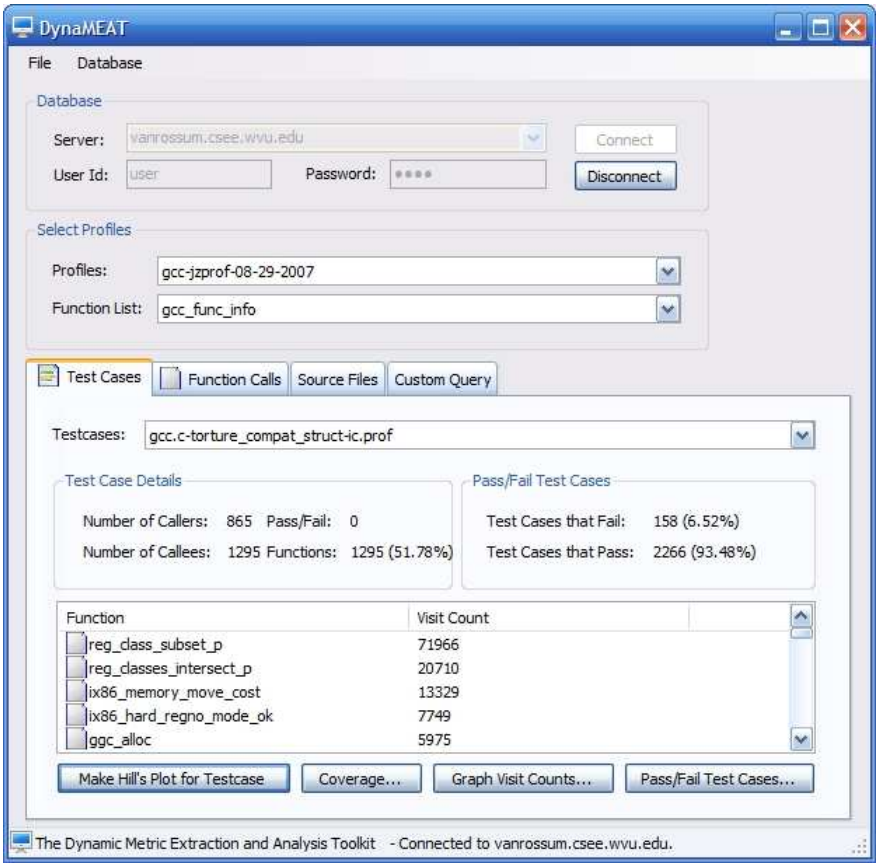

Figure 4.8: Main screen of `MetricView` showing a testcase of GCC.

The main screen of `MetricView` is show in Figure 4.8. Once connected to the profiling database, the user is presented with a list of profiles. When a profile has been selected, the user can then examine testcases or individual functions. With `MetricView`, the user can:

- Get the number and percentage of passed and failed testcases.
- Get a list of the functions called and the number of times each function is called for each testcase.
- See the coverage of the testcase - how many functions and files of the source

code are utilized when executing the testcase.

- Graph the visit counts ($VC_{function}$) for each function in selected testcases (Figure 4.9).
- Produce a list of functions that were never called in any testcase ($FI_{function} = 0$).
- Produce a list of functions that do not call any other functions in any testcase ($FO_{function} = 0$).
- Create a comma-separate values file of fan-outs, fan-ins, and visit counts to be passed to analysis module.
- Analyze the skewness of executions by creating a Hill plot for each testcase.
- Execute custom queries and save the results as a comma-separated values file.

