Using Full System Simulation to Perform Non-Intrusive Measuring of Code Coverage

ROBIN WESTBERG

Master’s Degree Project
Stockholm, Sweden 2005

TRITA-NA-E05039
Using Full System Simulation to Perform Non-Intrusive Measuring of Code Coverage

ROBIN WESTBERG

TRITA-NA-E05039

Master’s Thesis in Computer Science (20 credits) at the School of Computer Science and Engineering, Royal Institute of Technology year 2005
Supervisor at Nada was Inge Frick
Examiner was Stefan Arnborg
Abstract

Coverage measurement is a way of finding out how much of a program has been used in terms of things such as number of source code lines executed or number of branches taken. This is commonly used in the process of software testing, to determine how effective a certain test is. Most automated tools for coverage measurements modify the source code of the program under inspection.

This master's project examines how to do this on the instruction level instead, by using the full system simulator Simics. This method allows the measurements to be taken completely non-intrusively.

The project evaluates different types of coverage and their applicability for measurement on the instruction level. A prototype tool is developed that implements some of these coverage metrics. As a practical example, the prototype is used to measure the effectiveness of Simics’ own test suite.

Användandet av fullsystemsimulation för att utföra icke-intrusiv mätning av kodtäckning

Att mäta kodtäckning är att ta reda på hur mycket av ett program som har använts i temner av olika saker så som antal källkodsradar ködda eller antalet förgreningar utförda. Detta har man användning för när man testar programvara, för att ta reda på hur effektivt ett visst test är. Den vanligaste metoden bland existerande verktyg som gör denna mätning är att modifiera källkoden till det program som ska undersökas.

Detta examensarbete undersöker hur detta kan göras på instruktionsnivå istället, genom att använda fullsystemsimulatorn Simics. En fördel med detta är att det blir möjligt att utföra mätningarna utan att påverka programmet på något sätt.

# Contents

1 Introduction .................................................. 1
   1.1 Background ............................................. 1
   1.2 Goals of the project ................................... 2
   1.3 Limitations ............................................. 2
   1.4 Organization of the report ............................ 2

2 Test coverage .................................................. 3
   2.1 Coverage on different levels ......................... 3
   2.2 Common metrics ........................................ 4
      2.2.1 Statement coverage ............................... 5
      2.2.2 Branch coverage .................................. 5
      2.2.3 Condition coverage ............................... 6
      2.2.4 Condition/decision coverage ..................... 7
      2.2.5 Path coverage ..................................... 7
   2.3 Other metrics .......................................... 8
   2.4 Summary of metrics and their implementation suitability ........ 8
   2.5 Monitoring and intrusion ............................. 9
      2.5.1 Source code instrumentation ..................... 9
      2.5.2 Object code instrumentation ...................... 9
      2.5.3 Hardware monitoring ............................. 10
      2.5.4 Simulators ....................................... 10
      2.5.5 Examples of existing tools ...................... 10

3 Simics ......................................................... 12
   3.1 General description .................................... 12
   3.2 Scripting and API ..................................... 12
   3.3 Contexts and virtual memory ........................ 13
   3.4 Profiling .............................................. 13
   3.5 Symbolic debugging ................................. 13
   3.6 Test suite ............................................ 14

4 The coverage tool ............................................ 15
   4.1 The prototype ......................................... 15
      4.1.1 Design goals ..................................... 16
      4.1.2 Environment ....................................... 16
      4.1.3 Overview on how the prototype works .......... 16
      4.1.4 System call interception details ............... 17
      4.1.5 About binary files .............................. 17
Chapter 1

Introduction

1.1 Background

Testing is an important aspect of software development today. It is not uncommon for testing to take up half of the time set aside for development [3]. With this in mind, there is obviously an interest in evaluating the effectiveness of testing, and one way is by measuring coverage.

Coverage is a way of expressing how much of the program under test that has actually been exercised. For example, one measurement could be presenting a percentage value of how many lines of source code that has been executed, compared to the total number of lines in the program.

This measuring can be done manually; one way is to print out all the source code on paper. By supplying the initial conditions and following the flow through the source code as it would be executed, one can mark each visited line with a marker pen. When the printed source code shines yellow, you know you have accomplished full coverage.

While this method has actually been used in practice in the past [2], it becomes cumbersome as programs grow larger and larger. Therefore it is of interest that this process is automated.

A common approach to building an automated tool that measures coverage is to insert instrumentation in either the source code or the binary program executable. There are also tools that use special hardware to accomplish the task without modifying the software at all. These are mostly used for embedded software.

When instrumenting the code it is inevitably changed in some way, and this change could effect the execution in different ways. This would mean that the measured behavior differs from how the program runs when not being monitored, and is known as the probe effect [7].

One approach to avoid the probe effect would be to measure coverage through a simulator. Since all measurements taken by a simulator are done outside of the simulated world, it will not affect the simulated program in any way. This is an example of completely non-intrusive measuring.

Simics is a full system simulator which can run unmodified software including operating systems and device drivers. It is effective enough to boot an operating system within a reasonable time frame. Simics has good capabilities
for instrumenting a simulation without affecting its behavior in any way.

1.2 Goals of the project

The goals of the project were to evaluate different coverage metrics, and their suitability for implementation with the help of a simulator. It was also a goal to implement one or more coverage metrics that were found to be suitable, and to implement the functionality needed to isolate a user process in a full system simulator. Furthermore, it was of interest to show that the method of measuring coverage through a simulator could be used on complex applications that use mechanisms such as dynamic loading of code, and not just small test applications. Since the intention is to try the coverage tool on a real application test suite, this has priority over implementing more exotic forms of coverage metrics.

1.3 Limitations

There are very many different coverage metrics defined, most of which involve advanced knowledge of the flow through the program on the source code level. Such information is difficult to obtain through a simulator, as it sees execution on the instruction level. Because of things like compiler optimizations, it is not obvious how flow through instructions translates to flow through source code. This means that complex source code based coverage metrics were difficult to implement. This project did only investigate simpler metrics, and time constraints also limited the number of coverage metrics that were implemented.

1.4 Organization of the report

Chapter 2 describes the theory on coverage measurement, common metrics, and existing tools that are used to obtain this information. It also discusses the difference between measuring coverage on the source code level and the instruction level.

Chapter 3 describes the system simulator used for this project in detail, including the test suite and the useful facilities provided.

Chapter 4 describes the development of the coverage tool prototype, and the various problems that were encountered and their solutions. It also presents the information obtained when examining the Simics test suite.

Chapter 5 includes possible ways to extend the tool. It examines the possibilities for more advanced metrics, and better ways to present the information. It also discusses the issues that could arise when adapting the tool to work in different environments such as with other processors and operating systems. Possible solutions to these problems are also examined.

The final chapter 6 presents the conclusions that could be drawn from this project.
Chapter 2

Test coverage

To measure coverage is to quantify in some way just how much of a certain program has been used during an execution. This is commonly used in combination with software testing, especially when constructing various forms of input data and checking whether the program gives the expected results. One set of input data is called a test case, and these are usually grouped together in a larger test suite [2], whose goal is to test as much of the program as possible. The perfect test suite would leave no aspect of the program untested. But it is not obvious how to know whether a particular test or even if the entire test suite is effective, perhaps it is only checking a small part of the functionality that the program has. This is where measuring coverage comes in handy.

