Stony Brook University

OFFICIAL COPY

The official electronic file of this thesis or dissertation is maintained by the University Libraries on behalf of The Graduate School at Stony Brook University.

© All Rights Reserved by Author.

Compiler-Assisted Software Model Checking and Monitoring

A Dissertation Presented

by

Xiaowan Huang

to

The Graduate School

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

December 2010

Stony Brook University
The Graduate School

Xiaowan Huang

We, the dissertation committee for the above candidate for the
Doctor of Philosophy degree, hereby recommend
acceptance of this dissertation.

Dr. Scott A. Smolka — Dissertation Adviser
Professor, Department of Computer Science

Dr. Radu Grosu — Dissertation Adviser
Associate Professor, Department of Computer Science

Dr. Scott D. Stoller — Chairperson of Defense
Professor, Department of Computer Science

Dr. Klaus Havelund
Senior Research Scientist, Jet Propulsion Laboratory, NASA

This dissertation is accepted by the Graduate School.

Lawrence Martin
Dean of the Graduate School

Abstract of the Dissertation

Compiler-Assisted Software Model Checking and Monitoring
by
Xiaowan Huang

Doctor of Philosophy
in
Computer Science

Stony Brook University
2010

In this dissertation we present a compiler-assisted execution-based software model checking method targeting all languages that are acceptable by the compiler. We treat the intermediate representation of the program under compilation as a language and interpret it using a customized virtual machine. Our model checkers are based on this intermediate representation level virtual machine and have full access to its states. We implemented two model checkers: a stateless Monte Carlo model checker GMC^2 and a bounded concrete-symbolic model checker using the *dynamic path reduction* algorithm for reachability problems of linear C programs.

We also introduce the new technique of *Software Monitoring with Controllable Overhead* (SMCO). SMCO is formally grounded in control theory, in particular, the supervisory control of discrete event systems. Overhead is controlled by dynamically disabling event interrupts, but such interrupts are disabled for as short a time as possible so that the total number of events monitored, under the constraint of a user-supplied target overhead, is maximized.

We have implemented SMCO using a technique we call *Compiler-Assisted Instrumentation* (CAI). Benchmark shows that SMCO successfully controls overhead across a wide range of target-overhead levels. Moreover, its accuracy monotonically increases with the target overhead, and it can be configured to distribute monitoring overhead fairly across multiple instrumentation points.

Contents

List of Figures vii

List of Tables viii

1 Introduction 1

2 Compiler-Assisted Instrumentation 4
- 2.1 GCC Intermediate Representation . . . 5
- 2.2 Instrumenting GIMPLE . . . 7
  - 2.2.1 GCC Plug-in Architecture . . . 7
  - 2.2.2 GCC Internals . . . 8
  - 2.2.3 Pass Managers . . . 9
- 2.3 GLua: Scripted Program Instrumentation . . . 10
  - 2.3.1 GLua Usage . . . 10
  - 2.3.2 The GLua API . . . 11
  - 2.3.3 Related Work . . . 17
- 2.4 GLua Program Instrumentation Tools . . . 17
  - 2.4.1 Control-Flow Graph Visualizer . . . 17
  - 2.4.2 Tree Tracker . . . 19
  - 2.4.3 Function Duplicator . . . 19
  - 2.4.4 Basic Block Profiler . . . 20
  - 2.4.5 Symbolic Executor . . . 21

3 Compiler-Assisted Software Model Checking 23
- 3.1 Overview of Software Model Checking . . . 23
  - 3.1.1 Abstract Software Model Checking . . . 23
  - 3.1.2 Concrete Enumerative Software Model Checking . . . 24
  - 3.1.3 Symbolic Software Model Checking . . . 25
- 3.2 The GIMPLE Virtual Machine . . . 26
  - 3.2.1 GVM Implementation . . . 26
  - 3.2.2 The GVM API . . . 28

4 GIMPLE-based Monte Carlo Model Checker 31
- 4.1 Monte Carlo Model Checking . . . 31
  - 4.1.1 Büchi Automata . . . 31

4.1.2 Random Lassos and Hypothesis Testing . . . 32
4.1.3 The Monte Carlo Model Checking Algorithm . . . 33
4.2 Monte Carlo Software Model Checking . . . 34
4.2.1 The Main Routine . . . 35
4.2.2 The `rLasso` Random-Lasso Routine . . . 35
4.2.2.1 Hash Table . . . 35
4.2.3 Routines `rInit` and `rNext` . . . 36
4.2.3.1 Program State . . . 36
4.2.3.2 Routine `rInit` . . . 36
4.2.3.3 Routine `rNext` . . . 37
4.2.4 Routine `interpret` . . . 37
4.3 Experimental Results . . . 39

5 Software Model Checking with Dynamic Path Reduction 42
5.1 Nondeterministic Conditional and SSA Form . . . 42
5.2 DPR-Based Model Checking Algorithm . . . 43
5.2.1 Global Search Algorithm . . . 43
5.2.2 Weakest Precondition Computation . . . 44
5.2.3 Learning From Infeasible Sub-paths . . . 45
5.2.4 Pruning Unexplored Paths . . . 46
5.2.5 Path Reduction Algorithm . . . 47
5.3 Implicit Oracle Enumeration using SAT . . . 49
5.4 Experimental Evaluation . . . 51

6 Software Monitoring with Controllable Overhead 54
6.1 Target Specification . . . 57
6.2 Plant Models . . . 57
6.2.1 Hardware Plant . . . 57
6.2.2 Software Plant . . . 59
6.3 Controllers . . . 60
6.3.1 Global Controller . . . 60
6.3.2 Cascade Controller . . . 61
6.3.2.1 Secondary Controllers. . . . 62
6.3.2.2 Primary Controller. . . . 63
6.4 Integer Range Analysis . . . 64
6.5 Clock Thread . . . 66
6.6 Controller Design . . . 66

7 SMCO Experimental Evaluation 68
7.1 Testbed . . . 68
7.2 Workloads . . . 69
7.3 Range Checker . . . 70
7.3.1 Global Controller . . . 70
7.3.2 Cascade Controller . . . 71
7.3.3 Controller Comparison . . . 72

7.3.4 Memory Overhead 74
7.4 Controller Optimization 75
7.4.1 Clock Frequency 75
7.4.2 Integrative Gain 77
7.4.3 Adjustment Interval 79
7.5 Conclusion 83

8 Conclusion and Future Work **84**
8.1 Compiler-Assisted Techniques Conclusion 84
8.2 Future Work 85

Bibliography **86**

List of Figures

2.1 The GCC architecture . . . 5
2.2 Sample C Program and Corresponding GIMPLE representation . . . 6
2.3 Architecture of *GLua* . . . 10
2.4 A GLuascript to count functions and prints their referenced variables . . . 12
2.5 A GLua script to traverse basic blocks and gimple statements . . . 15
2.6 glua-help: a GLua script to list GLua API . . . 16
2.7 Sample C program . . . 18
2.8 Control-flow graph of function *simple ( )* . . . 18
2.9 Tree Tracker . . . 19
2.10 Sample Lua code doing function duplication . . . 20
2.11 duplicated function *simple ( )* . . . 20
2.12 *basic block profiler* script . . . 21

3.1 The GVM Plugin . . . 27
3.2 Architecture of GVM . . . 28
3.3 A GVM script to print static variable assignments ten times . . . 30

4.1 Example lasso probability space. . . . 33

5.1 A sample C program (left), its SSA form (middle), and SSA graph representation (right). . . . 43
5.2 A C program in SSA form (left), its graphical representation with a highlighted execution path (middle), and the remaining paths after learning from the highlighted path (right). . . . 47
5.3 An example control flow graph. . . . 49

6.1 Plant (P) and Controller (Q) architecture. . . . 54
6.2 Generic SMCO Architecture. . . . 55
6.3 Automaton for the hardware plant P of one monitored object. . . . 58
6.4 Automaton for the software plant P of all monitored objects. . . . 59
6.5 Automaton for global controller. . . . 60
6.6 Overall cascade control architecture. . . . 62
6.7 Automaton for secondary controller Q. . . . 62
6.8 Timeline for secondary controller. . . . 62
6.9 Automaton for the primary controller. . . . 63
6.10 SMCO architecture for `range-checker`. . . . 64

6.11 range-checker adds a distributor with a call to the SMCO controller. The distributed passes control to either the original, uninstrumented function body shown on the left, or the instrumented copy shown on the right. . . 65

7.1 Event distribution histogram for the most updated variable (a) and 99th most updated variable (b) in bzip2. Execution time (x-axis) is split into 0.4 second buckets. The y-axis shows the number of events in each time bucket. 69
7.2 *Global controller with range-checker* observed overhead (y-axis) for a range of target overhead settings (x-axis) and two workloads. 71
7.3 *Cascade controller with range-checker* observed overhead (y-axis) for a range of target overhead settings (x-axis) and two workloads. 72
7.4 *Comparison of range-checker accuracy* for both controllers with bzip2 workload. Variables are grouped by total number of updates. 73
7.5 *Observed overhead for global controller clock frequencies* with 4 different clock frequencies and 2 workloads using range-checker instrumentation. . . . 76
7.6 *Accuracy of global controller clock frequencies* with 4 different clock frequencies using range-checker instrumentation. Variables are grouped by total number of updates. 77
7.7 *Local target monitoring percentage (m_{lt}) over time* during bzip2 workload for range-checker with cascade control. Results shown with target overhead set to 20% for 4 different values of K_I and 3 values of T. 78
7.8 *Observed overhead for primary controller K_I values* using range-checker with T = 400ms and four different K_I values. 79
7.9 *Observed overhead for an ad hoc cascade controller's* T *values* with 4 different values of T and two range-checker workloads. 80
7.10 *Observed monitoring percentage over bzip2 range-checker execution.* The percent of each adjustment interval spent monitoring for 2 values of T. The target monitoring percentage is shown as a dotted horizontal line. . . 81
7.11 *Accuracy of cascade controller* T *values* with 4 values of T on the bzip2 workload using range-checker instrumentation. Variables are grouped by total number of updates. 82

List of Tables

4.1 Deadlock freedom for the symmetric and fair C implementation. 40
4.2 Running time of GMC^2 for TCAS. 41

5.1 Bounded model checking with DPR of Randomized MAX-3SAT 52
5.2 Bounded model checking with DPR of NFA for floating-point expressions. 52

7.1 `range-checker` memory usage, including executable size, virtual memory usage (VSZ), and physical memory usage (RSS). 74

Acknowledgments

I would like to gratefully and sincerely thank my two doctoral co-advisors, Professor Scott Smolka and Professor Radu Grosu, for their understanding, encouragement and patience during my Ph.D. studies at Stony Brook University. I shall never forget Scott's willingness and patience to go through all my writings. And Radu was constantly brilliant and instructive, correcting my errors and pointing out new research directions. Without their support, I could not have done what I was able to do.

Thanks to the other two in my dissertation committee, Professor Scott Stoller and Dr. Klaus Havelund. We have been working together in the SMCO project for more than a couple of years. I thank them for their continuing guidance and inspiration.

Thanks to Justin Seyster and Sean Callanan, the two brilliant and diligent HCOS group co-workers from the File System Lab. I feel honored to work with them. Both of them are great implementers. Sean was always sharp and accurate at analyzing technical problems. Justin was the one who gave me the most help during the SMCO project. He generously shared his thoughts with me whenever I had an obstacle. I cannot remember how many times I adopted his ideas and thus solved my problems. I will definitely miss the opportunities to talk with him after I leave the department.

Ketan Dixit, my lab colleague for the last year and a half and co-worker in the SMCO project, kindly helped me design experiments, run benchmarks, and collect tons of data. I thank him for the hard work. It was a pleasant experience to work with him.

Tushar Deshpande, my another lab colleague. It was always fun to talk with him. We shared research progress and exchanged ideas in different areas.

And thanks to my parents, for being all the time supportive and tolerant.

Chapter 1

Introduction

Model checking [16, 55, 10] is a widely used formal verification technique that, given a model of a system, automatically verify if this model meets certain specifications. Traditional model checking is achieved by state-space exploration and restricted to the verification of properties of abstract automata-models. Software model checking [35, 59, 5], however, specifically refers to directly performing model-checking on arbitrary software without the effort of manually abstracting models, therefore holds the very potential to close the gap between the programmer's intent and actual product. Common software model checking approaches fall into two kinds: automatic abstraction-based or concrete execution-based. The execution-based approach usually relies on program instrumentation, or rewriting, or virtual machine interception. Model abstraction and program instrumentation are based on static analysis and require parsing the program source at least into syntax level. Many features of modern programming languages, such as object-orientation, dynamic dispatch, and high-order control flow, appear to be obstacles to these approaches. Support from virtual machines, on the other hand, is restricted to interpreted languages only.

We present a compiler-assisted, execution-based software model-checking technique targeting all languages accepted by the given compiler. We treat the intermediate representation of the program under compilation as an executable language and interpret it using a customized virtual machine. Our model-checking method is based on this virtual machine and has full access to its internal states. We have implemented two model checkers: a stateless Monte Carlo model checker *GMC*^{2} [39], and a bounded concrete-symbolic model checker which uses *dynamic path reduction* [74] for reachability problems of linear C programs.