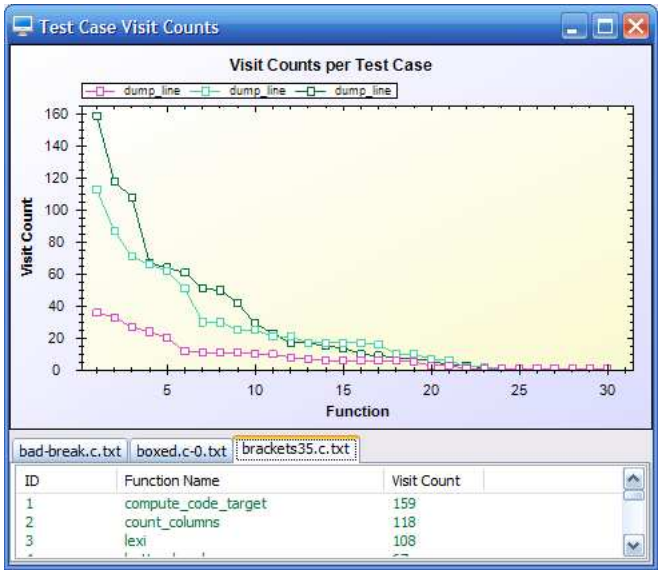

Figure 4.9: Function visit counts for three `indent` testcases.

Chapter 5

Case Studies

The case studies presented here are part of a larger research effort to study the relationships between faults and failures and their effect on software reliability assessments [19, 14, 32].

In these case studies, we applied DynaMEAT to open-source projects of small, medium, and large sizes. The software applications chosen as case studies are `indent` [38], `gcc` [37], and several small test programs developed by Siemens [17]. The applications were chosen based on their size, and availability of past versions and regression testsuites.

5.1 Siemens Test Suite

The Siemens test suite was created by researchers at the Siemens Corporate Research. The suite consist of seven small C programs, along with mutants and testcases for each program. Because of the abundance of mutants and test cases, these programs lend themselves very well to software quality research, especially on the selection

and minimization of test suites [5, 25, 36, 41] and fault-localization [11, 26]. Of the seven programs, to this point we have utilized DynaMEAT to study two of them, `printtokens` and `replace`.

`printtokens` has 7 versions - an oracle and 6 mutants. Each version was profiled using `jzprof` and the resulting profiles were parsed and inserted into the database. `replace` has 32 versions - an oracle and 31 mutants. The profiles of these versions were also parsed and inserted into the database. Again, `MetricView` was used to extract the required metrics from the database for analysis.

5.2 `indent`

`indent` is a GNU open-source, code beautification tool for C source code. It is a medium-sized program, containing approximately 10,000 lines of C code. A regression test suite is available, consisting of 155 test cases. In our research we utilized the regression test suite of version 2.2.9 with the binary from version 2.2.0 instrumented with `jzprof`.

The profiles of the test suite executions created by `jzprof` were parsed by the database scripts and then inserted into our profile database. `MetricView` was then used to create the visit count, fan-in, and fan-out matrices. The matrices would then be used by our research group to study the executions.

Using `MetricView` we can also analyze how many functions were executed from each source file for each testcase. Figure 5.1 is a pie chart created by `MetricView` that shows the the number of functions in each source code file of `indent` that was executed in one testcase. For example, from Figure 5.1 we can tell that 11 functions

in backup.c were executed in the testcase. Applying static analysis to determine the total number of functions in each source file would allows us to determine the coverage of each testcase.

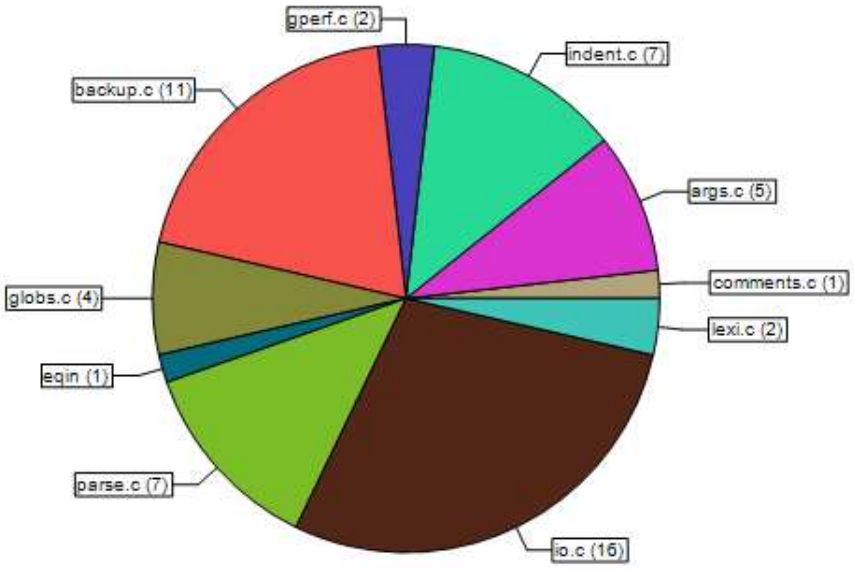

Figure 5.1: Number of functions in `indent` source files executed during an `indent` testcase.