2.1 Coverage on different levels

Most of the literature on test coverage deals with source code coverage. This is natural, because the coverage information is usually most relevant to what the programmer has actually written. But coverage can just as well be applied on other levels that are closer to the actual instructions executed by the processor.

Figure 2.1 shows the basic levels that code goes through from source code to finally executing instructions on a processor. Source code refers to what the programmer has actually written in a high level language, but it is also possible to write programs in assembly code which closely corresponds to the instructions that a processor is capable of executing. Assembly code can usually also be generated by the high level language compiler. The object code is the result of compilation of the source code or assembly code to machine code instructions. Before it can be used though, it has to go through the process of linking that combines the object code from multiple sources including libraries to produce an executable file. The instructions in memory refer to the actual instructions that are executed by the processor when the program of interest is run. The instructions can generate code dynamically, and during execution it is possible that more libraries are linked dynamically.

It is possible to measure coverage on all these different levels. For the purpose of this project the focus is on measuring coverage on the instructions in memory, since this is what the simulator sees. This does not come without its problems though. One problem with measuring coverage on the instruction level is that
it can be difficult or impossible to trace the executed instructions back to the originating source code. This is mostly a result of compiler optimization, and is discussed in more detail in the sections detailing the creating of the prototype for the coverage tool.

Advantages of instruction level coverage include the possibility to measure basic coverage on programs for which no source is available. But as coverage analysis mostly is a tool to aid the development of software it can be assumed that the source code is usually available when coverage measuring is desired.

Another advantage is that it also is possible to measure coverage on code that has been generated during the execution of the program. For this type of generated code there does not exist any source code to measure coverage on, which leaves instruction level coverage as the only option.

## 2.2 Common metrics

Many different coverage metrics have been defined. According to [11], the best known metrics are statement, branch and path coverage. In [2] these are known as C1, C2 and C* respectively. Statement coverage means that every statement in the source code should be executed at least once. Branch coverage means that every branch (transfer of control) in the program should have been taken at least once. Finally path coverage means that every possible path through the program has been exercised. A path is a unique combination of branches. For example, a program that contains two branches will have up to four possible paths. As the number of paths grows exponentially with the number of branches, the total number quickly becomes huge even for smaller programs.

The different metrics are usually defined in terms of the source code. For this project, the measuring was done on the instruction level instead. This has a
few interesting effects. In [2] an argument is made that full statement coverage on the source code level does not necessarily have to indicate that full statement coverage on the instruction level has been achieved. To reach that, it is needed to achieve full branch coverage on the source code. However the reverse is true, meaning that having full statement coverage on the instruction level guarantees full statement coverage on the source code level. This is an argument in favor of measuring coverage on the instruction level.

There is however a required condition for this to hold true, and that is that certain cases of dead code [1] must be disregarded. Dead code means statements in the source code that are never executed, and in some cases the compiler can detect this and not generate any corresponding instructions in the final executable. There is no way for the instruction level coverage measurer to detect that the corresponding source code lines should be counted as not executed. Source code lines that do not generate any corresponding code such as comments and other constructs in the source language are also not considered. The same is true for lines that do not generate any code due to compiler optimization, which is discussed in section 4.2.1.

There are many other metrics defined. Generally they come in somewhere between C2 and C*, since C2 is usually considered a minimum requirement for thorough testing, and C* is generally impossible to achieve for anything but the most trivial of programs, since the number of tests required grows exponentially.

Following is a more in-depth look at the basic metrics, and a discussion of how suitable they are for implementing in this project. More detailed descriptions and other measures can be found in [8] and [4].

### 2.2.1 Statement coverage

This metric, also known as C1, measures how many of a program’s total statements or lines are executed. This is a weak metric, since failing to have full statement coverage would mean that there are parts of the program that are completely untested. A common weakness is the inability to correctly analyze certain control structures. For example, the following C code fragment

```c
int *p = NULL;
if (condition)
  p = &variable;
*p = 4711;
```

If condition evaluates to true, statement coverage would be reported as 100%. But if condition should ever become false, this code would fail. Therefore the reported 100% coverage does not mean that the program is bug free.

### 2.2.2 Branch coverage

The demonstrated weakness above is overcome with branch coverage, which is also known as C2 and decision coverage. This metric takes into account whether boolean expressions tested in control structures (such as if and while) evaluated to both true and false. In other words, for full coverage it requires that all conditional branches have both been taken and not taken.
For the above example, this measure would only report 100% coverage if condition has evaluated to both true and false. There are however still control structures that are problematic for this model. An example could be the following code:

```c
if (a && (b || c()))
    statement1;
else
    statement2;
```

It is possible for branch coverage to consider the above structure fully covered even though the function c() was never called, since branch coverage only requires the full expression checked by the if statement to evaluate to both true and false. This expression will evaluate to true when a and b is true, and to false when a is false. This is however only a problem in languages that uses short-circuit boolean evaluation such as C. The term short-circuit boolean evaluation means that evaluation of boolean expressions stop as soon as the result is determined, even if there are still terms left in the expression. In the above example it will stop evaluation if seeing that a is false, because now there is no way for the entire expression to have another result than false. There is however other languages that always evaluate the full statements.

When measuring branch coverage on the instruction level it first must be determined what this actually means. There are no complex boolean expressions evaluated on this level, because this is accomplished by splitting evaluation into several instructions. But the condition that all conditional branches should be both taken and not taken is still directly applicable, since there are conditional branch instructions. They do however act on flags set by previously executed comparison instructions to determine whether or not to take a branch.

When measuring coverage on the instruction level, the problem shown in the example is not as apparent. When compiling the code with GCC, the GNU C compiler, the following assembly code is generated for the if statement (registers replaced with variable names for readability):

```assembly
cmp $0, a
je .false
cmp $0, b
jne .true
call c
testl %eax, %eax
jne .true
jmp .false
```

To achieve full branch coverage each of the three branch statements (je and jne) needs to be taken and not taken. This means that input data that generates a call to c() is now required to satisfy this, since if c is not called the last jne instruction will never be evaluated.

### 2.2.3 Condition coverage

Condition coverage is a measure that is similar to branch coverage. It requires that all conditions that are used for making decisions evaluate to both true and
false. This would catch the problem in the example that branch coverage did not handle well. Unfortunately the reverse is also true; there are cases when full condition coverage does not mean full branch coverage. The problem can be seen in the following example:

```c
bool f(bool c) {
    return false;
}
if (f(a && b))
    statement;
```

If the input is varied so that a and b evaluates to both true and false full condition coverage would be achieved. But the if statement would still always evaluate to false, which does not satisfy full branch coverage. In this simple example there is of course no way to achieve full branch coverage, but a real program could possibly do more complex calculations in the called function f().

This has the same meaning when measuring coverage on the instruction level. Practical issues arise however when trying to find out which variables or registers contains conditions that will be used for later decisions. On a source code level it suces to analyze the control structures used for decisions such as if statements, but this information is not present at the instruction level.

### 2.2.4 Condition/decision coverage

This measure is common in source instrumenting tools, and is a combination of the two previously mentioned measurements. This simply means that every decision must evaluate to both true and false, and at the same time each of the values used in those decisions must have been evaluated to both true and false. There is also a version that is called modified condition/decision coverage, shortened as MC/DC. This measure requires that each value used in the decision can be shown to independently affect the outcome. This measure is required by the standard RTCA/DO178B [6] which describes requirements for software used in aviation, where software quality obviously is of critical importance.