GMC^2 is a software model checker for C based on the generic Monte Carlo model checking algorithm [40]. Integrated with GCC, it takes the gimlified control-flow graph of target program as input, as well as a C function representing the LTL property of interest. GMC^2 interprets the GIMLPE statements during compilation. The target program can contain concurrency primitives including process fork and POSIX message passing. GMC^2 checks safety properties and liveness properties. In the case of safety properties, the property function is called to check for property violations in the target program. In

the case of liveness properties, the property function is called to check if an accepting state of the target program is visited infinitely often, viewing the target program as a succinct representation of a Büchi automaton.

Dynamic path reduction (DPR) is a general algorithm to prune redundant paths from the state space of a program under verification. It works in the context of the bounded model checking of sequential programs with nondeterministic conditionals. The DPR approach is based on the symbolic analysis of concrete executions. For each explored execution path π that does not reach an abort statement, we repeatedly apply a weakest-precondition computation to accumulate the constraints associated with an infeasible sub-path derived from π by taking the alternative branch to an *if*-statement. We then use a satisfiability modulo theory (SMT) solver to learn the minimally unsatisfiable core of these constraints. By further learning the statements in π that are critical to the sub-path's infeasibility as well as the control-flow decisions that must be taken to execute these statements, unexplored paths containing the same unsatisfiable core can be efficiently and dynamically pruned.

The technique of *runtime verification* [24, 7, 8, 19, 42, 41, 27] emerged in recent years as a complement to exhaustive verification methods such as model-checking and theorem proving, as well as incomprehensive solutions such as testing. It is a combination of formal verification and program execution. Its objective is to ensure that a system satisfies desirable properties at runtime; i.e., each computation of system under inspection is observed and analyzed by a decision procedure called the *monitor*. In contrast to the classical model checking approach, where a simplified model of the target system is verified, runtime verification is performed while the real system is running. Thus, runtime verification or as it is sometimes referred to, passive testing, increases the confidence on whether the implementation conforms to its specification. Furthermore, it allows a running system to reflect on its own behavior in order to detect its own deviation from the pre-specified behavior.

In classic runtime verification, a system is composed with an external observer, called the *monitor*, which is normally an automaton synthesized from a set of properties. Monitors are triggered by events in the sense that every change in the state of the system invokes the monitor for analysis. Therefore the drawback of this event-triggered runtime verification is the potential overhead imposed on the system by the monitor in a short period. Existing runtime verification frameworks, such as DTrace [14, 58] and DProbes [57], let users insert *probes* into a production system at certain execution points. These techniques have some limitations. One of them is that they are always on, meaning that frequently occurring events can cause significant monitoring overhead. This raises the fundamental question: *is it possible to control the overhead due to software monitoring while achieving high accuracy in the monitoring result*?

To answer this question, in this dissertation, we introduce the new technique of *Software Monitoring with Controlled Overhead* (SMCO) [48]. SMCO is formally grounded in control theory, in particular, a novel combination of supervisory control of discrete event systems[63, 1] and linear proportional-integral-derivative (PID) control[72] for continuous systems. Overhead control is realized by disabling interrupts generated by moni-

tored events, and hence avoiding the overhead associated with processing these interrupts. Moreover, such interrupts are disabled for as short a time as possible so that the number of events monitored, under the constraint of a user-supplied target overhead o_t, is maximized.

To ensure the system is controllable, we instrument the application and the monitor so that they emit events of interest to the controller. The controller catches these events, and perform the control logic to control the monitor by enabling or disabling monitoring and event signaling. SMCO uses a source-code instrumentation technique called *compiler-assisted instrumentation* (CAI). CAI is based on a plug-in architecture of GCC [12]. The CAI plug-ins can be dynamically loaded into GCC and then instrument target programs by modifying various GCC internal data structures.

We applied SMCO to a number of monitoring problems. In this dissertation we mainly discuss the case of *integer range analysis*, which determines upper and lower bounds on the values of integer variables, in Section 7.3. We present our result in Chapter 7, followed by discussion about SMCO's ability to control overhead in a high event rate system while retaining accuracy in the monitoring results.

The structure of the rest of this dissertation is as follows. Chapter 2 introduces the compiler-assisted instrumentation technique, with our scripting instrumentation tool *GLua* and its various application. Chapter 3 gives a general survey in software model checking area and introduces a virtual machine implementation at compiler's intermediate representation level (GVM). Chapter 4 discusses GMC^2, the open source model checker based on the generic Monte Carlo model checking algorithm. Chapter 5 explains the dynamic path reduction algorithm for software model checking. Chapter 6 focuses the control theory basis of SMCO. Chapter 7 evaluates performance of SMCO and analyzes various controller optimization factors. Chapter 8 is the conclusion of compiler-assisted technique and future work.

Chapter 2

Compiler-Assisted Instrumentation

Program instrumentation, as a technique to insert code fragments into target programs without hardware support, is widely used in software debugging, testing, monitoring and profiling. Typical program instrumentation techniques include:

- Source-level instrumentation. An additional parser is used to parse the target program and locate the instrumentation spots. Almost all *Aspect-Oriented Programming* (AOP) tools [52, 67, 2] fall into this category. AOP reads the target program, the additional code (*advice*) that one wants to apply to existing model, and the specification of execution spots (*pointcut*) where advices shall be applied, then outputs new versions of the program with instrumentation code inserted in. This process is called *code-weaving* [65].
- Abstract-Syntax Tree (AST) level instrumentation. One example is the *C Intermediate Language* (CIL) [31, 60]. CIL is a simplified subset of C, as well as a set of tools for transforming C programs into that language. CIL compiles all valid C programs into a few core constructs with a very clean semantics. Its syntax-directed type system makes it easy to analyze and manipulate C programs. Several other tools use CIL as a way to have access to a C abstract- syntax tree.
- Compiler directly supported instrumentation, for example, the *gnu gprof* with support from the *GCC* compiler, which can inject profiling code at the beginning and the end of functions and around the call sites by a command line option *-pg*.

All these software instrumentation techniques have some drawbacks. First of all, all these approaches are *code-oriented*: they interpose on execution of specified instruction in the code; events triggered by hardware or from environment (e.g., network packages) can hardly be tracked. For AOP approaches, the code patterns of determining pointcuts are usually limited into, e.g., before/after call sites, function front/ends, variable assignments, and so on. Furthermore, any AOP tool is bounded with only one specific language and requires a tedious re-implementation of program parser.

2.1 GCC Intermediate Representation

Program instrumentation in SMCO is facilitated by a technique called *Compiler-Assisted Instrumentation* (CAI). CAI is based on a plug-in architecture for GCC [13, 12]. It can separately compile instrumentation plug-ins as shared libraries, which are then dynamically loaded into GCC. Plug-ins have read/write access to various GCC internal structures, including abstract-syntax trees (ASTs), control flow graphs (CFGs), three-address code (GIMPLE statement), static single-assignment (SSA) and register-transfer language (RTL). In August 2010, the plug-in architecture has been officially adopted into GCC 4.5 [32]. Based on the GCC compiler, CAI achieves the following advantages:

Versatility: Access to the full parse tree and control flow graph of a program allows instrumentation of a wide variety of code patterns with full type information; furthermore, CAI accepts any program in any language supported by the GCC compiler.

Accuracy: Instrumentation can be used in combination with full compiler optimization, making results as close as possible to the uninstrumented program.

Speed: Compiler-assisted instrumentation makes monitoring functionality part of the program itself.

CAI focuses on GCC whose architecture is illustrated in Figure 2.1. We now briefly discuss each individual intermediate representation used by GCC and explain why the *GIMPLE* intermediate representation is the most suitable level to be instrumented.

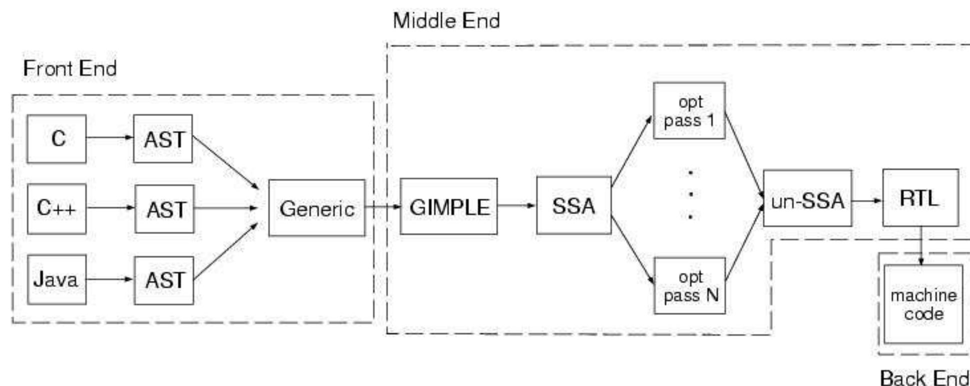


Figure 2.1: The GCC architecture

Abstract-Syntax Trees (AST) After parsing program source, compilers represent the program as a collection of trees, namely the abstract-syntax trees. Each tree denotes the syntactic structure of a function. Each node of the tree denotes a construct occurring in the function. The syntax is "abstract" in the sense that it does not represent every detail that appears in the real syntax. For instance, grouping parentheses are implicit in the tree structure, and a syntactic construct such as an if-condition-then expression may be denoted by a single node with two branches.

At the AST level, instrumentation can most easily detect common programming

idioms — such as use of iterators in a loop — without performing extra analysis.

GENERIC Form In GCC, the GENERIC [56] form simply provides a language-independent way of representing ASTs, which differ from one programming language to another. Some new tree nodes are added to this form. Certain information of specific language idioms, e.g., `x++` in C/C++, are removed from generic form. The GENERIC representation is used as the input to compiler's middle-end.

GIMPLE Form GIMPLE [56] is a simplified GENERIC, in which various constructs are lowered to multiple three-address statements. These statements take no more than three operands (except function calls). For example, a GIMPLE assignment takes at most two values and applies an operator on them to produce a third. GIMPLE statements retains all the type information that was discovered during parsing, and share the building blocks of the abstract-syntax tree representation, but much are simpler to manipulate. Since the number of operands are limited, GIMPLE statements can easily be written in the form of tuples. For instance, an assignment `x=a+1;` can by represented as a 5-element tuple `(GIMPLE_ASSIGN, x, +, a, 1)`.

The compiler pass which converts GENERIC to GIMPLE is referred to as the *gimplifier*. The gimplifier works recursively, generating GIMPLE tuples out of the original GENERIC expressions. This procedure is called *lowering*. To *lower* multiple-operand GENERIC statements into three-address GIMPLE statements, the *gimplifier* generates temporary variables for intermediate values; additionally, it simplifies the control-flow structure by constructing a control-flow graph and replacing more sophisticated structures with conditional *goto*s. Figure 2.2 shows a C program and its corresponding GIMPLE representation.

```
1.  int foo( )  {
2.      int a, b;
3.      a = 5;
4.      b = 2 * bar(a);
5.      if (a > b)
6.        return a+(a*b);
7.      else
8.        return a;
9.  }
```

```
 1.  int foo( )  {
 2.      int a, b, T1, T2, T3;
 3.      a = 5;
 4.      T1 = bar(a);
 5.      b = 2 * T1;
 6.      if (a > b) goto L0 else goto L1
 7.  L0: T2 = a * b;
 8.      T3 = a + T2;
 9.      return T3;
10.  L1: return a;
11.  }
```

Figure 2.2: Sample C Program and Corresponding GIMPLE representation

Static Single Assignment (SSA) Form SSA form [23, 61] is a special GIMPLE representation in which every variable is assigned exactly once. Variables in the original GIMPLE are split into versions, new variables typically indicated by the original name with a subscript, so that every definition gets its own version.

Sometimes, flow of control makes it impossible to determine the most recent version of a variable. In these cases, the compiler inserts an artificial definition for that variable called *PHI* function or *PHI* node. This new definition merges all the incoming versions of the variable to create a new name for it. For instance,

```
1.  if (...)
```

```
2.      a_1 = 5;
3.  else if (...)
4.      a_2 = 2;
5.  else
6.      a_3 = 13;
7.
8.  # a_4 = PHI <a_1, a_2, a_3>
9.  return a_4;
```

Most of the tree optimizers rely on the data flow information provided by the SSA form, including constant propagation, dead code elimination, global value numbering, partial redundancy elimination, and register allocation.

Register-Transfer Language (RTL) The last part of the compiler work is done on a low-level intermediate representation called the Register-Transfer Language. In this language, the instructions to be output are described, pretty much one by one, in an algebraic form that describes what the instruction does. RTL is close to assembly code. It determines the kind of storage that each variable requires, then performs register allocation and final instruction selection.