`MetricView` can also show the number of functions executed in one test case relative to entire test suite. Figure 5.2 shows the number and percentage of functions executed during the execution of one test case. Of all the functions called during the entire test suite, this particular test case executed 31, or 55.36%, of those functions.

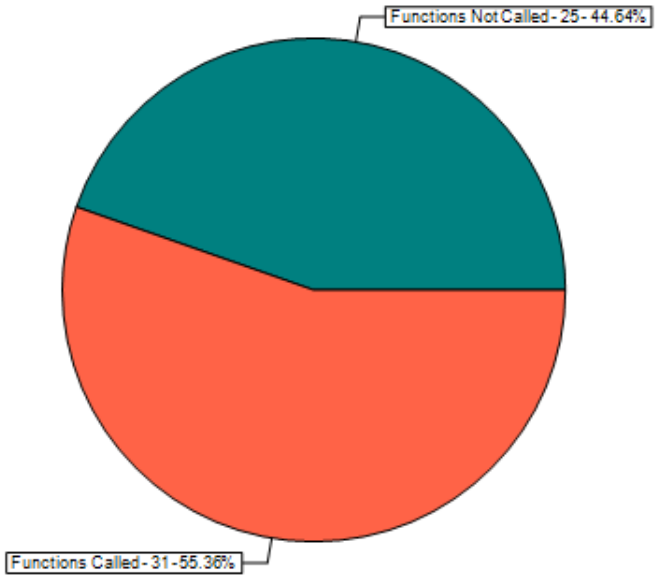

Figure 5.2: Number of functions executed during one `indent` test case execution created by `MetricView`.

5.3 `gcc`

`gcc`, the GNU Compiler Collection, is a software package to compile many different programming languages, however, we only focused on `gcc`'s C compiler, `cc1`. `cc1` is composed of approximately 300,000 lines of source code. In our tests, we used the regression test suite (2,424 testcases) from `gcc` version 3.3.3 on the compiled binary from `gcc` version 3.2.3 instrumented with `jzprof`. Once the profiling was complete, the profiles were parsed and inserted into the database.

Not only does the `gcc` test suite contain more test cases than the `indent` test suite, but the executions of each `gcc` test case are much larger than any of `indent`'s. The table in the database containing the `gcc` profiles had over 5 million rows, compared

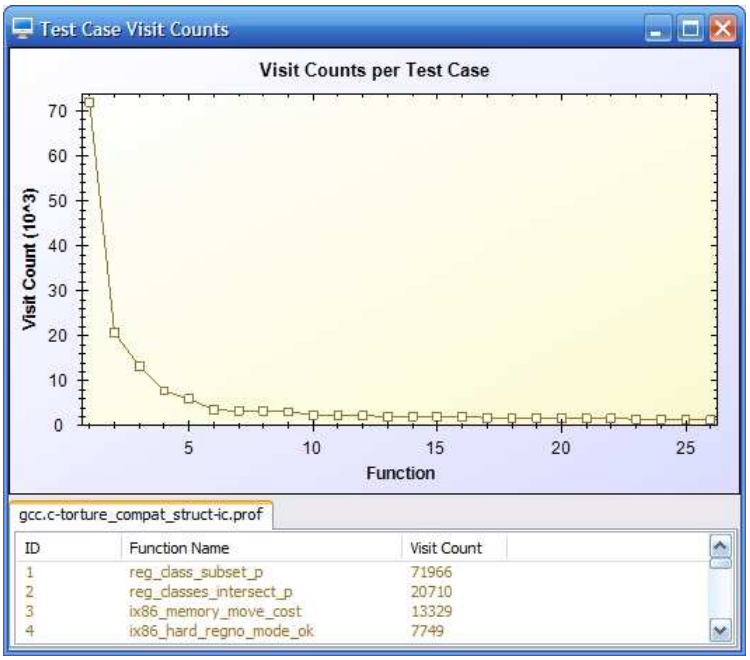

Figure 5.3: Visit counts for a `gcc` testcase.

to less than 7 thousand for `indent`. `MetricView` did not have any problem with the increase of size. Like with `indent`, we used `MetricView` to create visit counts, fan-in, and fan-out matrices for analysis.