It relates to instruction level coverage as described in the two previous sections. It can be of interest to note that the RTCA/DO178B allows MC/DC to be measured on either the source code level or at the instruction level in order to pass certification [8].

### 2.2.5 Path coverage

This measure is also known as C*. To achieve full path coverage, it is required that all unique paths through a program are exercised. For anything larger than trivial programs this is practically impossible, since the number of paths grows exponentially with the number of branches. A program that only consisted of 16 consecutive if-statements would have 65536 different paths that would need exercising. Furthermore, it can be the case that not all paths are even possible to achieve. A simple example of this problem can be seen in the following piece of code:

---

1The author has not had access to this document; it is only included as a reference to point out where this standard can be found. Its contents have been inferred from [8] and [4].
2.3 Other metrics

While metrics that are simpler than statement coverage may not be of much theoretical interest, they can be of practical use. Subsets of statement coverage such as function coverage and source file coverage can aid in figuring out what kind of tests to add to increase coverage percentage. Function coverage measures whether or not a specific function was invoked. Source file coverage is even less granular and measures whether or not any code from a certain source file was executed.

2.4 Summary of metrics and their implementation suitability

In table 2.1 the previously discussed coverage metrics are summarized together with a comment on how suitable they are for implementing in a tool that measures coverage on the instruction level. It is also noted, when applicable, how suited they are for presenting the results in terms of the original source code instead of the executed instructions.

Table 2.1: Summary of metrics and their implementation suitability

<table>
<thead>
<tr>
<th>Coverage metric</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source file</td>
<td>Suitable</td>
</tr>
<tr>
<td>Function</td>
<td>Suitable</td>
</tr>
<tr>
<td>Statement</td>
<td>Suitable</td>
</tr>
<tr>
<td>Branch (decision)</td>
<td>Best suited for showing results on instruction level but could be related to the source code in most cases</td>
</tr>
<tr>
<td>Condition</td>
<td>Mostly of interest when measuring coverage on the source code level</td>
</tr>
<tr>
<td>Path</td>
<td>Only of theoretical interest</td>
</tr>
<tr>
<td>MC/DC</td>
<td>Difficult in practice to implement, since showing that a value can affect the outcome of a decision is not trivial [8].</td>
</tr>
</tbody>
</table>

```plaintext
if (a)
  b();
c();
if (a)
  d();
```

If the call to c() does not modify a, it will not be possible to find input data that evaluates the first if statement to false and the second to true. There exist several other coverage measures that attempts to deal with these and other difficulties, but such measures will not be examined here.
2.5 Monitoring and intrusion

When doing measurements on any kind of scientific experiment, it is of interest that the act of measuring does not affect the property being measured. In [9] it is shown that instrumentation of code to measure performance can cause a lot of perturbations in the monitored program, such as increased execution time, changed memory reference patterns and register interlock stalls. Such effects are unwanted when monitoring code coverage since it could potentially cause the monitored program to take different actions. The act of altering the monitored program in any way is called intrusion, and would ideally be avoided. In [7] this is referred to as the probe effect and is discussed in terms of concurrent programs. This term refers to programs that execute several things at once, and where synchronization is of high importance. However, [8] details several other cases where non-concurrent programs can be affected by instrumentation as well. An example of this is that probes inserted into a program makes the program use more memory, and could possibly cause it to exceed a limit that it would not have if it were to be run without these probes.

Even though a non-intrusive method of measuring coverage would be preferred if there were no other factors to consider, intrusive methods of measuring coverage are used because it is usually easier to implement them and relate their results to the source code.

Insertion of probes into a program at some level to monitor its progress is one technique that is used for measuring coverage. There exists several other techniques as well to measure this, and they vary in their intrusiveness. The following sections describe these techniques in detail and the final section presents some examples of existing tools that makes use of them.

2.5.1 Source code instrumentation

A common technique is to modify the source code before compiling it by inserting special code that monitors the flow through the program. Since most of the existing coverage metrics refer to measuring coverage on the source code level, this is a natural approach. It is not completely without drawbacks though. One problem is that after modification, it could be argued that it is no longer the original program that is examined. Unlikely, but not impossible, things such as compiler bugs could cause the modified program to behave differently.

Another problem with instrumenting the source code is that there is no way to account for dynamically generated code. Since this code simply does not exist at compile time, there is no source in which instrumentation can be inserted.

There are several tools available to do this kind of coverage measuring; some examples are Clover and Bullseye Coverage which are described in section 2.5.5.

2.5.2 Object code instrumentation

Another technique to gain information on the program flow is to insert instrumentation directly into code after it has been compiled. This could either be in the object code before linking, or into the final executable. This avoids the potential problem of compiler errors, but still retains the other drawbacks that source code instrumentation has. It could possibly be adapted to be used with
dynamically generated code as well; depending on at which times the instrumentation is inserted. If the coverage measurement tool is able to detect in some way that generated code is about to be executed, it could insert instrumentation into that code as well.

2.5.3 Hardware monitoring

These tools are hooked into the actual hardware running the program, and monitor their execution non-intrusively. With this approach the program under inspection is not affected in any way. In theory, this is a perfect way of non-intrusive monitoring, but there are a number of practical problems with this approach that need to be considered. Most common processors do not have support for recording an execution trace, which means that hardware monitors will have to check for executed instructions through other means. One way is to monitor memory reads and match these to instructions, but performance enhancements in the processor such as caching and speculative execution does not guarantee that an instruction that was read from memory was actually executed. In [2] from 1990 Beizer predicts that future hardware will support measuring of instruction, statement and branch coverage natively. This has as of yet not become a reality. However there exist tools that can be used on embedded systems that do provide an external way of monitoring execution. One example of this is G-Cover, described in section 2.5.5.

2.5.4 Simulators

Monitoring coverage through a simulator is similar to hardware monitoring, but with the benefit that many of the practical problems are easier to overcome. After all, one of the reasons hardware simulators exist is to make it possible to make measurements that just are not feasible on real hardware. The prototype being developed in this project fits into this category. There does not seem to be many similar tools in existence, which may be caused by the fact that there are not many full system simulators available. But a similar area where simulation is much more common is in hardware design. This simulation is done on the gate level of circuits, and several of these hardware simulators also provide features for measuring coverage.

2.5.5 Examples of existing tools

Table 2.2 gives a list of existing coverage tools and what section they fit into, and also what kind of coverage measurement they can provide. This is by no means a complete list. It only points out some examples that use the various methods previously discussed.
2.5. MONITORING AND INTRUSION  CHAPTER 2. TEST COVERAGE

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clover</td>
<td>Source code instrumenting</td>
<td>This is a commercial tool for measuring function, statement and branch coverage on Java applications. It can be found on: <a href="http://www.cenqua.com/clover/">http://www.cenqua.com/clover/</a></td>
<td></td>
</tr>
<tr>
<td>Bullseye Coverage</td>
<td>Source code instrumenting</td>
<td>Another commercial tool that measures function and condition/decision coverage on C++. It can be found on: <a href="http://www.bullseye.com/">http://www.bullseye.com/</a></td>
<td></td>
</tr>
<tr>
<td>GCoV</td>
<td>Source code instrumenting</td>
<td>This tool is provided as a part of the GNU Compiler Collection. It is a little different from the other two mentioned tools that instrument source code, because GCoV is built into the compiler. This means that it does not actually modify the source files before compilation, but instrumentation is inserted at some point during the compilation. It can be found on: <a href="http://gcc.gnu.org">http://gcc.gnu.org</a></td>
<td></td>
</tr>
<tr>
<td>G-Cover</td>
<td>Hardware monitoring</td>
<td>A commercial tool intended for certification of code running on embedded systems according to the aviation standard DO178B. This tool supports statement and decision coverage, and also MC/DC when the program is compiled with their own compiler. It can be found on: <a href="http://www.ghs.com/products/safety_critical/gcover.html">http://www.ghs.com/products/safety_critical/gcover.html</a></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: Existing coverage tools
Chapter 3

Simics

Simics is a full system simulator. This means that it simulates a complete computer system, including the central processor and various devices such as memory controllers and network interfaces. There are a few terms that are necessary to define before discussing Simics further. First, the host is the computer that runs the Simics software. The target is the simulated machine running inside Simics.