For program instrumentation, the GIMPLE intermediate representation is highly recommended because of its simplicity, flexibility, convenience and language independence. The GIMPLE representation is designed for ease of manipulation by programmers [61]. GCC developers have already provided a rich API to manipulate GIMPLE statements [20]. In contrast, GENERIC form and RTL form are much more complicated. At GIMPLE level, a function's control-flow graph is ready for data-flow and control-flow analyses. Also the three-address code form provides a very simple syntactic structure which reduces the number of side-effects that must be considered when instrumenting. The fact that most optimization passes are performed at the GIMPLE level gives developers more flexibility whether or when to insert their own transformation pass. For example, running a virtual machine at GIMPLE level usually prefers to be done prior to constant propagation optimization pass, so that it is easier to track which line of source code corresponds to the virtual machine's current program counter; if it runs after all optimization passes, then one achieves maximum efficiency: least code to be executed.

2.2 Instrumenting GIMPLE

2.2.1 GCC Plug-in Architecture

Before GCC 4.5 was released, we extended GCC to support *plug-ins* by modifying the GCC source code. This modification is small and does the following tasks:

- Compile libtool library *ltdl* into GCC and link GCC with *-export-dynamic* option. This allows GCC to load external shared objects and allows a shared object to access GCC interfaces.
- Add the compiler option *-ftree-plugin=<filename>* to allow users to specify plug-ins and provide arguments to those plug-ins on the command line.
- Modify GCC's optimization pass manager and set up several additional optimization passes which will invoke functions that GCC finds in the loaded plug-ins. Two

new optimization passes are inserted at the beginning and end of translation for each file, i.e., they are executed once per translation-unit. Other passes are called once for every function in the program under compilation. The most commonly used one is inserted right after the *inter-procedural analysis* pass, allowing the plug-in to manipulate the GIMPLE representation of the current function, including the control-flow graph.

Since the release of GCC 4.5, we ported our plug-in implementation to the new architecture. GCC 4.5 supports plug-in by nature and provides a rich plug-in API which offers more flexibility and enforces code security. However, basic ideas of the two architectures are similar. Hence the migration from our old architecture to GCC 4.5 is trivial.

2.2.2 GCC Internals

Instrumenting programs under compilation is all about manipulating GCC's internal data structures. Every GCC plug-in developer must have knowledge about those key data structures and corresponding programming interfaces, which, unfortunately, is not nicely documented.

From a plug-in developer's view, the most important data structures in GCC are *tree*, *gimple statement*, *basic block*, and *function*.

Tree Tree is the central data structure used by internal representation. A tree is a pointer type. The object it points to may be of a variety of types. Every construct in the program under compilation is a tree in GCC, e.g., identifiers, type definitions, constant values, strings, expressions, statements, blocks, functions, or even container data structures such as vectors and lists. For each type of the tree, there is a set of macros and functions can apply. For example, for a tree denoting a function call expression, there are macros or functions to get function name, function type and a way to iterate all its parameters. Cautions are needed to manipulate trees since applying wrong macros (functions) to a tree will cause compiler internal error.

Gimple Statements Gimple statements in GCC are tuples of different types and variable length. The first element in gimple statement tuple is always its type. The rest of this tuple depends on the type and operands of the gimple statement. For example, for a conditional goto statement, the tuple contains these elements: the type, the conditional expression, the label of the destination of then-branch, and the label of the destination of else-branch. GCC provides API to traverse gimple statements within a basic block or function, access any element in a gimple tuple, and insert/remove a gimple statement into/from a sequence of gimple statements.

Basic blocks Basic blocks are straight-line sequences of gimple statements with only one entry and only one exit. Each basic block has a unique identifier and denotes one node in the control-flow graph. It may have multiple predecessors and multiple successors. Connected basic blocks are linked by another data structure *edge*s. Edges represent possible control flow transfers from the end of some basic block A to the head of another basic block B. GCC provides API to visit all basic blocks in a CFG. The following snippet illustrates a common way to traverse all the statements

of the program in the GIMPLE representation.

```
basic_block bb;
FOR_EACH_BB(bb)
{
    block_stmt_iterator si;
    for (si=bsi_start(bb); !bsi_end_p(si); bsi_next(&si))
    {
        tree stmt = bsi_stmt(si);
        print_generic_stmt(stderr, stmt, 0);
     }
}
```

Function Every *function* data structure in GCC represents a corresponding function definition in program source, if not inlined. A *function* has four core parts: the name, the parameters, the result, and the body. GCC provides a global variable `cfun`, which is a pointer to a *function* data structure representing current function, as an implicit parameter for all optimization passes. Starting from `cfun`, one is able to track down till any tree node in any gimple statement.

2.2.3 Pass Managers

Code optimization in GCC occurs during any phase of compilation, however most optimizations are performed after the syntax and semantic analysis of the front-end and before the code generation of the back-end. In other words, optimization mostly occurs at middle-end upon GENERIC or GIMPLE form.

Passes are those procedures that GCC invokes to analyze or transform intermediate representations. Typically one pass serves for one specific purpose, e.g., building control-flow graph. GCC defines about 60 gimple optimization passes. When one turn on the *-O2* compiler flag, 53 gimple optimization passes will be invoked [33].

Results by one pass persists to the next pass. Some passes rely on the output of previous passes to proceed, e.g., all Tree-SSA passes require the *build_ssa* pass. This dependency relation between passes are managed by GCC's *pass manager*. Its job is to run all of the individual passes in the correct order, and take care of standard bookkeeping that applies to every pass.

The GCC plug-in architecture allows users to add their own custom passes without patching and recompiling the GCC source code. The plug-in shared object shall export a function called `plugin_init` that is called right after the plug-in is loaded. This function is responsible for registering all the callbacks required by the plug-in and do any other required initialization. GCC's plug-in API provides basic support for inserting new passes or replacing existing passes. A plugin registers a new pass with GCC by calling `register_callback` with the `PLUGIN_PASS_MANAGER_SETUP` event and a pointer to a `struct register_pass_info` object. GCC inserts the callback function or replaces existing ones with it at any position in the chain of passes.

2.3 GLua: Scripted Program Instrumentation

Although the GCC plug-in architecture provides great convenience for the development and maintenance of GCC expansions, the developers may still suffer from the complexity of GCC internals. Though the GIMPLE intermediate representation is the simplest level to manipulate, a plug-in developer still needs to directly deal with low-level GIMPLE language, sometimes delve into the even more cumbersome tree nodes manipulations. Writing a plug-in therefore requires quite a familiarity with GCC internals.

To address this problem, we developed *GLua*, a special GCC plug-in that reads and executes user-specified script to facilitate plug-in development. *GLua* 's architecture is depicted in Figure 2.3.

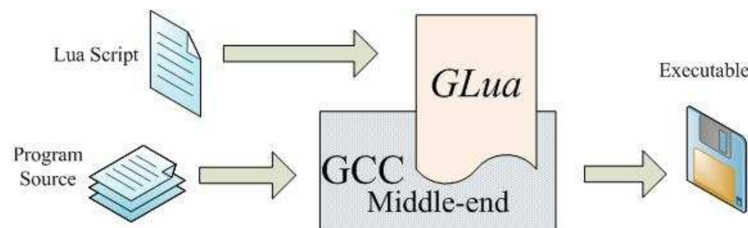


Figure 2.3: Architecture of GLua

GLua is an idea to export the accesses of the most important GCC internal data structures to a light-weighted, easy-to-extend scripting language *Lua* [54]. Functionalities that used to require a lot of coding, such as control-flow graph traversal, gimple statement manipulation, tree node properties extraction, now turn into a few lines of Lua code. We exported GCC's *tree*, *gimple statement*, *basic block*, *function* data structures as four Lua objects. Functions to access these GCC data structures turn to be Lua methods of corresponding object, e.g., *entryblock* for the *function* object to retrieve the entry basic block.

Using *GLua* saves user from directly handling low-level data structures. Lua's simple procedural syntax and powerful data description facilities help C programmers to grasp program logic better. Also a Lua script is much faster to write and easier to debug than a C program.

2.3.1 GLua Usage

GLua is implemented as a plug-in that runs a Lua script when the plug-in's pass get chances to execute. Path of the Lua script and various arguments are passed to *GLua* via command line. *GLua* also supports interactive mode. User can pause the execution and talk with Lua virtual machine by entering Lua language statements, just like running *Python* in a console.

This easiest case of invoking *GLua* is by command line arguments. By default, *glua* accepts the following arguments.

- script=*path/to/lua/script*: specify Lua script to execute; default is *gcclua.lua*.

- ref-pass-name=*name*: specify a reference pass name; default is **all_optimizations*.
- insert-*[after|before|replace]*: execute Lua script *after*/*before* or *replace* the referenced pass; default is *after*.
- interactive: run in interactive mode.

In GCC 4.5, arguments passed to plug-in shall be prefixed `-fplugin-arg-`. Therefore a typical command to invoke *GLua* is like as follows:

```
gcc4.5 -c -g -O2 test.c -fplugin=libgcclua.so
  -fplugin-arg-script=duplicate.lua
  -fplugin-arg-ref-pass-name=ccp -fplugin-arg-insert-before
```

GLua also provides a way to specify arguments via environment variables. Supported environments variables are:

- GCCLUA_SCRIPT = *path/to/lua/script*
- GCCLUA_REF_PASS = *name*
- GCCLUA_INSERT = *[after|before|replace]*
- GCCLUA_INTERACTIVE = *[false|true]*: default is *false*.

With the environment variable settings loaded in a shell, a user can use a much shortened command line: only has to specify the path to the Lua script. Note if an environment variable argument conflicts with a command-line argument, *GLua* acknowledges the command-line argument.

Once a Lua script is loaded by *GLua*, it tries to execute three global functions with specific names declared in the script:

gcclua_pre This Lua function will be executed when the *GLua* plug-in is initialized. In this function, a user usually initialize local variables or open log files.

gcclua_post This Lua function will be executed when the optimization passes have been all finished. Typically this is the chance that a user can report synthesized results or finalize a log file.

gcclua This Lua function will be executed once for each function, depending on user specifications, before or after a particular reference optimization pass. *GLua* will not pass any parameter to this function. However, it can access *GLua*'s global variable *cfun* to get the handle of current function being investigated. Given the *cfun* object, the user can track down to all its local variables, basic blocks, gimple statements, etc.

The Lua script in Figure 2.4 shows an example of *GLua* script. This script counts the number of functions in a source file and prints all their referenced variables.

2.3.2 The GLua API

Note that using *GLua* does not mean the users do not have to have any knowledges about GCC internals at all. The users still need to know basic ideas of *tree*s, *gimple*s, and *basic block*s. However, they are now free from much detailed GCC internal macros and APIs.

```
local count

function gcclua_pre()
    count = 0
end

function gcclua_post()
    print("total functions: ", count)
end

function gcclua()
    count = count + 1
    local fn = gcc.cfun
    print(fn);
    local vars = fn:referenced()
    print('referenced:', #vars)
    for _, v in ipairs(vars) do
        print(v, desc)
    end
end
```

Figure 2.4: A GLuascript to count functions and prints their referenced variables

Respectively, these *GLua* objects supports the following methods:

- *tree* object

artificial	*tree* is compiler-created artificial node.
operand(i)	if *tree* is an expression, return the *i*-th operand.
iscomponentref	if *tree* is a pointer to a structure.
isconstructor	if *tree* is a structure variable initializer.
isarrayref	if *tree* is an array address.
n_operands	if *tree* is an expression, return the number of its operands.
codeinfo	return a tuple of four: (operand, tree code name, tree code class, tree code length).
ptr	return the absolute address of *tree*.
type_decl	if *tree* is a type declaration, return the name.
static	if *tree* is a static variable.
public	if *tree* is a public variable.
isptr	if *tree* is a pointer.
decl_initial	if *tree* is a variable declaration, return its initial value.
istype	if *tree* is a type definition.
code	return the tree code number.
type_cmp(another)	check if two *tree*s are definition of the same type, e.g., *unsigned int* and *size_t*; throws an error if either of two is not a type..
isfndecl	if *tree* is a function definition.

type_quals	if *tree* is a type, return a tuple of three boolean values: (const, volatile, restrict), namely the type qualifiers.
isindirectref	if *tree* is an indirect reference, e.g., `*p` in C.
isvardecl	if *tree* is a variable declaration.
type	return the type of *tree*.
isssaname	if *tree* is a temporary *ssa* (single static assignment) name variable.
addressable	if *tree* is an addressable variable, i.e., can be used at the left hand side in an assignment.
ssaname_var	if *tree* is an *ssa* variable, return the actual variable being referenced.
isdecl	if *tree* is a variable or function declaration.
decl_name	if *tree* is a declaration, return the name of the object as written by the user.
constant	if the *tree* expression is constant.
readonly	if the type *tree* is const-qualified.
volatile	if the type *tree* is volatile-qualified.
decl_isexternal	if *tree* is declared externally.
iscmp	if *tree* is a comparison expression.
isref	if *tree* represents a reference.
isconstant	if *tree* represents a constant value.
properties	for any *tree* node, return a list of applicable properties and their values; user can iterate this list and find out all available information of the *tree*. its usage is shown in Section 2.4.2.