5.4 Using `MetricView` to Analyze the Skewness of Software Executions

It is commonly said that during a program's execution 20% of the program's functions will be visited 80% of the time. Using `MetricView` we can see if that rule holds true for the case studies.

The skewness of an execution can be determined using the Hill estimator [21], which is a method to estimate the tail index α of a Pareto type model given by

$$1 - F(x) = P[X > x] = x^{-\alpha}L(x) \tag{5.1}$$

where $L(x)$ is slowly varying as $x \to \infty$. Let $X_1, X_2, ...X_n$ denote the function visit counts ordered in descending order, such that $X_{(1)} \geq X_{(2)} \geq ... \geq X_{(n)}$. The basis of the Hill Estimator is to sample from the part of the distribution that most resembles a Pareto distribution. Therefore, we choose $k < n$ and compute the Hill estimator

$$H_{k,n} = \frac{1}{k}\sum_{i=1}^{k} \log X_{(i)} - \log X_{(k+1)}. \tag{5.2}$$

For each value of k we get an estimate of the tail index, $\alpha_{k,n} = \frac{1}{H_{k,n}}$. Typically, the estimates of the tail index $\alpha_{k,n}$ are plotted as a function of k. When k is small, the Hill plot usually varies greatly, but as k increases the plot stabilizes as more data points in the tail of the distribution are included. Once the plot stabilizes, we can infer the value of the tail index α. The lack of stabilization is a strong indication that the data is not consistent with the heavy-tailed distribution (5.1).

It follows that if $1 < \alpha \leq 2$, the distribution has a finite mean and an infinite variance. If $\alpha \leq 1$, the distribution has an infinite mean and infinite variance (a few functions are called substantially more than the other functions).

`MetricView` includes the ability to automatically create Hill plots of executions. To apply the Hill estimator, `MetricView` first orders the function visit counts of a testcase in descending order (Figure 5.4). `MetricView` then applies the Hill estimator to the function visit counts and graphs the result (Figure 5.5).

Figure 5.5 shows the Hill plot created from the visit counts in Figure 5.4. The

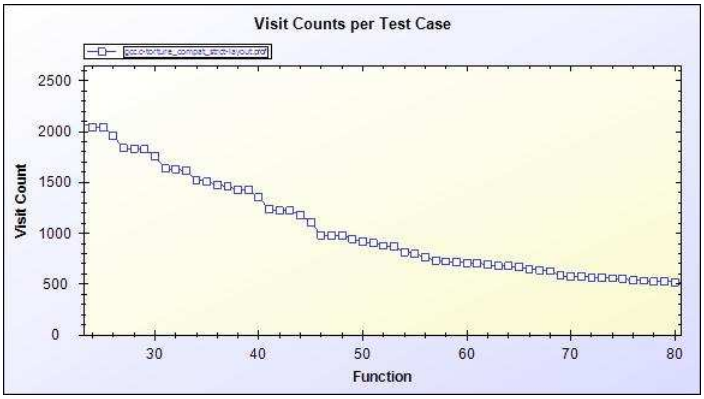

Figure 5.4: Visit counts for an execution of `gcc`.