For this project, two different versions of Simics were used. The latest development version was used on the host for running the coverage tool, and an older version was run inside the simulated machine for the purpose of executing the various tests from the Simics test suite.

3.1 General description

Simics is the flagship product of the company Virtutech AB. It is a full system simulator that attempts to strike a balance between performance and accuracy. It is accurate enough to run unmodified operating systems such as Solaris, Linux and Windows XP, and at the same time fast enough to run realistic workloads including things such as the SPEC CPU2000 benchmark and interactive desktop applications. Simics simulates processors at the instruction-level and provides models of several different processors such as UltraSparc, MIPS, Alpha and x86.

Simics is primarily controlled through a command line interface, CLI, which is similar to the front-end of a debugger such as the GNU debugger. It also has a built in Python interpreter that makes it possible to load and execute Python scripts from the prompt, and also evaluate Python statements entered directly on the console. For more detailed information, see [5].

3.2 Scripting and API

A big feature of Simics is that it is quite extensible. It allows users to write new device models, add new commands and write control and analysis routines. The Simics API is written in C, and is also exported automatically to Python when Simics is built. This API exports many functions, data types and interfaces. It
gives the user a lot of power and flexibility, an example of this is that the CLI in Simics is actually written in Python using the Simics API.

### 3.3 Contexts and virtual memory

Simics supports the concept of different contexts in the simulated processor. This can be used to map for example a specific user level process to a context. Since Simics does not have special knowledge about the operating system running inside the simulator, the actual detection of when the process of interest is active is up to the user or tool maker. Depending on the simulated machine and the operating system, this can be more or less difficult.

By using a context it is possible to use virtual memory addresses. But since Simics does not know how the operating system manages virtual memory there can sometimes be problems accessing data at such addresses. It is required that the memory of interest can be found in the active translation lookaside buffer, TLB. This is a cache mechanism in modern processors to quickly map virtual memory addresses to physical ones. For the prototype this causes some problems when trying to inspect character strings passed to certain functions, more on this in a later section.

### 3.4 Profiling

Simics has facilities for doing various kinds of profiling on the simulated workload. One useful function is the ability to record information about which parts of a memory interval executed code, and also counts the number of times the code branched. This functionality is provided by the `branch-recorder`, and it works with the Sparc and MIPS processor architectures. The information gathered only contains information about the number of times these things happen; there is no way to reconstruct the order in which they occurred. For coverage measurements such as statement and decision coverage this information is enough, but it would not be sufficient to implement path coverage for example. Simics does however provide the possibility of doing a full trace of the execution, so it would be possible to use this if more advanced coverage measures were to be considered.

### 3.5 Symbolic debugging

To provide support for symbolic debugging of code running in the simulated machine Simics provides the module `symtable`. It provides various means of examining code and variables in terms of the source code. The capabilities of loading symbolic information from binary files and allowing mapping from a memory address to the corresponding source code line were used during the development of the prototype.
3.6 Test suite

To test the product Simics, Virtutech AB uses an automated test-suite which is made up of a number of groups testing different aspects of the program. Each group usually contains a number of specific tests that are run in sequence. Since one of the goals of the prototype tool was to be able to use it with real world test applications, it was decided to use the Simics test suite as a subject.
Chapter 4

The coverage tool

4.1 The prototype

A goal of this project was to create a working prototype for a coverage measuring tool. This would help indicate whether Simics could be a suitable tool for doing this kind of analysis, and determine what kind of pitfalls would be encountered. The following list highlights the features that the developed prototype includes:

- Non-intrusive measurement.
- Identifies and isolates a Solaris user process.
- Separates collection of data from analysis.
- Aggregates data collected from several runs.
- Implements the functionality as a Simics module.
- Handles dynamically loaded code.
- Provides instruction level and source level statement coverage per text segment.

This is a list of features that were originally planned but not implemented due to time constraints and other reasons:

- User-friendly presentation and navigation of results.
- Measuring of dynamically generated code.
- Instruction level branch coverage.

The following sections detail what the original goals of the prototype were, and what happened during development that made it necessary to adjust some of the plans.
4.1 Design goals

There were several design goals that were envisioned when creating the prototype. One was to separate the collecting of information from the analysis. In this way the data collected could be used together with several different analyzers, depending on what kind of information was desired. It should also be possible to add together collected data from different runs, to see if and how additional testing adds to the total coverage figure.

The prototype was intended to be able to monitor a user process running under Solaris, and be able to detect if code was dynamically loaded during the execution. It was also intended to be able to work on unmodified binary files. It was furthermore of importance that the prototype did its measuring completely non-intrusive, since this was intended to be one of the reasons for doing this kind of coverage measuring in the first place.

4.1.2 Environment

For the purpose of creating this prototype, a specific environment was chosen. The host machine was running Simics for 32bit x86 on RedHat Linux. The simulated target machine however was a Sparc running the Solaris operating system. Since use of the branch-recorder functionality in Simics was desired, it narrowed down the choices to either Sparc or MIPS, of which Sparc was chosen. There was no particular preference here though, either would probably have worked fine.

The prototype was designed to be used as a Simics module. This makes it easy to use and interact with from inside the Simics command line interpreter. To write modules for Simics there are basically two choices of source language available, Python and C. Modules written in Python are slower since it is not a compiled language, but on the other hand it provides better facilities for managing data structures. Since the prototype would not be running any time-critical code but instead relying on Simics functionality and only collect data at certain times, Python was chosen as the language.

4.1.3 Overview on how the prototype works

The prototype adds a few commands to the Simics command line that provides the functionality to create monitor objects and analyzer objects. When creating a monitor object, the name and path of the program of interest is provided, and when the simulation is running the monitor watches for the execution of the specified program and goes into action when that happens. Figure 4.1 shows how the monitor fits in the general scheme of things. It is run inside Simics, and monitors non-intrusively both the user application under inspection and the libraries that it uses. While it sees all libraries that are loaded, it ignores all libraries that reside in the folder “/usr/lib” which are the ones provided by the operating system. The reason behind this is that these files are not part of the program under testing. It would however be simple to adjust this mechanism if the user determines that coverage of the system libraries is interesting as well.

After a run is complete the collected data can be saved to a file for later analyzing or passed to an analyzer object directly. The prototype provides two different analyzers, one that shows total percentage of executed statements from
4.1.4 System call interception details

There are five different system calls intercepted by the coverage monitor. These are listed in table 4.1 together with information on what they do and why they need to be monitored.