- *gimple* object

ptr	return the absolute address of *gimple*.
basicblock	return the *basicblock* object containing this *gimple*.
codename	return *gimple*'s code name in text form. i.e., gimple_assign, gimple_cond, ...
n_operands	return the number of operands of *gimple*.
operand(i)	return the *i*-th operand of *gimple*.
exprcode	if *gimple* is an assignment, return the right-hand side expression operator code; if *gimple* is a conditional statement, return the code to comparison expression.
location	return the location of *gimple* in a pair: (filename, lineno).

tuple	returns variable length tuple according to type of *gimple*. returned tuple can be either case of the following:

  - "gimple_assign", 1hs, opcode, rhs1, rhs2
  - "gimple_call", lhs, fn, {args...}
  - "gimple_cond", lhs, opcode, rhs, true_label, false_label
  - "gimple_goto", label
  - "gimple_label", label
  - "gimple_nop"
  - "gimple_phi", result, {arg1=from1, ...}
  - "gimple_return", retval
  - "gimple_switch", index, {case_labels...}, default_label

- *basicblock*

ptr	return absolute address of the *basicblock* object.
index	return the index number of current *basicblock*.
prev	return the previous *basicblock* in chain.
next	return the next *basicblock* in chain.
preds	return the set of predecessors of current *basicblock*.
succs	return the set of successors of current *basicblock*.
gslist	return the iterator of *gimple*s in *basicblock*.
philist	return the iterator of ϕ-statements in *basicblock*. note that a basic block may have several ϕ-statements, and they can only be accessed by ϕ-statement iterators.
loopdepth	return loop depth of *basicblock*.
insert_before(ref, new)	insert *new gimple* statement before reference statement *ref*.
insert_after(ref, new)	insert *new gimple* statement after reference statement *ref*.

- *function*

ptr	return absolute address of the *function* object.
decl	return the function declaration *tree* object.
entryblock	return the entry *basicblock*.
exitblock	return the exit *basicblock*.
referenced	return a set of *tree*s represents variables referenced by this *function*.
params	return the list of function parameters as *tree*s.
n_basicblocks	return the number of basic blocks in the *function*.
basicblocks	return the set of all *basicblock*s in the *function*.

n_edges	return the number of edges.
edges	return the set of all edges connecting basic blocks in pairs: (src, dest).
name	return the function name.
duplicate_body(tmpvar, dist)	makes another copy of function body (blocks except entry and exit along with their topology). it also adds a distributor block to direct to two bodies. The distributor block is like the following:

```
1   int tmpvar;
2   tmpvar = dist();
3   if (tmpvar != 0)
4       goto <NEW_BODY>
5   else
6       goto <OLD_BODY>;
```

if *dist* is NULL, line 2 is replaced by `tmpvar = 0;` then it is the user's duty to further populate the distributor block.

The Lua script in Figure 2.5 shows how to traverse and print all basic blocks and GIMPLE statements in a function.

```
function gcclua()
    local fn = gcc.cfun
    print(fn);
    local bb = fn:entryblock()
    while (bb) do
        local line = tostring(bb) .. '  [ '
        for _, succ in ipairs(bb:succs()) do
            line = line .. '#' .. tostring(succ:index()) .. ' '
        end
        print(line .. ']')
        for gs in bb:gslist() do
            print("  "..tostring(gs))
        end
        bb = bb:next()
    end
end
```

Figure 2.5: A GLua script to traverse basic blocks and gimple statements

Besides these objects, GLuasupports the following auxiliary functions or data members. They are mostly used for program instrumentations.

filename	filename of current translation unit under compilation.
quit_compilation	force GCC to quit compilation.

cfun	current function under work.
dump_stack	print Lua VM's current stack frames.
recursive_dump(table)	a utility function to recursively print the content of a Lua table.
static_vars	return the set of all global variables.
build_tmp_var(ty)	make a temporary function local variable of type *ty*.
build_local_var(ty, name)	make a function local variable with specified *name* of type *ty*.
get_builtin_type(typename)	get a *tree* representing the type specified by *typename*. currently *GLua* supports: "void", "bool", "char", "unsigned char", "short", unsigned short", "int", "unsigned int", "long", "unsigned long", "long long", "unsigned long long", "float", "double", "void *", "char *", "int *", "float *", "double *".
build_string(str)	build a string constant.
build_int_cst(n)	build an integer constant.
build_function_decl(ret_ty, name)	build a function declaration with specified return type and function name, but with zero parameters.
build_gimple_assign(lhs, op, rhs1, rhs2)	create a new *gimple* assignment statement: `lhs = rhs1 op rhs2`.
build_gimple_call(lhs, fn_decl, args, ...)	create a new *gimple* function call statement: `lhs = fn_decl(args, ...)`

Usage of these auxiliary functions are best shown in Figure 2.12, where we insert a trace function at the beginning of every basic blocks except the entry block and the exit block.

Due to the capability of introspection of a dynamic language, the users can always use the program in Figure 2.6 to print out the entire *GLua* API.

```
function gcclua()
    local libs = { "gcc",
                   "gcc_function",
                   "gcc_basicblock",
                   "gcc_gimple",
                   "gcc_tree" }
    for _, lib in ipairs(libs) do
        print(' * ' .. lib .. ' * ')
        for k, v in pairs(_G[lib]) do
            print(type(v), k)
        end
        print()
    end
    gcc.quit_compilation(0)
end
```

Figure 2.6: glua-help: a GLua script to list GLua API

2.3.3 Related Work

Although *GLua* is working on the cutting edge of GCC plug-in framework, it is not the only known compiler-based program instrumentation technique. Almost at the same, our research group developed *InterAspect* [65], an aspect-oriented instrumentation framework, upon GCC plug-in infrastructure. *InterAspect* allows instrumentation plug-ins to be developed using the familiar vocabulary of Aspect-Oriented Programming *pointcuts*, *join points*, and *advice functions*. In AOP, a *pointcut* denotes a set of program points, called *join points*, where calls to *advice functions* can be inserted by a process called *weaving*. But unlike traditional AOP systems which implement a special AOP language to define pointcuts, *InterAspect* provides a C API to do this.

InterAspect also supports customized instrumentation, where specific information about each join point in a pointcut, as well as results of static analysis, can be used to customize the inserted instrumentation. *InterAspect*'s API allows users to customize the weaving process by defining *callback functions* that get invoked for each join point. Callback functions have access to specific information about each join point; the callbacks can use this to customize the inserted instrumentation, and to leverage static-analysis results for their customization.

The benefits of *InterAspect* are obvious. It is based on GCC plug-ins and has access to GCC internals, therefore allows users to exploit static analysis during the weaving process. It simplifies program instrumentation by not having to be intimately familiar with GCC internals. Furthermore it does not require knowledge to any AOP language; a programmer with just C background can do.

Both based on GCC plug-in infrastructure and acting for GCC internals, *GLua* differs from *InterAspect* by exposing GCC internals with Lua objects. Instead of C API. *GLua*'s objects — *tree*, *gimple*, *basicblock* and *function*, are adopted from GCC's notion, which means *GLua* is not GCC-transparent. However, *GLua* objects offer more information and programming flexibility than *InterAspect*. For example, *GLua* exposes all properties of a tree node, and allows iterating basic blocks and GIMPLE statements in a function. Utilizing Lua's scripting power, *GLua* can easily realize *InterAspect*'s capability with less coding.

As *InterAspect* is purposed to perform AOP-style program instrumentation, *GLua* can be viewed more than a program instrumentation tool. Section 2.4 shows some applications of *GLua* used as debugger and profiler.

2.4 GLua Program Instrumentation Tools

Upon the *GLua* framework, we developed a series of tools to facilitate program instrumentation. These tools can also be considered applications of *GLua*.

2.4.1 Control-Flow Graph Visualizer

Control-Flow Graph Visualizer is inspired by the *GIMPLE Development Environment (GDE)* [25] and *LLVM* [70]. The GDE tool consists of a GCC plug-in which extracts GCC's interme-

diate representation and dump it in text form to a file, and a Java program which loads the text file and visualize the CFG. GDE is interactive: CFG nodes are draggable inside the visualizer window; tree properties are shown in side panel by clicking CFG nodes. LLVM's CFG visualizer is merely an optimization pass which collects CFG information and dumps it as a plain-text graph description file (*.dot*), which can be converted to a picture by *graphviz* [4].

Our CFG Visualizer is a *GLua* script which: (1) dumps CFG in XML form; (2) converts the XML file to *graphviz* input language; (3) render it on screen. Figure 2.7 shows a sample C program and Figure 2.8 its corresponding CFG rendering.

```
int simple(int x, int y) {
    x *= 2;
    y += 5;
    if (x >= y)
        return 1;
    else
        return 0;
}
```

Figure 2.7: Sample C program

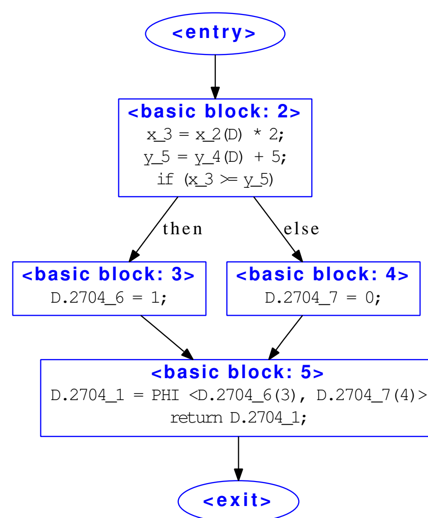
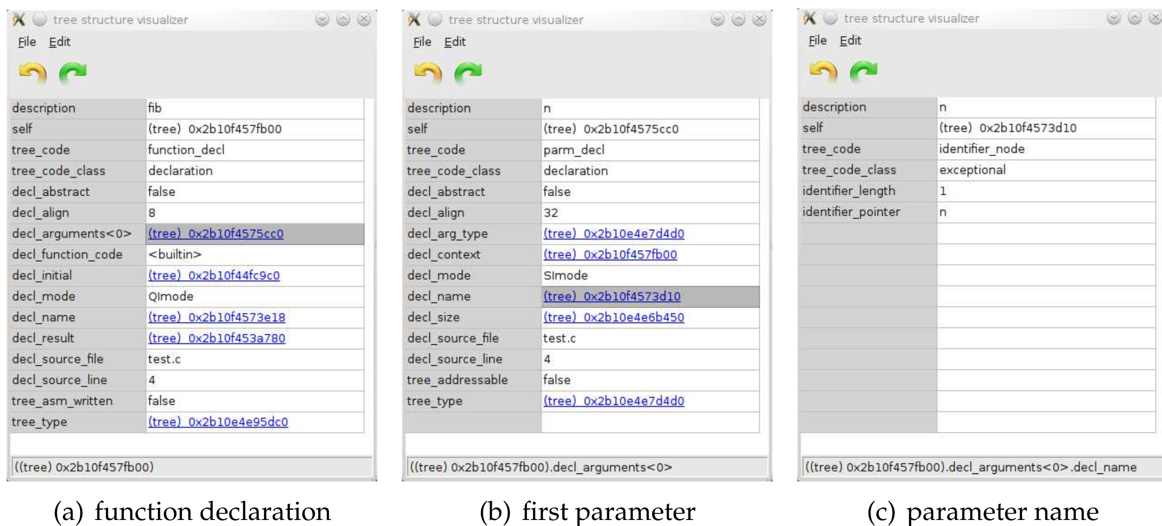


Figure 2.8: Control-flow graph of function `simple()`

Using CFG Visualizer facilitates debugging plug-in. As it shows the effect of instrumentation on screen, the developers gain insight into whether the instrumentation task has correctly taken effect, without having to deploy and run instrumented program. It also helps understand how GCC optimizes code at GIMPLE level.

2.4.2 Tree Tracker

When coding GCC plug-in, one of the challenging problems is accessing the tree node. Applying macros on wrong types of trees will cause GCC to abort, with very vague error message. The *tree tracker* provides a way to visually show the type, memory address, and text description of a tree node, along with all properties the tree node may have. It it shown as a two-column table with property names at the left column and the values of the properties at the right column; see the three screenshot pictures in Figure 2.9.



(a) function declaration (b) first parameter (c) parameter name

Figure 2.9: Tree Tracker

Note that if the value of a property is an another tree node, then this tree node is printed in blue color and click-able. In Figure 2.9, picture (a) is the function declaration tree node of function `int fib(int n)`; picture (b) is the parameter declaration of that function; picture (c) is the name identifier of that parameter. Picture (b) is brought by clicking the greyed area in picture (a) and picture (c) is brought by clicking the greyed area in picture (b).

Tree tracker is bounded with GCC and requires no intermediate file dump. Its GUI infrastructure is *wxlua*, a lua export of *wxWidgets* [68] library.