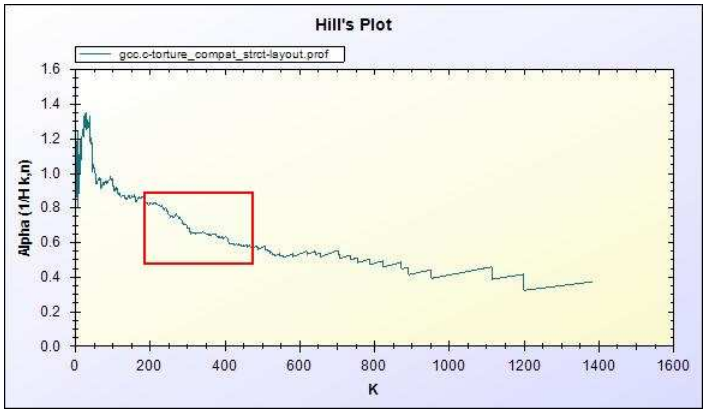

Figure 5.5: Hill plot of the testcase shown in Figure 5.4.

area of the graph in the square is where the plot stabilizes. Zooming in on this area allows us to better infer the value of α (Figure 5.6).

From Figure 5.6, we can estimate that α is approximately 0.65, which indicates that the mean and variance of the distribution are both infinite. We can conclude that this particular execution of `gcc` is skewed and that it follows the Pareto distribution.

The Hill estimator is not a good measure of skewness for `indent` and the programs in the Siemens Test Suite because those applications do not contain enough functions

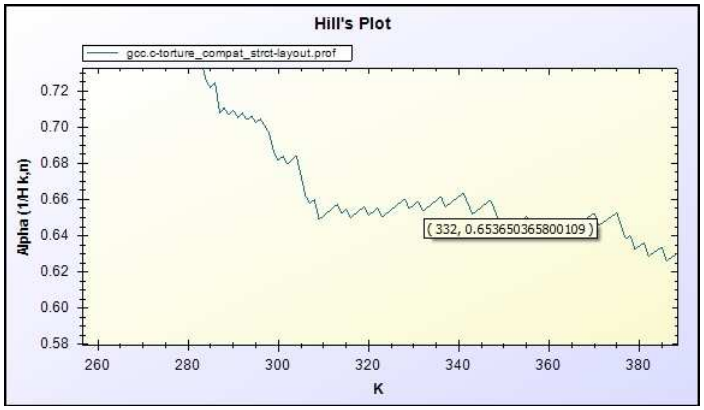

Figure 5.6: Hill plot of the testcase shown in Figure 5.4.

to provide an accurate value for α.

5.5 Conclusions of the Case Studies

Using the components of DynaMEAT allowed us to profile and collect dynamic metrics at a much faster pace than previously possible. Additionally, we can now easily analyze other aspects of the execution such as skewness and testcase coverage.

Chapter 6

Related Work and Contributions

6.1 Profilers

6.1.1 `gprof`

UNIX `gprof` [16] is a sampling-based, call-graph profiler for C. `gprof` counts the number of calls to each function by instrumenting the code at compile time using `gcc`'s *-g* and *-pg* options. However, to calculate the time spent in each function, `gprof` samples the program counter to determine the state of the executing program. This can lead to inaccuracies when a function executes entirely within the sampling interval. For example, if a function executes entirely within the sampling interval it will not be detected by the profiler, as demonstrated in Figure 2.1.

`gprof` contains a few nuances that the user should be aware of prior to using `gprof`. For instance, `gprof` does not count time spent in calls to `sleep()`, which can skew the profiles. Additionally, when calculating the time spent in each function, `gprof` incorrectly assumes that each call to a function takes the same amount of time

and does not differentiate between each call to a function. Because of this, it has been concluded that the use of `gprof` should be limited to situations where function descendants take a constant time to execute [42].

6.1.2 `hrprof`

`hrprof` [43], or the High Resolution Profiler, is an event-based profiler for C. Like `jzprof`, `hrprof` uses the `gcc` functions *__cyg_profile_func_enter()* and *__cyg_profile_func_exit()* to instrument the source code at compile time. By making use of the Pentium Timestamp Counter, `hrprof` provides more accurate timing information than `gprof`. The Pentium Timestamp Counter is often used in this manner [34, 45]. Additionally, `hrprof` produces the profiles in the standard `gprof`-compatible format.