4.1.5 About binary files

The prototype is able to measure coverage on binary files in the ELF [12] format. This is a common format on Unix operating systems. An ELF file is divided into segments of different types which contains various types of information. For the coverage monitor the text segment is of most interest. This is where almost all the executable code in the program resides, in particular all the code that corresponds directly to lines in the source code. There are however several more segments that can contain code, two of them that are also considered by the coverage monitor is the init and fini segments. These are executed first and last respectively during the execution of the program. The coverage monitor makes use of the fini segment to determine when the program has stopped executing.

An ELF file usually contains a lot of other segments as well. The segments that contain symbolic information are of interest to the coverage analyzers, but this information is not handled directly by the prototype. For this it relies on functionality in the Simics symtable module.

4.1.6 Monitor implementation details

When the monitor object has been created it in turn creates a branch recorder object but does not connect it to any context. It then registers callbacks to allow it to detect when execution of the program under inspection begins. This is done by monitoring system calls until a call to the function `execve` is made. If
4.1. THE PROTOTYPE

CHAPTER 4. THE COVERAGE TOOL

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>open (0x05)</td>
<td>This call is used to open a file from disk. It returns a filehandle which is passed to other system functions. This handle is saved in order to identify what files are referenced by execve and mmap.</td>
</tr>
<tr>
<td>close (0x06)</td>
<td>Used to close an open file handle. Removes this handle from the internal list.</td>
</tr>
<tr>
<td>execve (0x3b)</td>
<td>Loads the specified file into memory and executes it. Used to catch when the program of interest actually starts.</td>
</tr>
<tr>
<td>mmap (0x73)</td>
<td>Maps a file into memory. This allows detection of dynamically loaded modules. The return value is the address they are mapped to, which is used to monitor that region.</td>
</tr>
<tr>
<td>munmap (0x75)</td>
<td>Unmaps a file or parts of it from memory. If this matches a monitored region information is extracted from the branch recorder.</td>
</tr>
</tbody>
</table>

Table 4.1: Intercepted system calls

the name matches the monitored file, information about where the code segment of the program will be located in memory is retrieved from the binary file. The monitor then creates a execution breakpoint at that memory address to detect when the program has started executing. This means that the initialization code segment will not be considered for coverage monitoring. The reasoning behind this is that this segment is generated by the compiler and has no corresponding source code.

When the breakpoint is triggered the current context register is saved and a Simics context is created to which the branch recorder is attached. The monitor then monitors for calls to mmap and munmap which requires it to take action. When mmap is called the monitor registers a new interval of memory to watch for code execution. When munmap is called the monitor saves the branch recorder information for all currently watched memory regions, and clears the branch recorder. This is needed to avoid conflicts if a later loaded module should occupy regions that were previously in use. Finally when the monitor sees that the program is executing its finalizing code section it determines that the run is over.

4.1.7 Analyzer implementation details

The analyzers that measure total percentage coverage iterates over the information collected on each binary file that was present in a monitored run. It then adds together the total amount of instructions and how many of those were invoked.

The source code statement analyzers also iterates over each binary file. For each memory address it asks simtable about the corresponding source code line. A data structure is then built that adds a count to each source line and memory address pair. After analyzing this allows the output to show how many of each lines corresponding instructions were executed. For most lines this is either
4.2 Problems

To create the tool, there were a number of obstacles to overcome. Since the prototype was intended to work on unmodified binary files, there were both theoretical and practical issues. In the following subsections are descriptions of problems that were encountered and how they were solved.

4.2.1 Theoretical problems

The major theoretical issue is the fact that while this tool gathers information on the assembly code level, it is more desirable to view the information in terms of the source code. But the possibility of mapping the instructions back to the corresponding source code cannot be guaranteed in the general case. The main cause of this problem is the use of code optimization.

Since there are lot to be gained in terms of program execution speed when optimizing the code that a compiler generates, programs are usually compiled with optimizations enabled. This unfortunately makes it harder to present statement coverage, since assembly instructions and source lines can have a many-to-many relationship. One example of this is found in the GCover\(^1\) manual, and looks like this:

```c
if (a != b)
    c = 1;
else
    c = 0;
```

This can be compiled to one instruction on some machines. This means that coverage would be reported the same way, no matter whether the result is 0 or 1.

A similar problem that was encountered during the first stages of development was that the compiler optimizes several calls to a function as just one call, but varies the parameter. An example is the following code:

```c
if (a < b) {
    printf("The first number was smaller\n");
} else if (b < a) {
    printf("The second number was smaller\n");
} else {
    printf("The numbers were equal\n");
}
```

\(^1\)A coverage tool described in section 2.5.5.
When compiled the resulting assembly code only contains one call to printf, which the debug information attributes to the first line. So even when passing values such that b is less than a, stepping through the program will look like it still executes the first printf statement.

There is no general way to solve this problem. To improve on the situation, there are a number of measures that can be taken.

One way is to compile the program without optimization. This makes the relationship between the compiled code and the source a lot simpler. This is also why this is often done when debugging programs. A drawback with this approach is that it is now a modified program that is inspected instead of the original program.

Another way to make the problem less apparent is to present the information with less resolution. For example, this can be done by only showing percentage values for whole functions, instead of every source line. The reasoning behind this is that optimization is primarily performed on the statement level. The compiler does however have access to optimization techniques which involves whole functions such as inlining of code. This could make function coverage inaccurate on optimized code as well, because there once again is a one-to-many relationship between the source code and the instructions generated by the compiler just like the previous example.

4.2.2 Monitor implementation problems

The prototype was intended to run completely non-intrusively, in an environment where the simulated machine was running Solaris 9. In order to obtain the information needed to present coverage, there were a number of practical problems to solve.

4.2.2.1 Context detection

Since Simics is a full system simulator, it sees all instructions that are executed on the target machine. This includes code from the operating system and other processes. But when measuring coverage, it is only the program under inspection that is interesting to monitor. Therefore, it was necessary to be able to distinguish between different processes.

On the Sparc v9 there is a context register in the MMU [13]. This register is used by the operating system to optimize context switching, since it allows the operating system to switch between register windows with very low overhead. Therefore, monitoring this register for changes will make it possible to detect when the operating system switches between processes. There is no guarantee that the operating system keeps the same context number for a process during the entire execution, but during experiments this has always been true.

To find the actual context register number that the process of interest was assigned some method for detecting this was needed. One intrusive method was to run a helper program inside the simulated operating system that would execute the program of interest and report the context number to the simulator module. This has some drawbacks however, which are discussed further in the section on dynamically loaded code.

The second method that was considered was to monitor operating system calls for the execution call. This is the method that was chosen for the prototype,
and is further discussed in the section on system call interception.

4.2.2.2 Finding the total set

For the purpose of showing a percentage value on how much of a binary file was executed, it is not only necessary to find out which instructions that were executed, but also to find out the total number of possible instructions. To find this information, it is required to extract it from the binary file. The tool supports the ELF format for executable files as discussed in section 4.1.5, and the information it needs are the size and location of the text and init segments.

There are several possible ways to retrieve this information, for example there exists libraries to help with ELF file management. For the prototype the command "objdump -h <file>" was executed, and the output parsed for the wanted information. This is a simpler method than managing ELF files manually, but there are some drawbacks. One is that the prototype is limited to the types of files that objdump can understand. This problem was seen when trying to get information about 64 bit binaries compiled under Solaris, which it did not support.