2.4.3 Function Duplicator

Function duplicator creates a copy of the body of every function to be instrumented. A distributor block at the beginning of the function calls a user-specified external function to determine which version of function body shall be executed: the original one or the duplicated one. It is especially suitable for the cases that the user wants to dynamically toggle instrumentation at runtime. For that the user can instrument only the duplicated function body and use the external distributor function to switch between instrumented and uninstrumented code. Figure 2.10 shows a typical code snippet doing function duplication. Figure 2.11 shows the effect after function duplication upon the C program listed in Figure 2.7.

```
if dist_call == nil then
    local tree_inttype = gcc.get_builtin_type("int")
    local tree_dist_call = gcc.build_function_decl(
                              tree_inttype, "distribute")
    dist_call = gcc.build_gimple_call(nil, tree_dist_call)
end
fn:duplicate_body("__tmpvar__", dist_call)
```

Figure 2.10: Sample Lua code doing function duplication

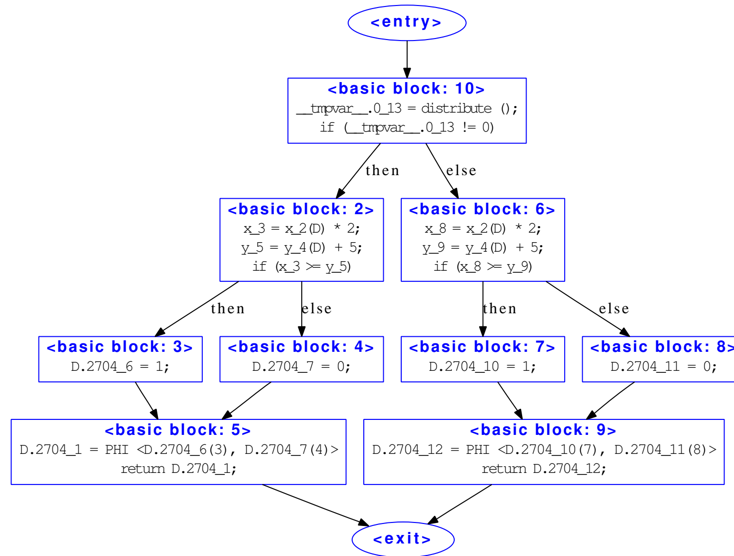


Figure 2.11: duplicated function simple ( )

2.4.4 Basic Block Profiler

GCC's program profiling tool *gprof* [38] is limited at function level. However sometimes profiling inside a function may give programmers more insight on where to further improve program execution efficiency, for example, a frequently executed loop body. It may also lead to a direction of so called *profile-guided optimization (PGO)* [49]. For example, information obtained by execution path profiling allows the compiler to put sequential basic blocks in a frequently executed path conjointly in the object file. Therefore their code are more likely to appear in same memory page, thus improves CPU cache-hit chance.

Our *basic block profiler* is a good illustration of how to instrument program using *GLua*. It inserts a `visit_basicblock` function at the beginning of every basic blocks. This `visit_basicblock` function is defined at an external library which need to be linked with the program being instrumented. It takes the function id and basic block index as parameters:

```
void visit_basicblock(register int fn_id, register int bb_id)
```

The two parameters `fn_id` and `bb_id` are fixed values provided by GCC compiler

and passed to `visit_basicblock` as constants.

When the basic block is executed, `visit_basicblock` will record path information. When program exits, the external library saves collected path information to disk file, particularly an XML file in our implementation.

The main part of the *basic block profiler* script is listed in Figure 2.12.

```
function gcclua()
    local fn = gcc.cfun
    local bb = fn:entryblock()
    while (bb) do
        if bb ~= fn:entryblock() and bb ~= fn:exitblock() then
            local tree_fnid, tree_bbid, tree_voidtype
            local tree_fndecl, gs_call

            tree_fnid = gcc.build_int_cst(fn_id)
            tree_bbid = gcc.build_int_cst(bb:index())
            tree_voidtype = gcc.get_builtin_type('void')
            tree_fndecl = gcc.build_function_decl(
                                tree_voidtype, 'visit_basicblock')
            gs_call = gcc.build_gimple_call(
                            nil, tree_fndecl, tree_fnid, tree_bbid)
            for gs in bb:gslist() do
                if gs:codename() ~= 'gimple_label' then
                    if not bb:insert_before(gs, gs_call) then
                        print('... failed')
                    end
                    break
                end
            end
        end
        bb = bb:next()
    end
    fn_id = fn_id + 1
end
```

Figure 2.12: basic block profiler *script*

2.4.5 Symbolic Executor

Symbolic execution is primarily use in model checking [17, 74]. It may also provides informations that can not be achieved by compiler's optimization passes or any static analyzers. For example, symbolic execution can detect infeasible execution paths that static analysis cannot find. Consequently, using the computation result from symbolic execution, a compiler can perform more extensive dead-code elimination, leading to reduced code page size and improved cache-hit frequency.

Our symbolic executor is based on the SMT tool *Yices* [21]. To invoke *Yices* from Lua, we developed a *Yices* Lua wrapper *YicesLua*. This wrapper library exports a part of *Yices* C API and wrap it in Lua interface so that *GLua* can call *Yices*.

The symbolic executor takes the SSA-form control-flow graph and path branching choices as input. The path branching choices indicates which branch to take when the executor meets a conditional goto statement. If this path is feasible, the symbolic executor answers yes; otherwise answers no and reports how far along the path did the execution fail. Given the C program as follows:

```
int foo(int n) {
    int x = 1, y = 7;

    if (y > n)  y -= n;

    if (x <= y) x += y;
    else        x = 2*y-1;

    if (x > 1) return 1;
    else       return 0;
}
```

the symbolic executor outputs

```
function: int foo (int)
checking path 000: no, failed at depth: 2
checking path 010: no, failed at depth: 3
checking path 100: no, failed at depth: 2
checking path 110: no, failed at depth: 3
checking path 001: no, failed at depth: 2
checking path 011: yes
checking path 101: no, failed at depth: 2
checking path 111: yes
```

which means only paths $\{false, true, true\}$ and $\{true, true, true\}$ are feasible. So at the last *if*-statement, only the *then*-branch is possible to be taken. Thus the program is functionally equivalent to

```
int foo(int n) {
    return 1;
}
```

Chapter 3

Compiler-Assisted Software Model Checking

3.1 Overview of Software Model Checking

The root of software model checking can be traced back to logic and theorem proving. Initially the focus of program verification research was on manual reasoning, and the development of axiomatic semantics and logics for reasoning about programs provided a treated as logic objects [46, 26]. As the size of software systems grew, providing entire manual proofs became too cumbersome. This situation leads to a trend toward automating program reasoning, and further broadened the scope of the automated techniques, both in the scale of programs and in the variety of properties that can be checked.

In the Eighties, the researches of techniques and the development of accompanying tools for automatic *model checking* for *temporal logics* [15, 62] provided much powerful algorithmic tools for state-space exploration. Meanwhile the theory of *abstract interpretation* [22], as one of the static analysis techniques, established a link between the infinite state spaces of real programs and the logical finite representations. More recently, techniques of *symbolic model checking* with *BDD*s [55] or *SAT-Solver*s [10] further extended program scale that can be handled.

Modern software model checkers appear to be combinations of these techniques that traditionally classified as theorem proving, or model checking, or static analysis. Moreover, we tend to regard tools that perform throughout testing also as software model checkers, as long as the most important characteristic of model checking is exhibited: when a specification is found not hold, a *counterexample* is given.

3.1.1 Abstract Software Model Checking

For infinite state programs, concrete model checking may run out of resources and symbolic reachability analysis may not terminate, or take an inordinate amount of time or memory to terminate. Abstract model checking trades off precision of the analysis for efficiency. In abstract model checking, reachability analysis is performed on an abstract domain which captures some, but not necessarily all, the information about an execu-

tion, using an abstract semantics of the program [22]. A proper choice of the abstract domain and the abstract semantics ensures that the analysis is sound (i.e., proving the safety property in the abstract semantics implies the safety property in the original) and efficient.

But in general abstract model checking is incomplete, i.e., the abstract analysis can return a counterexample even though the program is safe. Therefore a technique called *refinement* is needed to determine whether the counterexample is genuine, i.e., can be reproduced on the concrete program. If the counterexample is spurious, we would additionally like to automatically refine the abstract domain, that is, construct a new abstract domain that can represent strictly more sets of concrete program states.

Famous abstract model checkers include:

Slam The *Slam* [6] model checker introduced *boolean program*s, an imperative programs where each variable is boolean, as an intermediate language to represent program abstractions. A tool (called *c2bp*) implemented predicate abstraction for C programs. A tool called *bebop* performs symbolic execution upon the boolean program. Finally, abstraction refinement was performed by a tool called *newton*, which implemented a greedy heuristic to infer new predicates from the trace formula. Slam was used successfully within Microsoft for device driver verification and developed into a commercial product.

Blast The Blast model checker [45, 9] implements an optimization of abstraction and refinement called *lazy abstraction*. Its core idea is that the computationally intensive steps of abstraction and refinement can be optimized by a tighter integration which would enable the reuse of work performed in one iteration in subsequent iterations. Lazy abstraction tightly couples abstraction and refinement by constructing the abstract model on-the-fly,and locally refining the model on-demand. Blast is designed for C.

3.1.2 Concrete Enumerative Software Model Checking

Algorithms for concrete enumerative model checking essentially traverse the graph of program states and transitions using various graph search techniques. The term *concrete* indicates that the techniques represent program states exactly. The term *enumerative* indicates that these methods manipulate individual states of the program, as opposed to *symbolic* techniques (described in Section 3.1.3), which manipulate sets of states.

Depending on whether the model checking algorithm maintains the set of reached states and the set of reachable states in the frontier when traversing the graph, the concrete enumerative model checking algorithms can be divided into two categories: *stateful* or *stateless*. For stateless search, non-determinism is employed to pick execution paths without actually storing the set of visited states. Stateless concrete enumerative model checkers are usually execution-based, meaning using the runtime system of a programming language implementation to implement enumerative state space exploration.

Examples of concrete enumerative software model checkers are:

Verisoft The *Verisoft* [35] tool pioneered the idea of execution-based stateless model checking of software. Verisoft takes as input the composition of several Unix processes that communicate by means of message queues, semaphores and shared variables that are visible to the Verisoft scheduler. The scheduler traps calls made to access the shared resources, and by choosing which process will execute at each trap point, the scheduler is able to exhaustively explore all possible interleavings of the processes executions. Based on Verisoft, the author also published a work of exploring very large state space using genetic algorithms [36].

Java Path Finder *Java Path Finder* (JPF) [43] is an execution-based model checker for Java programs that modifies the Java virtual machine to implement systematic search over different thread schedules. By the support from JVM it is possible to store the visited states, which allows the model checker to use many of the standard reduction-based approaches (e.g., symmetry, partial-order, abstraction) to combating state-explosion. As the visited states can be stored, the model checker can utilize various search-order heuristics without being limited by the requirements of stateless search. One can also use techniques like symbolic execution and abstraction to compute inputs that force the system into states that are different from those previously visited thereby obtaining a high level of coverage. JPF has been used to successfully find subtle errors in several complex Java components developed inside NASA and is available as a highly extensible open-source tool.

CMC *CMC* [59] is an execution based model checker for C programs that explores different executions by controlling schedules at the level of the OS scheduler. CMC stores a hash of each visited state. In order to identify two different states which differ only in irrelevant details like the particular addresses in which heap-allocated structures are located, CMC canonicalizes the states before hashing to avoid re-exploring states that are similar to those previously visited. CMC has been used to find errors in implementations of network protocols and file systems.

3.1.3 Symbolic Software Model Checking

While enumerative techniques capture the essence of software model checking as the exploration of program states and transitions, their use in practice is often hampered by severe state space explosion. This led to research on symbolic algorithms which manipulate representations of sets of states, rather than individual state, and perform state exploration through the symbolic transformation of these representations.

The power of symbolic techniques comes from advances in the performance of constraint solvers that underlie effective symbolic representations, both for propositional logic (SAT-solvers or BDDs) and more recently for combinations of first order theories (SMT-solvers).

Restricted by the underlying constraint solver, symbolic software model checker usually can only handle linear programs. And it may also use concrete execution as a complement of symbolic execution. Examples of know symbolic model checkers are:

DART *DART* [37] refers to directed automated random testing. It automatically tests software by combining three main techniques: (1) automated extraction of the in-

terface of a program with its external environment using static source-code parsing; (2) automatic generation of a test driver for this interface that performs random testing to simulate the most general environment the program can operate in; and (3) dynamic analysis of how the program behaves under random testing and automatic generation of new test inputs to direct systematically the execution along alternative program paths. DART runs the program under test both concretely, executing the actual program with random inputs, and symbolically, calculating constraints on values at memory locations expressed in terms of input parameters. The main strength of DART is thus that testing can be performed completely automatically on any program that compiles — there is no need to write any test driver or harness code. DART is developed by the same author of Verisoft.

CREST *CREST* [50, 11] is an automatic test generation tool for C. It works by inserting instrumentation code (using CIL) into a target program to perform symbolic execution concurrently with the concrete execution. The generated symbolic constraints are solved (using Yices [21]) to generate input that drive the test execution down new, unexplored program paths. CREST uses several heuristic search strategies, including one guided by the control flow graph of the program under test, to speed-up exploring path space.