6.1.3 Commercial Profilers

In addition to open source profilers, there are also many commercial profilers available. Intel's ThreadProfiler [22] assists developers with creating multi-threaded applications and applications built for multi-processor systems. ANTS Profiler [24] by RedGate software profiles .NET applications on Windows. JProbe [23] from Quest Software analyzes Java code.

6.2 Visual Call-Graphs

A lot of work has been done on visual call-graphs, however, our work adds this functionality to a profiler. The type of visual call-graph presented in this thesis is referred to as a *Calling Context* tree in [2]. Taking it one step farther, [46] labels our

visual call-graph as a *Complete Calling Context Tree* because the call-graph includes all caller/callee pairs during an execution, as opposed to an Approximate Calling Context Tree which is constructed via sampling [3]. Constructing a calling context tree by instrumenting function entrances and exits was proposed in [2] and [39].

6.3 Dynamic Metrics

Work involving the definition of dynamic metrics is plentiful [4, 6, 8, 9, 44]. Dynamic metrics are not limited to being useful only in software quality and reliability. An intrusion detection method described in [10] monitors the runtime behavior of known safe software. When an unknown software executes and its behavior differs from the behavior of known safe software the proverbial red flag will be raised.

6.4 Contributions

This thesis provides a toolkit for profiling software executions, to automate the extraction of dynamic metrics for subsequent analysis. The following are contributions provided by this thesis and DynaMEAT.

- Identified requirements for creating an effective and useful profiler.
- Developed `jzprof`, a very capable, small, and efficient C profiler with the ability to capture extensive profiling data:
  - The function trace of the execution shows when functions are entered and exited.

  - The visual representation of the call-graph provides a graphical means of viewing transfer of control during the execution.
  - The flat profile includes function timing information that often can not be determined by `gprof`.

- `jzprofgui` is a cross-platform tool to view profiles from `jzprof`. It allows the analyst to view the flat-profile, call-graph, execution trace, and visual call-graph from an easy to use graphical interface.

- Scripts for preparing the data to be inserted into a database. These scripts can operate on any profile in `gprof` format providing flexibility to use any profiler and the ability to compare `gprof` and `jzprof` profiles.

- A database schema to store the profiles. Having a uniform method of storing profiles allows for easy collaboration and sharing.

- `DynaMEAT` gives the analyst the power to extract metrics from a database and to derive any new metrics. Using `DynaMEAT` the analyst can extract the following metrics:
  - Fan-in and fan-out for each testcase or aggregated over all testcases.
  - Function visit counts for each testcase or aggregated over for all testcases.
  - Any other metric that can be derived on the function level.

- `DynaMEAT` can create Hill plots for individual testcases to analyze the skewness of executions.

Chapter 7

Conclusions

The Dynamic Metric Extraction and Analysis Toolkit provides a complete methodology for collecting, extracting, and analyzing dynamic metrics. `jzprof` is an accurate, light-weight profiler that provides more profiling data than `gprof`. Any function-level metric can be derived from the profiles stored in the database. `MetricView` provides convenient access to the database which allows for faster analysis of the profiles.

The toolkit has the potential to be used for a long period of time. It does not require any notable maintenance and the only possible required changes would be to extend the functionality of `jzprof` or `MetricView`.

The Dynamic Metric Extraction and Analysis Toolkit allows researchers to focus their attention more on the study of software executions rather than on the collection of metrics required for the analysis.

7.1 Future Work

There are several ways in which DynaMEAT can be improved to offer additional functionality.

- **Using Time as a Dynamic Metric**

  Incorporating time as a dynamic metric may provide a new way of analyzing and comparing profiles. `jzprof` includes the ability to time the execution of functions, but this timing data is not currently inserted into the database. Modifying the AWK scripts to gather this data could provide lots of new metrics and provide a new perspective on the profiles.

- **Function Trace Analysis**

  `jzprof` captures function traces but we have not developed any method to insert these traces into the database. Having the traces in the database would allow for fast detection of cycles, allow for analyzing function returns, and provide an ordered execution of the program.

- **Visual Call-Graph**

  The visual call-graph produced by `jzprof` can be improved in the following ways. First, when viewing the visual call-graph it is impossible to determine the order of execution. Secondly, for large executions the visual call-graph can potentially be very large resulting in many nodes and arcs. To simplify and compact the visual call-graph, functions contained in a cycle could be combined into one node which would then be substituted for the cycle.