When dealing with programs that load code dynamically from other files, these files also needs to be examined for their code content. When this happens during an execution the coverage tool detects this, which is described in the next section. However, it is possible that the program does not load all the files that should be considered for the coverage calculations. This means that it is not possible for the coverage tool to automatically determine which files should be examined during analysis. A possible solution is to add these files manually so that they are included as completely uncovered in the percentage calculations.

When showing coverage information in terms of the source code there is a similar problem as well. To do the mapping between instructions and source code, the prototype requires that symbolic information is present in some form that the Simics module symtable can read. When going through this symbolic information, all source files that were used during the compilation of the program will be found. The problem of identifying source code files that are present in the source code tree but does not actually generate any code in the resulting binary is not dealt with by this tool.

4.2.2.3 Dynamically loaded code

Since one of the goals of this coverage prototype was to show that it could work on real programs that use dynamic loading of code, this was a problem that had to be solved. Some types of dynamic loading can be analyzed by looking at the executable file and at the files it references. It is also quite possible for a program to not know until runtime what other files it will load. The only certain way to find this information is to run the program and see what actually happens.

To see what happens, a method of knowing when a program loads code dynamically must be devised. The first method that was implemented involved running a helper program as a separate process inside the simulation. This helper program would periodically invoke the command "pmap <pid>" which shows a map of the specified process that contains information about what modules have been loaded and their addresses. This information would then be
4.2. PROBLEMS

CHAPTER 4. THE COVERAGE TOOL

communicated through special means to a module in the simulator. However it can be argued that this method violates the non-intrusiveness that is desired for this tool. While the helper program is run in a separate process from the program under inspection and thus does not modify it in any way, it can still affect timing that would be different had the inspected program been running by itself.

To make sure that the coverage monitor was completely non-intrusive, it was of interest to find a different method of detecting dynamically loaded code. Later during the development such a method was devised, which is detailed in the next section.

4.2.2.4 System call interception

The second method of detecting dynamic loading of code was to monitor operating system calls. This was possible to do from the simulator module, which meant that nothing extra apart from the program under inspection had to be run inside the simulated operating system.

Programs use various operating system calls to request certain type of services from the operating system. Such services include executing other programs, and mapping files into memory. These calls are rarely invoked directly from a user program but are instead provided through higher level calls in standard libraries. The actual invocation of a system call in the Solaris operating system on a Sparc processor is made by putting required values into registers and then triggering a software exception. Simics provides a callback that is used for detecting these exceptions which makes it easy to intercept them.

Since the actual system calls are embedded into operating system provided libraries for which no source code is provided, there is a practical problem of finding out what register values correspond to what call. Some of this information could be found in a Simics module that emulates a small part of the Solaris operating system, and the rest was found simply by writing test programs and monitoring what happened.

A problem with this approach is that values passed to the system calls are using virtual memory addresses, but the simulator only has access to the physical memory. Simics provides a method to map a virtual address to a physical address, but if the required information is not present in the active TLB as discussed in section 3.3, this does not work. Very detailed knowledge of the paging mechanisms in the operating system would be required in order to resolve this. A possible work-around that could improve the chances of being able to perform this mapping would be to retry the mapping at a later time, but the prototype does not currently support this. During experiments it was very rare to run into this problem. The problem sometimes occurred when checking the arguments passed to execute in order to determine if the interesting program was about to be executed. When this happened the test had to be re-run.

4.2.3 Analyzer implementation problems

The prototype tool provides two different analyzers. The analyzer that shows percentage of total code execution is fairly simple since all the required information such as how much code a binary contains and what parts have been
executed is exactly the information that the monitor provides. To show statement coverage on the source code level more detailed analysis is required.

4.2.3.1 Finding symbolic information
To map executed addresses to source code lines, symbolic information must be present in some form. A common format for this is the stabs format [10]. During development the idea of reading this information directly was considered, but since the Simics module symptom that provides symbolic debugging has the required functionality it was used instead. There would probably be performance improvements if the debug information was handled separately, since the coverage tool can make certain assumptions such as that it will be reading the information sequentially from start to finish, which the symptom module cannot. But for a prototype optimization considerations have low priority.

4.2.3.2 Presentation
Presenting source code coverage for a real program can be tricky, since there is so much information to present. The current analyzer simply prints out the information for one source file at a time. For this type of information to be useful some kind of browser should be implemented. Suggestions on things that could be done in this area are discussed in the future work chapter. The source code statement coverage analyzer does however also provide the option of just outputting summaries like its instruction level counterpart.

4.3 Results
This section describes some actual results when using the coverage tool on the Simics test suite. The version of Simics that was used for testing was version 1.6.11. Since the full test suite is rather large only a subset was analysed. This version of Simics provided several different binaries depending on which processor it simulated. The subset of the test suite that was chosen was the tests provided for the sparc-u3-32 binary. This binary file is used for simulating the Sparc U3 processor on a 32 bit host.

4.3.1 Overview of tests in the test suite
The test suite contains a number of groups of tests that tests various aspects of the Simics functionality. These groups contain one or more specific tests. Table 4.2 contains information about what tests are relevant for the chosen binary, and comments about whether or not they were chosen to be run on the simulator as well. The comments about test failures indicate whether or not the test failed on a real machine, unless otherwise noted. There were a few tests that did not run on the simulated machine despite working on a real machine. Most of these failures were due to timeouts, and this happens because the simulated machine runs at a low clock frequency. Some others required large files to be present on the simulated machine.
### 4.3. RESULTS

#### CHAPTER 4. THE COVERAGE TOOL

<table>
<thead>
<tr>
<th>Test name</th>
<th>Used</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>t02_generic_bugs</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>t03_sparc_bugs</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>t04_sparc_memhier</td>
<td>X</td>
<td>One subtest failed.</td>
</tr>
<tr>
<td>t09_sparc_n3</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>t15_checkpoints</td>
<td></td>
<td>Requires a lot of disk space in the simulated machine, so it was not run.</td>
</tr>
<tr>
<td>t16_trace</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>t24_symtable</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>t26_distributed</td>
<td></td>
<td>Failed.</td>
</tr>
<tr>
<td>t30_fronedt</td>
<td>X</td>
<td>One subtest failed in the simulator.</td>
</tr>
<tr>
<td>t34_ext_mem_bus</td>
<td></td>
<td>Failed.</td>
</tr>
<tr>
<td>t37_proxy</td>
<td></td>
<td>Failed in the simulator.</td>
</tr>
<tr>
<td>t40_remote_fronedt</td>
<td></td>
<td>Failed.</td>
</tr>
<tr>
<td>t42_userinstall</td>
<td></td>
<td>Failed.</td>
</tr>
<tr>
<td>t43_ext_pci_dev</td>
<td></td>
<td>Failed.</td>
</tr>
<tr>
<td>t47_api_doc</td>
<td></td>
<td>Failed.</td>
</tr>
<tr>
<td>t48_caches</td>
<td>X</td>
<td>Some subtests failed in the simulator.</td>
</tr>
<tr>
<td>t49_v9_multipro</td>
<td>X</td>
<td>Not all subtests were run due to time constraints.</td>
</tr>
<tr>
<td>t51_license</td>
<td></td>
<td>Some subtests failed.</td>
</tr>
<tr>
<td>t53_sparc_vis</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>t54_sparc_fpu</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>t55_flush_stc</td>
<td>X</td>
<td>One subtest failed.</td>
</tr>
<tr>
<td>t56_checkpoint</td>
<td></td>
<td>Failed.</td>
</tr>
<tr>
<td>t57_sfp</td>
<td></td>
<td>Not run in the simulator due to time constraints.</td>
</tr>
</tbody>
</table>

Table 4.2: Tests in the test suite
4.3. RESULTS

CHAPTER 4. THE COVERAGE TOOL

4.3.2 Coverage results

The results from the binary coverage analyzer and the summaries from the source code coverage analyzer can be presented in diagrams. Figure 4.2 shows the percentage values that the tests covered over a few selected binary files from Simics. Since not all tests exercise all binary files, some of the ones that are most commonly used during the test execution were chosen. The final column show the total percentage that was covered when the coverage from all the tests were aggregated.