KLEE *KLEE* is essentially not a model checker but a symbolic virtual machine built on top of the LLVM [70] compiler infrastructure. Its underlying constraint solver is STP [30].

3.2 The GIMPLE Virtual Machine

Building a virtual machine at GCC's GIMPLE representation level takes advantage of the compiler's capability of syntax parsing, type checking, and code optimization. Such a virtual machine requires no program loading and instruction set design. GIMPLE is regarded as the simplest form of GCC's intermediate representation: all expressions take at most two operands. Thus the GIMPLE statements can directly serve as the virtual machine instructions.

Our *GIMPLE Virtual Machine* (GVM) is essentially a process virtual machine [66]: it provides a virtual *application binary interface* (ABI); instruction interpretation is an emulation of native code execution; and every guest process exclusively owns a virtual execution environment. Since GVM's *instruction set architecture* (ISA) is exactly the GIMPLE statements whose operands are named memory locations (or constants), it is apparently register-based. And also GVM is a low-level language virtual machine. Features of high-level language, such as object-orientation and dynamic typing, will not be exhibited at GIMPLE level.

3.2.1 GVM Implementation

We developed our *GIMPLE Virtual Machine* (GVM) as a GCC plug-in, as illustrated in Figure 3.1.

GVM takes two inputs: (1) the control-flow graph set representing all functions in the

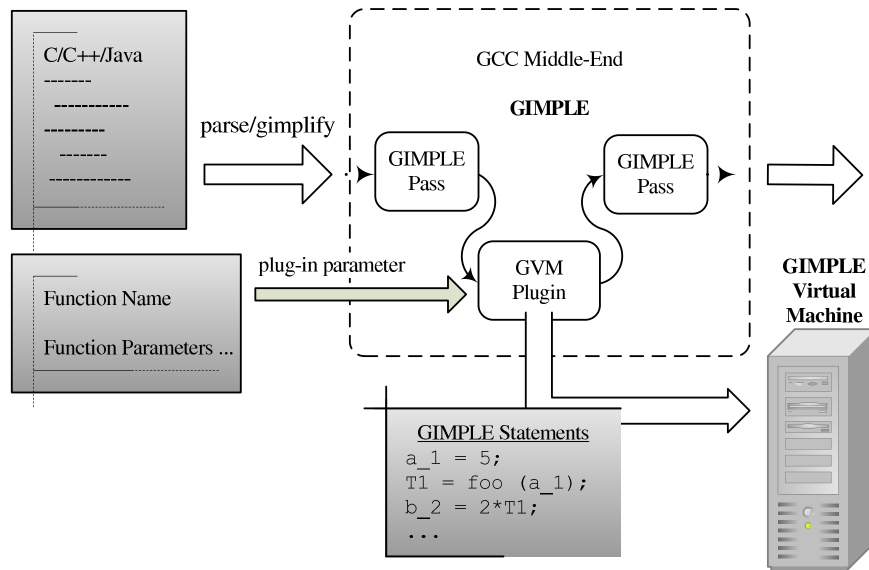


Figure 3.1: The GVM Plugin

compilation unit; (2) user-specified *main* function name and its parameters. As depicted in Figure 3.1, GVM consists of two parts: the GIMPLE instruction reader and the interpreter. GVM keeps a *program counter* indicating the position of next instruction. When GVM is initialized, the program counter is set to the first GIMPLE statement in the successor of *entry* basic block of the *main* function.

Since execution path will not diverge within a basic block, it is safe to feed all statements in a basic block to the interpreter at once. Upon finishing executing all statements in a basic block, the interpreter will calculate where is the next program counter and tell the instruction reader.

GVM excessively employs the idea of *memory segment*. A memory segment is a table of name/value mapping. Values in memory segments are variants: a union of *char*, *int*, *double* and *std::string*, plus a type indicator. In GVM, the global memory is a memory segment, the heap is a memory segment, and all stack frames are memory segments. A stack is a vector of memory segments.

GVM supports function calls. Function calls and returns are as simple as pushing/popping elements in the memory segment vector, along with modifying the program counter.

GVM supports *malloc*, *free*, pointer dereferencing, and point arithmetic. A memory chunk created by *malloc* will be assigned a hidden unique name. A pointer variable's value is the name of the memory chunk in the *heap* memory segment. Pointer dereferencing is to get the value indexed by the pointer variable's value.

GVM is built upon *GLua*. It is essentially an extension of the latter, as shown in Figure 3.2. Like *GLua*, GVM executes user-specified Lua scripts, in which the user has full access to the *GLua* objects. GVM allows to access virtual machine internal state, read/write

interpreted program's private memory space, pause, resume, reset or terminate interpretation, or set callback functions at various break points.

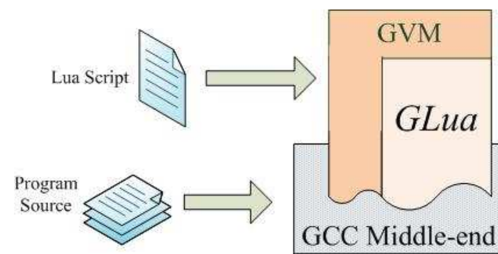


Figure 3.2: Architecture of GVM

GVM has the following shortcomings. It does not support cross-file function call, since GVM is bound to a GCC instance which handles only one compilation unit at a time. It has very limited C library API support. It aims for multi-threading but currently not implemented yet.

Another problem of GVM is variable indexing. Unlike other register-based virtual machines, e.g., the Parrot Virtual Machine, which encodes function local variables into numbers at compilation time and indexes instruction operands by numbers at runtime, GVM has to index local variables by names, which is more expensive.

3.2.2 The GVM API

GVM is designed to be a program concrete-execution engine. Therefore it needs to provide a way to expose its internal status, and let the user to control its execution.

Much like GLua, GVM is implemented as a GCC plug-in and provides a Lua programming interface. The GVM plug-in takes the following arguments:

- script=*path/to/lua/script*: specify a Lua script to execute; default is *gvm.lua*.
- entry=*name*: specify the entry function; default is *main*.
- interactive: run GVM in interactive mode.

Typical command line to invoke GVM is like:

```
gcc4.5 -c -g -O2 test.c -fplugin=libgvm.so
  -fplugin-arg-script=monte-carlo.lua
  -fplugin-arg-ref-entry=main
```

Note if the script is not specified, GVM will not intervene the interpretation and the user will not gain control forever. If the script is specified, GVM will turn to execute a particular function called *exec_gvm* defined in the script, instead of interpreting target program. The user shall trigger GVM to start interpretation in its script.

GVM provides the following Lua API:

set_entry	set the entry function by name.
interpret	trigger GVM to start interpretation.
resume	resume suspended interpretation.
exit	terminate GVM.
reset	reset program counter to entry function and re-initialize GVM.
set_callback_gs(cb)	set callback function `cb` at every gimple statement, `cb` here is a Lua function.
set_callback_bb(cb)	set callback function `cb` at the beginning of every basic block.
set_callback_fn(cb)	set callback function `cb` at every function call.
clear_callback(cb)	unregister callback function `cb`.
globals	get target program's global variables.
n_frames	get the number of stack frames.
frame(i)	get the name of function of the i-th stack frame. i starts from 1.
frame_locals(i)	get the local variables of the i-th stack frame.
dump_frame(i)	print all variables and their values of the i-th stack frame.
read_var(i, var)	query the value of variable *var* in the i-th stack frame; $i == 0$ means to query a global variable.
write_var(i, var, val)	set the value of variable *var* in the i-th stack frame to be *val*; $i == 0$ means a global variable.
get_gs	get current *gimple* statement object, which is about to be interpreted.
get_bb	get current *basicblock* object.
get_fn	get current *function* object.
get_prev_gs	get the *gimple* statement object which was just interpreted.
get_prev_bb	get previous *basicblock* along the execution path within current function CFG.
interactive	go to interactive mode.

As an example, Figure 3.3 shows how to set and clear callback functions. Lua function `trace_static` prints assignments to all static variables. Note that `trace_static` unregisters itself when the `counter` reaches ten, meaning this script prints at most ten lines of screen output.

```
local count;

function trace_static()
    local gs = gvm.get_prev_gs()
    if gs:codename() == 'gimple_assign' then
        local _, lhs = gs:tuple()
        if lhs:static() then
            local lhs_name = tostring(lhs)
            local val = gvm.read_var(0, lhs_name)
            print(lhs_name, '=', val)
            count = count + 1
            if count == 10 then
                gvm.clear_callback(trace_static)
            end
        end
    end
end

function exec_gvm()
    count = 0;
    gvm.set_callback_gs(trace_static)
    gvm.interpret()
end
```

Figure 3.3: A GVM script to print static variable assignments ten times

Chapter 4

GIMPLE-based Monte Carlo Model Checker

GMC^2 [39] is a software model checker for C based on the generic *Monte Carlo model checking* algorithm [40]. It can be seen as an extension of Monte Carlo model checking to the setting of concurrent, procedural programming languages. Monte Carlo model checking is a technique that utilizes the theory of geometric random variables, statistical hypothesis testing, and random sampling of lassos in Büchi automata to realize a one-sided error, randomized algorithm for LTL model checking. To handle the function call/return mechanisms inherent in procedural languages such as C/C++, the version of Monte Carlo model checking implemented in GMC^2 is optimized for pushdown-automaton models. Integrated with GCC, GMC^2 takes the gimlified control-flow graph of target program as input, as well as a C function representing the LTL property of interest. GMC^2 interprets the GIMLPE statements during compilation. The target program can contain concurrency primitives like in Verisoft [35]. In the case of safety properties, the property function is called to check for property violations in the target program. In the case of liveness properties, the property function is called to check if an accepting state of the target program is visited infinitely often, viewing the target program as a succinct representation of a Büchi automaton.

4.1 Monte Carlo Model Checking

Monte Carlo model checking performs random sampling of lassos in a Büchi automaton (BA) to realize a one-sided error, randomized algorithm for LTL model checking. In this section, we provide an overview of this technique. In Section 4.2, we show how to extend this technique to hierarchic Büchi automata (HBA) in the context of software model checking.

4.1.1 Büchi Automata

A *Büchi automaton* $A = (\Sigma, Q, Q_0, \delta, F)$ is a five-tuple where: Σ is a finite *input alphabet*; Q is a finite set of *states*; $Q_0 \subseteq Q$ is the set of *initial states*; $\delta \subseteq Q \times \Sigma \times Q$ is the *transition*

relation; $F \subseteq Q$ is the set of *accepting states*. We assume, without loss of generality, that every state of a BA has at least one outgoing transition, even if this transition is a self-loop.

A sequence $\sigma = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1}, \ldots$, where $s_0 \in Q_0$ and for all $i \geq 0$, $s_i \xrightarrow{a_i} s_{i+1} \in \delta$ is called an *infinite run* of A if the sequence is infinite and a *finite run* otherwise. An infinite run is called *accepting* if there exists an infinite set of indices $J \subseteq \mathbb{N}$, such that for all $i \in J$, $s_i \in F$.

We say that σ is *ultimately periodic* if there exist $i \geq 0$, $l \geq 1$ such that for all $j \geq 0$, $s_{i+j} = s_{i+j \bmod l}$. This means that σ consists of a finite prefix $s_0 \xrightarrow{a_0} \cdots s_{i-1} \xrightarrow{a_{i-1}}$, followed by the "infinite unfolding" of a *cycle* $s_i \xrightarrow{a_i} \cdots \xrightarrow{a_{i+l-1}} s_i$. The cycle is called *simple* if for all $0 \leq j \neq k < l$, $s_{i+j} \neq s_{i+k}$; i.e., the cycle does not visit the same node twice. In the following, we shall refer to such a reachable simple cycle as a *lasso*, and say that a lasso is *accepting* if its simple cycle contains an accepting state.

Let S be a concurrent system, A_S the BA encoding S's state transition graph, and φ an LTL property. Using the tableau method, one can construct a Büchi automaton $A_{\neg\varphi}$ accepting the same language as $\neg\varphi$ [34]. The LTL model-checking problem $A_S \models \varphi$ is then naturally defined in terms of the emptiness problem for $B = A_S \times A_{\neg\varphi}$, which reduces to finding accepting lassos in B [71].

4.1.2 Random Lassos and Hypothesis Testing

Instead of searching the entire state space of B for accepting lassos, we successively generate up to M lassos of B on the fly, by performing random walks in B. The walks are *uniform* in the sense that they are generated by imposing a uniform distribution on the outgoing transitions of the current state along the walk. If the currently generated lasso is accepting, we have found a counter-example to emptiness, and stop.

To determine the number M of lassos we need to generate, we aim to answer, with *confidence* $1-\delta$ and within *error margin* ϵ, the following question: *how many independent lassos do we need to generate until one of them is accepting?* The answer is based on the theory of *geometric random variables* and *statistical hypothesis testing*. Let X be geometric random variable parameterized by the Bernoulli random variable Z (defined below) that takes value 1 with probability p_Z and value 0 with probability $q_Z = 1 - p_Z$. Intuitively, p_Z is the probability that an arbitrary lasso of B is accepting.