Bibliography

[1] A. J. Albrecht and J. E. Gaffney. Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation. *IEEE Transactions on Software Engineering*, 9(6):639–648, 1983.

[2] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 85–96, New York, NY, USA, 1997. ACM Press.

[3] M. Arnold and P.F. Sweeney. Approximating the Calling Context Tree via Sampling, 2007.

[4] Alessandro Bianchi, Danilo Caivano, Filippo Lanubile, and Giuseppe Visaggio. Evaluating Software Degradation through Entropy. In *METRICS '01: Proceedings of the 7th International Symposium on Software Metrics*, page 210, Washington, DC, USA, 2001. IEEE Computer Society.

[5] Lionel Briand and Yvan Labiche. Empirical Studies of Software Testing Techniques: Challenges, Practical Strategies, and Future Research. *SIGSOFT Soft-*

ware Engineering Notes, 29(5):1–3, 2004.

[6] F. Brito, e Abreu, and W. Melo. Evaluating the Impact of Object-Oriented Design on Software Quality. In *3rd International Software Metrics Symposium*, pages 90–99, 1996.

[7] Bruno De Bus, Dominique Chanet, Bjorn De Sutter, Ludo Van Put, and Koen De Bosschere. The Design and Implementation of FIT: A Flexible Instrumentation Toolkit. In *PASTE '04: Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 29–34, New York, NY, USA, 2004. ACM Press.

[8] Jana Dospisil. Software Metrics, Information and Entropy. In *Practicing Software Engineering in the 21st Century*, pages 116–142, Hershey, PA, USA, 2003. IGI Publishing.

[9] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic Metrics for Java. *ACM SIGPLAN Notices*, 38(11):149–168, 2003.

[10] Sebastian Elbaum and John C. Munson. Intrusion Detection Through Dynamic Software Measurement. In *ID '99: Proceedings of the 1st Conference on Workshop on Intrusion Detection and Network Monitoring*, pages 5–5, Berkeley, CA, USA, 1999. USENIX Association.

[11] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly Detecting Relevant Program Invariants. In *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, New York, NY, USA, 2000. ACM.

[12] Nathan Froyd, John Mellor-Crummey, and Rob Fowler. Low-Overhead Call Path Profiling of Unmodified, Optimized Code. In *ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing*, pages 81–90, New York, NY, USA, 2005. ACM Press.

[13] Emden R. Gansner and Stephen C. North. An Open Graph Visualization System and its Applications to Software Engineering. *Software - Practice and Experience*, 30(11):1203–1233, 2000.

[14] Katerina Goseva-Popstojanova, Margaret Hamill, and Xuan Wang. Adequacy, Accuracy, Scalability, and Uncertainty of Architecture-based Software Reliability: Lessons Learned from Large Empirical Case Studies. *ISSRE '06. 17th International Symposium on Software Reliability Engineering*, pages 197–203, 2006.

[15] Katerina Goseva-Popstojanova and Arin Zahalka. The Impact of Dynamic Metrics on Indentification of Failure Prone Parts of Software. In *NASA Software Assurance Symposium*, 2007.

[16] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A Call Graph Execution Profiler. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, 1982.

[17] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An Empirical Study of Regression Test Selection Techniques. *ACM Transactions on Software Engineering and Methodolody*, 10(2):184–208, 2001.

[18] Yann-Gael Gueheneuc and Tewfik Ziadi. Automated Reverse-engineering of UML v2.0 Dynamic Models. In *Proceedings of the 6th ECOOP Workshop on Object-Oriented Reengineering*, 2005.

[19] Margaret L. Hamill. Empirical analysis of software reliability. Master's thesis, West Virginia University, 2006.

[20] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A Methodology for Controlling the Size of a Test Suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, 1993.

[21] B.M. Hill. A Simple General Approach to Inference about the Tail of a Distribution. *The Annals of Statistics*, 3:1163–1174, 1975.