Another interesting way of examining the data is by showing what tests increase coverage the most, when examined in sequence. Figure 4.3 shows this. Each line represents one of the binary files, and they go from no coverage (0%) to the final values that are found when all tests are aggregated. This chart depends on the order in which the test results are aggregated, since if two tests exercise exactly the same parts of the program, only the first one will be shown as increasing total coverage.
A third way of examining the data is by comparing the coverage figures with regard to instruction level coverage and source code coverage. Figure 4.4 shows how they compare. The data is aggregated from all the executed tests.

The category *instructions* is the percentage covered when measuring coverage directly on the binary file, as seen in the total sum in figure 4.2. The category *source lines* means lines whose corresponding instructions were all executed, and *partially executed* means lines that had some but not all of its corresponding instructions executed. Finally, *instructions with corresponding source* is similar to the first category, but it does not include instructions that are shown to have no corresponding source code line.
Chapter 5

Future work

There are several possible ways to improve the developed coverage tool. One improvement would be to find solutions to the problems discussed in section 4.2 that were not handled by this prototype. There are also several aspects of the prototype that could be developed further. Sections 5.1 and 5.2 discuss some possibilities and what would be required to implement them. Section 5.3 discuss the various aspects that would have to be considered when adapting the tool to run on a different configuration such as another operating system.

5.1 Monitor

The execution monitor currently only saves information about executed instructions. To make more advanced forms of coverage measurements possible, it would be required to save branch information as well. The branch recorder module in Simics provides information about branches that has been taken, so this information would have to be combined with information about branches that has not been taken as well as those that were not executed at all. For the Sparc processor, it would be relatively easy to just look at all the instructions and determine if they are branch instructions or not. The problem becomes a little more complex on an architecture such as the x86 where it is not as easy to know where the instructions are located, but could probably be solved by using knowledge gained from the symbolic information. It could perhaps also be solved by extending the branch recorder to take note of untaken branches, but this would still miss branches that were never evaluated at all.

It would also be good if it was possible to tell the monitor about files that are part of the inspected program but are not detected during its execution. This would be used to show that these files have 0% coverage. It is possible to add an executable file for consideration manually in the current prototype, but some kind of automation such as specifying folders and files with wildcards could be useful here.

5.2 Analyzer

The analyzer has two main areas that could be improved upon, presentation of the coverage information and providing more types of coverage.
5.3. PORTING

CHAPTER 5. FUTURE WORK

The output from the analyzer could perhaps be output in a format that is suited for better presentation. For example when showing source code coverage it could output html pages that can be browsed. Another variation on this would be to have an external graphical browser that can read the unformatted output from the analyzer and allow a good overview and browsing. The coverage tool Clover uses this to present statement coverage for source code as seen in figure 5.1.

Another type of coverage that would be the logical next step to implement would be branch coverage. If the monitor would provide the needed information, the analyzer could use similar mechanisms as the current statement coverage code.

5.3 Porting

The prototype has been designed for monitoring a user level process running under Solaris on a Sparc processor. There are however a lot of other combinations that are interesting such as other processors with different instruction sets, and even other types of processes such as measuring coverage on an operating system kernel. This section discusses some of these possible variations and what problems could arise when handling these. For some problems various solutions are also discussed.

5.3.1 Different instruction sets

The Sparc processor uses instructions that are exactly four bytes long. This has several positive implications for measuring binary coverage. First, it is trivial
to figure out how many instructions are in an address range, it just requires dividing the length by four. Other instruction sets such as the x86 use variable length instructions. To figure out the total number of instructions here would require relying on symbolic information. It is possible that it would be needed on architectures with fixed length instructions as well if the compiler for example aligns functions to even 16 bytes or similar things. This would mean that there can be empty space between functions that should not be counted in the total.

5.3.2 Different operating systems

There are several parts of the monitor that would have to be adapted to work on a different operating system. For example, the mechanism that monitors system calls is most likely Solaris specific. Detection of switching between processes may also have to be revised, even for another operating system that runs on the Sparc processor. For a different architecture it is almost a certainty. For example, the x86 does not have a context register that can be read to easily determine when a certain process is running.

5.3.3 Different source code language

The tool has only been tested on code compiled from C, but it should work without modification as long as the compiler of choice can output symbolic information that can be read by the symtable module. If that is not possible support would have to be added in some form, possibly by adding it to the symtable module or by providing it in the coverage tool directly.

5.3.4 Non user level processes

There are more things than user level processes that could be of interest to measure coverage on. Two examples are operating system kernels and embedded software. To monitor a kernel the process detection code would have to be changed to only monitor when the processor is executing code outside of the user level processes. A problem with kernels is that they execute very complex code, possibly even self-modifying code.

Self-modifying code presents a problem because it cannot be mapped back to the source code since the compiler did not create it. There is also the problem that instructions that have been marked as covered can be changed which means that they should probably not be counted as covered anymore.

Embedded software typically does not have the concept of processes since these types of programs are usually run without an operating system. This means that they are easier to monitor since the monitor can simply stay active all the time. However, if the embedded software supports loading code at runtime such as plug-ins it could be hard to detect, since there would be no standard system calls to monitor.
Chapter 6

Conclusions

The goals of the project were to evaluate different coverage metrics and also determine their suitability for implementation on the instruction level with the help of a simulator. Furthermore, it was of interest to implement some of these metrics in a tool that could isolate a user level process in Simics and handle complex applications that loaded code dynamically. Finally, it was interesting to see what results it would give on the Simics test suite. The following sections discuss how well these goals were fulfilled, and the conclusions that could be drawn from doing this.

6.1 Fulfillment of goals

Most of the goals set up at the beginning of the project were fulfilled. Statement coverage on instruction and source code level was implemented fairly easily with the help of the branch recorder and the symtable module. Isolation of a specific process and detection of dynamic loading of code was also easy to implement once a suitable method had been devised. Finding a way of doing it non-intrusively was difficult at first, but such a method was eventually found.

6.2 Instruction level versus source code

The coverage tool developed measures coverage on the instruction level. Since most of the theory on coverage is written in terms of the source code, the impact of doing it on the instruction level instead had to be determined. Many of the simpler coverage metrics translate well to the instruction level, but some of the more advanced are difficult to determine how they would be used. However, since the aviation standard DO178B allows coverage to be measured on the instruction level in order to pass certification, it is probably safe to say that this method of measuring coverage is useful as well.