The cumulative distribution function of X for N independent trials of Z is: $F(N) = \mathbf{P}[X \leq N] = 1 - (1 - p_Z)^N$. Requiring that $F(N) = 1 - \delta$ yields: $N = \ln(\delta)/\ln(1 - p_Z)$. Given that p_Z is what we wish to determine, we assume for the moment that $p_Z \geq \epsilon$. Replacing p_Z with ϵ yields $M = \ln(\delta)/\ln(1 - \epsilon)$ which is greater than N and therefore $\mathbf{P}[X \leq M] \geq \mathbf{P}[X \leq N] = 1 - \delta$. Summarizing:

$$p_Z \geq \epsilon \quad \Rightarrow \quad \mathbf{P}[X \leq M] \geq 1 - \delta \quad \textbf{where} \quad M = \ln(\delta)/\ln(1 - \epsilon) \tag{4.1}$$

Inequation 4.1 gives us the minimal number of attempts M needed to achieve success with confidence ratio δ, under the assumption that $p_Z \geq \epsilon$. The standard way of discharging such an assumption is to use *statistical hypothesis testing*. Define the *null hypothesis* H_0

as the assumption that $p_Z \geq \epsilon$. Rewriting inequality 4.1 with respect to H_0 we obtain:

$$\mathbf{P}[X \leq M \,|\, H_0] \geq 1 - \delta \tag{4.2}$$

We now perform M trials. If no counterexample is found, i.e., if $X > M$, we reject H_0. This may introduce a type-I error: H_0 may be true even though we did not find a counter-example. However, the probability of making this error is bounded by δ; this is shown in inequality 4.3 which is obtained by taking the complement of $X \leq M$ in inequality 4.2:

$$\mathbf{P}[X > M \,|\, H_0] < \delta \tag{4.3}$$

Because we seek to attain a one-sided error decision procedure, we do not consider type-II errors in our application of hypothesis testing: as soon as we find a counter-example, we stop sampling and decide (with probability 1) that $A \not\models \varphi$.

4.1.3 The Monte Carlo Model Checking Algorithm

For a BA B, define the probability space $(\mathcal{P}(L), \mathbf{P})$, where $L = L_a \cup L_n$ is the set of all lassos of B and L_a and L_n are the sets of all accepting and non-accepting lassos of B, respectively. The probability $\mathbf{P}[\sigma]$ of a lasso $\sigma = s_0 \stackrel{a_0}{\to} \ldots \stackrel{a_{n-1}}{\to} s_n$ is defined inductively as follows: $\mathbf{P}[s_0] = k^{-1}$ if $|Q_0| = k$ and $\mathbf{P}[s_0 \stackrel{a_0}{\to} \ldots \stackrel{a_{n-1}}{\to} s_n] = \mathbf{P}[s_0 \stackrel{a_0}{\to} \ldots \stackrel{a_{n-2}}{\to} s_{n-1}] \cdot \pi[s_{n-1} \stackrel{a_{n-1}}{\to} s_n]$ where $\pi[s \stackrel{a}{\to} s'] = m^{-1}$ if $s \stackrel{a}{\to} s' \in \delta$ and $|\delta(s)| = m$. That $(\mathcal{P}(L), \mathbf{P})$ is actually a probability space is established in [40].

Probability of lassos Consider BA B of Figure 4.1. It contains four lassos, `11`, `1244`, `1231` and `12344`, having probabilities $1/2$, $1/4$, $1/8$ and $1/8$, respectively. Lasso `1231` is accepting.

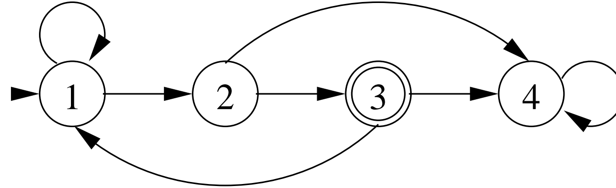


Figure 4.1: Example lasso probability space.

Lasso Bernoulli variable The *random variable* Z associated with the probability space $(\mathcal{P}(L), \mathbf{P})$ of a Büchi automaton B is defined as follows: $p_Z = \mathbf{P}[Z = 1] = \sum_{\lambda_a \in L_a} \mathbf{P}[\lambda_a]$ and $q_Z = \mathbf{P}[Z = 0] = \sum_{\lambda_n \in L_n} \mathbf{P}[\lambda_n]$.

Lassos Bernoulli variable For the Büchi automaton B of Figure 4.1, the lassos Bernoulli variable has associated probabilities $p_Z = 1/8$ and $q_Z = 7/8$.

Having defined Z, X and H_0, we are now ready to present our Monte Carlo decision procedure for emptiness checking of Büchi automata, called $\texttt{MC}^2$ in [40]. `MCM` consists of three statements. The first uses inequation 4.1 to determine the value for M, given parameters

ϵ and δ. The second statement is a for-loop that successively samples up to M lassos by calling the *random lasso* (`rLasso`) routine, described in Section 4.2. If an accepting lasso l is found, `MCM` decides false and returns l as a counter-example. If no accepting lasso is found within M trials, `MCM` decides true, and reports that with probability less than δ,

```
bool × lasso MC² (BA B = (Σ, Q, Q₀, δ, F), float 0 < ε, δ < 1)
{
  M = ln δ / ln(1 − ε);
  for (i = 1; i ≤ M; i++) if (rLasso(B)==(true,l)) return (false,l);
  return (true,nil); /* P[X > M | H₀] < δ */;
}
```

Theorem 4.1.1 ([40]). *Given a Büchi automaton B and parameters ϵ and δ, if `MCM` returns false, then $L(B) \neq \emptyset$. Otherwise, $\mathbf{P}[X > M \mid H_0] < \delta$ where $M = \ln(\delta)/\ln(1-\epsilon)$ and $H_0 \equiv p_Z \geq \epsilon$.*

`MCM` is very efficient in both time and space. The *recurrence diameter* of a Büchi automaton B is the longest loop-free path in B starting from an initial state.

Theorem 4.1.2 ([40]). *Let B be a Büchi automaton, D its recurrence diameter and $M = \ln(\delta)/\ln(1-\epsilon)$. Then `MCM` runs in time $O(MD)$ and uses $O(D)$ space.*

In the worst case, D is exponential in $|S| + |\varphi|$ and thus `MCM`'s does not improve on the space complexity of a typical model checker. In practice, however, one can expect `MCM` to perform much better than this.

4.2 Monte Carlo Software Model Checking

We have implemented a software model checker for GCC based on the generic Monte-Carlo model-checking algorithm of Section 4.1. Our model checker, GMC^2, is applicable to any program written in one of the procedural languages supported by GCC, e.g., C. Call this program the *target program* to be verified. GMC^2 also requires as input a procedure or function, call it the *property function*, representing the LTL property of interest. The target program can contain concurrency primitives similar to those supported by the Verisoft model checker [35]. In the case of safety properties, the property function is called to check for property violations in the target program. In the case of liveness properties, the property function is called to check if an accepting state of the target program is visited infinitely often, viewing the target program as a succinct representation of a Büchi automaton.

GMC^2 operates at the GIMPLE level and assumes that the target program and property function have been compiled into CFGs. Let `P` be the array of CFGs corresponding to the target program, one for each of its functions, and let φ be the CFG for the property function. At the heart of GMC^2 is a CFG interpreter that traverses the CFGs in `P` using GIMPLE's statement iterators and interprets the statements contained in the CFGs according to their semantics. This allows GMC^2 to generate the random lassos of the target program on the fly.

4.2.1 The Main Routine

Due to space considerations, we limit our discussion to the treatment of safety properties. Given an array of `P` of CFGs for the target C program, a CFG φ for the C function encoding a safety property, and parameters ϵ and δ, GMC^2 successively generates at most $\ln(\delta)/\ln(1-\epsilon)$ random lassos of `P`; see Section 4.1. While generating a lasso, φ is called to check whether or not φ is violated in the newly reached program state. If so, GMC^2 stops and returns the counter-example path leading to the violating state. If all states of all sampled executions satisfy φ, GMC^2 stops and reports with confidence greater than $1-\delta$ that it rejects $\mathtt{H}_0 \doteq \mathtt{p}_\mathtt{Z} \geq \epsilon$.

At the heart of GMC^2 is the `rLasso` routine for generating random lassos; `rLasso` conducts a random execution of the CFGs in `P` by interpreting their (possibly concurrent) C statements and checking for property violations.

4.2.2 The `rLasso` Random-Lasso Routine

In order to detect (global) lassos, the (concurrent) program state is stored in a *hash table* `ht` each time a context switch occurs. This is for efficiency purposes: the alternative, less efficient approach would be to store the program state after each statement execution. To ensure the soundness of this approach, we assume that the time between context switches is finite.

```
bool × lasso rLasso() /* global cfg array P, cfg φ */
{
  hashTbl ht = ∅; ready list ready = ∅; bool × state (f,s) = rInit();
  while (s ∉ ht) {
    insert(ht,s); if (¬f) return (true,lasso(ht));
    (f,s) = rNext(s); }
  return (false,lasso(ht));
}
```

4.2.2.1 Hash Table

The `ht` hash table is *optimized* so that common information among global states is shared. It is also *hierarchic* in the sense that all states belonging to a callee are linked to each other so that they can easily be removed from `ht` when the callee returns.

The pseudo-code for the `rLasso` routine is given above. The first line sets `ht` and `ready` to empty, and initializes the violation flag `f` and the current-state variable `s` by calling routine `rInit`. The while-loop searches for (violating) lassos. If the current state `s` is not in `ht`, then it is a new state and is inserted in `ht`. If it is also violating, signaled by `¬f` being true, then a violating lasso was found, which is returned together with the corresponding flag to GMC^2. Otherwise, another random next state is generated by calling `rNext`.

4.2.3 Routines `rInit` and `rNext`

Given a set V of typed variables, a *valuation* (or environment) of V is a mapping of variables in V to their type-correct values. If Γ and Γ' are lists, and σ is a list element, we write concat(Γ, Γ'), append(Γ, σ) and rest(Γ) for the lists obtained by concatenating Γ and Γ', appending σ to Γ, and taking the rest of Γ, respectively, and we write $\Gamma(i)$ for the i-th element of Γ. If Δ is a stack and ϕ a stack element, then we write push(Δ, ϕ), pop(Δ) and $\Delta.\phi$ for pushing ϕ onto the stack, popping the stack, and for the topmost element on the stack, respectively. If s is a statement, i.e., AST of a CFG, then s.a is a child of s.

4.2.3.1 Program State

The state $\Sigma = (\chi, \Gamma)$ of a concurrent C program consists of a valuation χ of the *shared variables* (channels and semaphores) and a list Γ of *process states*, one for each active process. The list is ordered by the order of process creation. The state $\sigma = (\kappa, \delta)$ of a process has two components: the *control state* κ and the *data state* δ. The control state $\kappa = (\gamma, \nu)$ consists of a *function name* γ and a *statement number* ν within γ. The data state $\delta = (\pi, \beta, \Delta)$ consists of a *heap* π, a valuation of *global variables* β and a *frame stack* Δ. Each *frame* $\phi = (\kappa, \rho)$ of Δ contains a return control state κ to the caller CFG and a valuation ρ for the *local variables* of the callee CFG.

4.2.3.2 Routine `rInit`

Execution of P starts in a random state Σ_0 defined as follows. All channels in χ_0 are empty and all semaphores are 0. The process-state list Γ_0 contains only the state σ_0 of the root process. The control state κ_0 of the root process has function main of P in γ_0 and 0 in ν_0.

```
bool × prgState rInit() /* global cfgarray P, cfg φ */
{
  sharedState χ = χ0; procStates Γ = ∅; frameStack Δ = ∅;
  cfgNm γ = main; stmNo ν = 0; controlState κ = (γ,ν);
  lclEnv ρ = ρ0; forall(x ∈ dom(P[γ].param)) ρ[x] = random(P[γ].param.type);
  frame ϕ = (trap,ρ); push(Δ,ϕ); dataState δ = (π0,β0,Δ);
  procState σ = (κ,δ); append(Γ,σ); prgState Σ = (χ,Γ);
  if eval(φ) return (true,Σ) else return (false,Σ);
}
```

The data state δ_0 of the root process consists of the empty heap π_0, valuation β_0 of the global variables, and stack frame Δ_0 with only frame ϕ_0 of main pushed. This frame has a predefined return control state trap (e.g. the stop point) and a valuation ρ_0 for the local variables. The valuation of the formal parameters in ρ_0 is chosen randomly within their corresponding range. Function eval evaluates a CFG in the current state and returns its value.

```
bool × prgState rNext(prgState s)
{
  /* global cfgarray P, hashTbl h, CFG φ, readylist ready */
  int i = random(|ready|); int nxt = ready[i];
  return interpret(s,nxt);
}
```

4.2.3.3 Routine `rNext`

Routine `rNext` randomly selects one of the ready processes and interprets it by calling routine `interpret`, described next. It regains control when `interpret` reaches a concurrency statement, which requires a context switch.