[22] http://www.intel.com/cd/software/products/asmo na/eng/286749.htm. Intel thread profiler 3.1 for windows.

[23] http://www.quest.com/jprobe/. Jprobe.

[24] http://www.red gate.com/products/ants_profiler/index.htm. Ants profiler.

[25] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria. In *ICSE '94: Proceedings of the 16th International Conference on Software Engineering*, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[26] James A. Jones and Mary Jean Harrold. Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *ASE '05: Proceedings of the*

20th IEEE/ACM International Conference on Automated Software Engineering, pages 273–282, New York, NY, USA, 2005. ACM.

[27] D. Kafura and G. R. Reddy. The Use of Software Complexity Metrics in Software Maintenance. *IEEE Transactions on Software Engineering*, 13(3):335–343, 1987.

[28] Naveen Kumar, Bruce R. Childers, and Mary Lou Soffa. Low Overhead Program Monitoring and Profiling. In *PASTE '05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 28–34, New York, NY, USA, 2005. ACM Press.

[29] James R. Larus and Thomas Ball. Rewriting Executable Files to Measure Program Behavior. *Software - Practice and Experience*, 24(2):197–218, 1994.

[30] Thomas J. McCabe and Charles W. Butler. Design Complexity Measurement and Testing. *Communications of the ACM*, 32(12):1415–1425, 1989.

[31] Andriy V. Miranskyy, Nazim H. Madhavji, Mechelle S. Gittens, Matthew Davison, Mark Wilding, and David Godwin. An Iterative, Multi-level, and Scalable Approach to Comparing Execution Traces. In *ESEC-FSE '07: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 537–540, New York, NY, USA, 2007. ACM Press.

[32] Ranganath Perugupalli. Empirical Assessment of Architecture-Based Reliability of Open-Source Software. Master's thesis, West Virginia University, 2004.

[33] Y. L. Traon R. Delamare, B. Baudry. Reverse-engineering of UML 2.0 Sequence Diagrams from Execution Traces. In *Workshop on Object-Oriented Reengineering in Conjunction with ECOOP '06*, Nantes, France, 2006.

[34] J. Regehr and J. Stankovic. Augmented CPU Reservations: Towards Predictable Execution on General-Purpose Operating Systems. In *RTAS '01: 7th Real-Time Technology and Applications Symposium*, 2001.

[35] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and Optimization of Win32/Intel Executables using Etch. In *NT '97: Proceedings of the USENIX Windows NT Workshop on the USENIX Windows NT Workshop 1997*, pages 1–1, Berkeley, CA, USA, 1997. USENIX Association.

[36] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites. In *ICSM*, pages 34–43, 1998.

[37] Free Software Foundation. GCC, the GNU Compiler Collection.

[38] Free Software Foundation. GNU Indent.

[39] J. M. Spivey. Fast, Accurate Call Graph Profiling. *Software - Practice and Experience*, 34(3):249–264, 2004.

[40] Amitabh Srivastava and Alan Eustace. Atom: A System for Building Customized Program Analysis Tools. *ACM SIGPLAN Notice*, 39(4):528–539, 2004.

[41] Sriraman Tallam and Neelam Gupta. A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization. In *PASTE '05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 35–42, New York, NY, USA, 2005. ACM.

[42] Dominic A. Varley. Practical Experience of the Limitations of Gprof. *Software - Practice and Experience*, 23(4):461–463, 1993.

[43] Pace Willisson. High Resolution Profiler.

[44] Sherif M. Yacoub, Hany H. Ammar, and Tom Robinson. Dynamic Metrics for Object Oriented Designs. In *METRICS '99: Proceedings of the 6th International Symposium on Software Metrics*, page 50, Washington, DC, USA, 1999. IEEE Computer Society.

[45] Shelley Zhuang, Kevin Lai, Ion Stoica, Randy Katz, and Scott Shenker. Host Mobility using an Internet Indirection Infrastructure. Technical Report UCB/CSD-02-1186, EECS Department, University of California, Berkeley, Jul 2002.

[46] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. Accurate, Efficient, and Adaptive Calling Context Profiling. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 263–271, New York, NY, USA, 2006. ACM Press.