6.3 Measuring coverage using Simics

Simics offered several useful facilities that aided in the development of the coverage tool. It provided a good mechanism for determining which parts of a
6.4. Test suite results

The numbers gathered when measuring coverage on the Simics test suite are somewhat low. While some tests that are used in practice were not included in the measuring, the results are probably fairly representative of what the final result would be since all but a few of the tests were used. The statement coverage on the instruction level for the main Simics binary was around 24%. This may seem low, but since the binary includes the full processor model which is generated by a special tool, it contains quite a lot of code that will only execute in special cases. Furthermore the test suite has not been written with maximized coverage as a goal, it serves mostly to test for bugs that were fixed to make sure they do not reappear in the future.
Bibliography


Appendix A

Chart data

This appendix contains the numbers used to draw the charts in section 4.3.2.

A.1 Table descriptions

Table A.1 contains the numbers of instructions and source lines for each of the examined binary files. Table A.2 contains the number of lines that were executed for each binary file. The binary files are numbered from 1 to 5 as seen in table A.1. The column headers that just contain a number refer to the corresponding binary. Table A.3 contains the number of instructions executed by each test per binary file. Table A.4 contains the number of instructions executed when the results from the tests are aggregated together in order.

A.2 Information usage

Figure 4.2 shows the number of instructions executed by each test from table A.3, divided with the total amount of instructions found in table A.1. Figure 4.3 divides the aggregated number of executed instructions from table A.4 with the same total amount. Figure 4.4 divides the information in table A.2 with the matching number from table A.1.

<table>
<thead>
<tr>
<th>Name</th>
<th>Instructions</th>
<th>Instr. with source</th>
<th>Source lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>simics-sparc-u3 (1)</td>
<td>1104339</td>
<td>1080224</td>
<td>54266</td>
</tr>
<tr>
<td>libimage-common (2)</td>
<td>16356</td>
<td>15967</td>
<td>960</td>
</tr>
<tr>
<td>libpython-frontend (3)</td>
<td>276809</td>
<td>274097</td>
<td>14347</td>
</tr>
<tr>
<td>libcheetah-rmmu (4)</td>
<td>25061</td>
<td>24887</td>
<td>1263</td>
</tr>
<tr>
<td>libsparc-irq-bus (5)</td>
<td>1677</td>
<td>1661</td>
<td>179</td>
</tr>
</tbody>
</table>

Table A.1: Number of instructions and source lines per binary
### A.2. INFORMATION USAGE

#### APPENDIX A. CHART DATA

<table>
<thead>
<tr>
<th>Binary</th>
<th>Full lines</th>
<th>Partial lines</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11192</td>
<td>15574</td>
<td>258878</td>
</tr>
<tr>
<td>2</td>
<td>306</td>
<td>441</td>
<td>5409</td>
</tr>
<tr>
<td>3</td>
<td>2280</td>
<td>2872</td>
<td>71053</td>
</tr>
<tr>
<td>4</td>
<td>523</td>
<td>655</td>
<td>12020</td>
</tr>
<tr>
<td>5</td>
<td>29</td>
<td>36</td>
<td>331</td>
</tr>
</tbody>
</table>

**Table A.2:** Number of partially and fully executed lines, and executed instructions with corresponding source code

<table>
<thead>
<tr>
<th>Test name</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>T02</td>
<td>75136</td>
<td>11714</td>
<td>46139</td>
<td>2721</td>
<td>268</td>
</tr>
<tr>
<td>T03</td>
<td>134192</td>
<td>2342</td>
<td>50977</td>
<td>4072</td>
<td>268</td>
</tr>
<tr>
<td>T04</td>
<td>121835</td>
<td>1795</td>
<td>54205</td>
<td>3247</td>
<td>291</td>
</tr>
<tr>
<td>T09</td>
<td>97073</td>
<td>2214</td>
<td>50479</td>
<td>3030</td>
<td>268</td>
</tr>
<tr>
<td>T16</td>
<td>161057</td>
<td>4273</td>
<td>53307</td>
<td>6105</td>
<td>308</td>
</tr>
<tr>
<td>T24</td>
<td>110899</td>
<td>1795</td>
<td>54146</td>
<td>2812</td>
<td>268</td>
</tr>
<tr>
<td>T30</td>
<td>109412</td>
<td>4848</td>
<td>60621</td>
<td>5100</td>
<td>268</td>
</tr>
<tr>
<td>T48</td>
<td>200338</td>
<td>3713</td>
<td>54425</td>
<td>8066</td>
<td>308</td>
</tr>
<tr>
<td>T49</td>
<td>96212</td>
<td>1795</td>
<td>48856</td>
<td>3000</td>
<td>291</td>
</tr>
<tr>
<td>T53</td>
<td>93968</td>
<td>1734</td>
<td>50355</td>
<td>2721</td>
<td>268</td>
</tr>
<tr>
<td>T54</td>
<td>92213</td>
<td>1734</td>
<td>50807</td>
<td>2812</td>
<td>268</td>
</tr>
<tr>
<td>T55</td>
<td>190024</td>
<td>4273</td>
<td>56746</td>
<td>8371</td>
<td>308</td>
</tr>
<tr>
<td>Total</td>
<td>261017</td>
<td>5414</td>
<td>71053</td>
<td>12020</td>
<td>331</td>
</tr>
</tbody>
</table>

**Table A.3:** Number of instructions executed by each test per binary

<table>
<thead>
<tr>
<th>Test name</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>T02</td>
<td>75136</td>
<td>11714</td>
<td>46139</td>
<td>2721</td>
<td>268</td>
</tr>
<tr>
<td>T03</td>
<td>134238</td>
<td>2342</td>
<td>51381</td>
<td>4072</td>
<td>268</td>
</tr>
<tr>
<td>T04</td>
<td>154454</td>
<td>2342</td>
<td>55223</td>
<td>4080</td>
<td>291</td>
</tr>
<tr>
<td>T09</td>
<td>155663</td>
<td>2342</td>
<td>55682</td>
<td>4680</td>
<td>291</td>
</tr>
<tr>
<td>T16</td>
<td>202902</td>
<td>4273</td>
<td>58559</td>
<td>7794</td>
<td>331</td>
</tr>
<tr>
<td>T24</td>
<td>210335</td>
<td>4275</td>
<td>61144</td>
<td>7794</td>
<td>331</td>
</tr>
<tr>
<td>T30</td>
<td>220697</td>
<td>5414</td>
<td>69846</td>
<td>9923</td>
<td>331</td>
</tr>
<tr>
<td>T48</td>
<td>254278</td>
<td>5414</td>
<td>70030</td>
<td>12020</td>
<td>331</td>
</tr>
<tr>
<td>T49</td>
<td>254969</td>
<td>5414</td>
<td>70039</td>
<td>12020</td>
<td>331</td>
</tr>
<tr>
<td>T53</td>
<td>256069</td>
<td>5414</td>
<td>70054</td>
<td>12020</td>
<td>331</td>
</tr>
<tr>
<td>T54</td>
<td>259943</td>
<td>5414</td>
<td>70054</td>
<td>12020</td>
<td>331</td>
</tr>
<tr>
<td>T55</td>
<td>261017</td>
<td>5414</td>
<td>71053</td>
<td>12020</td>
<td>331</td>
</tr>
</tbody>
</table>

**Table A.4:** Number of instructions executed by tests aggregated in order per binary