4.2.4 Routine `interpret`

Routine `interpret` traverses the CFGs in `P`, using statement iterators `succ`, `tsucc` and `fsucc`, and interprets each statement according to its semantics. Of particular interest are the process creation and synchronization statements, which force a return whenever a context switch is required, as well as function invocation and return statements, which induce a hierarchic structure on the hash table.

Since `interpret` may generate several states before it returns, it has to check whether property φ is true in all of them. Properties to be checked may also be inserted in a program, as `assert(p)` statements. The interpreter then checks whether predicate `p` is true in the current state and returns with a violation if this is not the case.

The pseudo-code for `interpret` is given below. Its body is an infinite loop, which according to the type of the current statement ν within the current CFG `P`$[\gamma]$, undertakes the actions defining the semantics of the statement. Due to space limitations, we consider a representative susbset of statement types, which does not include heap and pointer manipulation statements.

```
bool × prgState interpret(prgState Σ, int i)
{
  /* global cfgarray P, hashTbl ht, cfg φ, readylist ready */
  channels χ = Σ.χ; procStates Γ = Σ.Γ; procState σ = Γ[i];
  while (true) {
    cfgNm γ = σ.κ.γ; stmtNo ν = σ.κ.ν;
    frameStack Δ = σ.δ.Δ; globalEnv β = σ.δ.β;
    switch (P[γ][ν].type) of
    if:       /* if e goto t */ {
      ν = (eval(P[γ][ν].exp)) ? tsucc(P[γ][ν]) : fsucc(P[γ][ν]); }
    assert:   /* assert(e) */ {
      if (!eval(P[γ][ν].exp)) return (false,Σ); ν = succ(P[γ][ν]); }
    assign:   /* x = rhs */ {
      if (P[γ][ν].rhs.type == expr) { /* rhs == e */
```

```
      ν = succ(P[γ][ν]); (Δ.ρ:β)[P[γ][ν].var] = eval(P[γ][ν].rhs); }
    else if (P[γ][ν].rhs.fnc == toss) { /* rhs == toss(e) */
      ν = succ(P[γ][ν]);
      (Δ.ρ:β)[P[γ][ν].var] = random(eval(P[γ][ν].rhs.exp)); }
    else if (P[γ][ν].rhs.fnc == fork) { /* rhs == fork() */
      ν = succ(P[γ][ν]); (Δ.ρ:β)[P[γ][ν].var] = 0;
      append(Γ,((γ,ν),(π,β,Δ))); (Δ.ρ:β)[P[γ][ν].var] = |Γ|-1; }
    else if (P[γ][ν].rhs.fnc == recv) { /* rhs == recv(c) */
      c = P[γ][ν].rhs.chnl;
      if (empty(χ.c)) {append(χ.c.swait,i); return (true,Σ); }
      ν = succ(P[γ][ν]); (Δ.ρ:β)[P[γ][ν].var] = fst(χ.c.queue));
      rest(χ.c.queue); concat(ready,χ.c.swait); χ.c.swait = ∅; }
    else { /* rhs == f(a) */
      a = eval(P[γ][ν].rhs.act); κ = (γ,ν);
      γ = P[γ][ν].rhs.fnc); ν = 0;
      push(Δ,(κ,ρ_γ,0)); (Δ.ρ)[P[γ].fpar] = a; }
  return:    /* return e */ {
    e = eval(P[γ][ν].exp); (γ,ν) = Δ.κ; popLocal(ht); pop(Δ);
    (Δ.ρ:β)[P[γ][ν].var] = e; }
  send:      /* send(c,e) */ {
    c = P[γ][ν].rhs.chnl;
    if (full(χ.c)) {append(χ.c.rwait,i); return (true,Σ);}
    ν = succ(P[γ][ν]); append(χ.c.queue,eval(P[γ][ν].rhs.exp));
    concat(ready,χ.c.rwait); χ.c.rwait = ∅;
  }
  σ = ((γ,ν),(π,β,Δ)); Γ[i] = σ; Σ = (χ,Γ);
  if (!eval(φ)) return (false,Σ);
 }
}
```

For the sequential intra-procedural group of statements, we discuss the interpretation of `if` and (simple) `assignment`. The former evaluates the predicate in the current state and branches to the appropriate location by modifying ν. The latter evaluates the right-hand side expression and updates the corresponding local environment $\Delta.\rho$ (within the frame on the top of the frame stack) or global environment β, on the location given by the left-hand side variable, accordingly. By writing $\Delta.\rho : \beta$ we mean that both valuations are considered and that $\Delta.\rho$ has precedence over β; i.e., we first search the variable in the local valuation.

The modeling and verification statements presented are `toss` and `assert`. For `toss`, the interpreter first evaluates the argument expression to obtain a value v, and then it randomly generates a number within the range $[0, v]$. The obtained number is assigned to the location given by the left-hand side variable, in either $\Delta.\rho$ or β. For `assert`, the interpreter checks whether the predicate is true in the current state. If this is not the case, it returns `false` and Σ. Otherwise, it continues with the next statement by updating ν.

The inter-procedural statements presented are `call` and `return`. For `call`, a new

frame $\phi = (\kappa, \rho)$ is allocated on top of the frame stack Δ; κ is the current control state; ρ has the local variables of the target function γ initialized accordingly by $\rho_{\gamma,0}$ and the formal parameters evaluated in the current state. Control is then moved to the callee by updating γ and ν accordingly. For `return`, the interpreter does the following. First, it evaluates the return expression in a temporary variable. It then restores the control state from the frame stack, pops the frame stack and erases all the states corresponding to the callee from the hash table. Finally it assigns the temporary variable, to the location given by the variable of the statement pointed to by the control state, in either $\Delta.\rho$ or β.

The concurrency primitives considered so far include process creation, channels and semaphores. For simplicity, we only discuss `fork`, `send` and `recv`. The other are treated in a similar manner. The `fork` statement is handled by creating a new process (state) in Γ which is identical to the current except for the value assigned to the variable on the left-hand side of the fork assignment statement. This is zero for the child process and the index in Γ of the new process for the parent process. The `send` statement is treated as expected. If the channel is full, the process is put in the send wait queue of the corresponding channel, and control is returned to `rNext`. Otherwise, the message expression is evaluated and appended to the channel. Moreover, the process waiting in the receive queue of the channel is awaken, by moving it to the ready list. The `recv` primitive is treated in a similar way.

4.3 Experimental Results

To assess the performance and scalability of GMC^2, we compared it to Verisoft, a popular software model checker from Lucent Technologies [35], on two C benchmarks: dining philosophers and the Needham-Schroeder. Verisoft and GMC^2 were given the same C source files as input, each of which can be downloaded from [69]. We also ran GMC^2 on the TCAS benchmark. All GMC^2 experiments were performed on an Athlon 2600+ MHz processor with 1GB RAM running Linux 2.6.5.

Dining Philosophers. For this classical synchronization problem, we used a faulty *symmetric* but fair variant, where the number of philosophers varied from 4 to 16. The safety property we checked was *deadlock freedom*. Our experimental results are given in Table 4.1. The meaning of the column headings is the following: phi. is the number of philosophers; time is the execution time in mins:secs; ce.len is the length of the counter-example found; states is the number of states Verisoft visited until finding an error; transitions is the number of transitions that Verisoft traversed. The Verisoft experiments were performed on Sun Sparc Ultra-5.10 server running SunOS 5.6. Our experience shows that the Athlon/Linux environment performs approximately 3.4 times faster than the Sparc/SunOS environment.

Needham-Schroeder Protocol. This classic public-key protocol provides mutual authentication for two parties, before they engage in a transaction. In 1995, Lowe first reported a flaw in the protocol [53], by exhibiting an attack involving six message exchanges. Suppose A is the initiator, B is the responder and I is the intruder. Then the attack is as follows: (i) A sends a nonce to I. (ii) I sends same nonce to B. (iii) B sends the above received nonce and its new nonce to I. (iv) I sends the above received message

	GMC^2			Verisoft		
phi.	time	samples	ce.len.	time	states	transitions
4	0:00.07	2	12	0:00.61	16	37
6	0:00.11	4	12	0:16.60	773	1171
8	0:00.78	11	20	2:57.29	5431	8449
10	0:02.17	31	24	10:41	17908	31433
12	0:04.82	24	27	$>$ 2hr	N/A	N/A
14	0:06.22	22	44	$>$ 2hr	N/A	N/A
16	0:11.56	14	32	$>$ 2hr	N/A	N/A

Table 4.1: Deadlock freedom for the symmetric and fair C implementation.

to A. (v) A validates the authenticity of I and sends the second nonce from the message back to I. (vi) I sends this nonce back to B which now also validates I. We checked for the existence of the above attack in a C implementation of the protocol we obtained from Patrice Godefroid, who we greatfully acknowledge. GMC^2 found it in 6 hours and 37 minutes after having checked 10,682,639 lassos.

The same example and implementation was used in [36] to evaluate a novel genetic algorithm. The time usage reported there is 2 hours and 33 minutes to find 3 errors, which is superior to GMC^2 on this benchmark. They also attempted exhaustive and randomized search algorithms on this C program, neither of which could find an error in 8 hours. Their experiments were performed on a Pentium III 700 MHz processor with 256 MB RAM. Unfortunately, the genetic version of Verisoft is not publicly available, and we could not reproduce this result on our own machine. Its superior performance might be explained by the sequential nature of the protocol implementation, which essentially executes only one round of a reactive system. In this round, the system either deadlocks, produces a counterexample or it behaves correctly. Hence, lasso search seems to be less useful in this case than applying genetic heuristics.

TCAS. The *traffic alert and collision avoidance system* (TCAS) is used on board all US commercial aircrafts. It continuously monitors radar information to sense whether a neighboring aircraft could become a threat by getting too close. Such an aircraft is said to be an "intruder", which is entering the protected zone. In this situation TCAS issues a *traffic advisory* and estimates the time remaining until the two aircrafts reach the closest point of approach. Such estimates are used to compute the vertical separation between the two aircraft assuming that the controlled aircraft maintains its current trajectory. Depending on the results obtained, TCAS issues a *resolution advisory* (RA) suggesting the pilot to climb or to descend.

We have verified the RA component from Georgia Techs Siemens suite [64], with respect to the specifications in [18]. Each property is verified by checking the satisfiability of two rules, with specific initial values for variables. The details of these rules, initial conditions on values and the properties, can be found in [18]. Our experimental results are presented in Table 4.2, where the meaning of the column headings is as follows:

property	rule	GMC^2		
		bugs found	time	samples
Safe Advisory Selection	1	No	0.23	1278
	2	Yes	0.03	147
Best Advisory Selection	1	No	0.25	1278
	2	Yes	0.04	206
Avoid unnecessary crossing	1	Yes	0.01	36
	2	Yes	0.03	180
No Crossing Advisory Selection	1	Yes	0.01	27
	2	Yes	0.01	8
Optimal Advisory Selection	1	No	0.23	1278
	2	Yes	0.06	217

Table 4.2: Running time of GMC^2 for TCAS.

property name; corresponding rule number; indication of whether or not GMC^2 found a counter-example; time usage in seconds within which either a counter-example was found or a predefined number of samples was reached; if a counter-example was found, the last colum gives the number of samples taken to that point; otherwise it is the predefined number of samples to be taken: 1,278 corresponding to $\delta = 0.1$ and $\epsilon = 1.8 \times 10^{-3}$.

Chapter 5

Software Model Checking with Dynamic Path Reduction

Dynamic path reduction (DPR) [74] is a general algorithm to prune redundant paths from the state space of a program under verification. It works in the context of the bounded model checking of sequential programs with nondeterministic conditionals. Such programs arise naturally as a byproduct of abstraction during verification as well as being inherent in nondeterministic programming languages. Nondeterministic choice also arises in the modeling of randomized algorithms.

The DPR approach is based on the symbolic analysis of concrete executions. For each explored execution path π that does not reach an abort statement, we repeatedly apply a weakest-precondition computation to accumulate the constraints associated with an infeasible sub-path derived from π by taking the alternative branch to an *if*-statement. We then use a satisfiability modulo theory (SMT) solver (Yices [21]) to learn the minimally unsatisfiable core of these constraints. By further learning the statements in π that are critical to the sub-path's infeasibility as well as the control-flow decisions that must be taken to execute these statements, unexplored paths containing the same unsatisfiable core can be efficiently and dynamically pruned.

5.1 Nondeterministic Conditional and SSA Form

To illustrate the DPR approach to model checking, consider the C program of Figure 5.1(a). Its first two conditional statements are nondeterministic, denoted by placing an asterisk in the condition position. The property we would like to check is whether the program can reach the `abort` statement. The initial values of variables `x, y, z` are 5, 8, 20, respectively. Suppose the first executed path is $\pi = \langle 0, 1, 2, 5, 6, 7, 10, 13 \rangle$. Executing the program along this path avoids the `abort` statement and ends with the `halt` statement. After executing this path, most existing model checkers will backtrack to Line 6 and explore the else-branch in the next execution. Since there are two nondeterministic choices in the program, four executions are required to prove that it cannot be aborted.

Analyzing the execution trace π allows us to learn that the assignments `x = 5` and

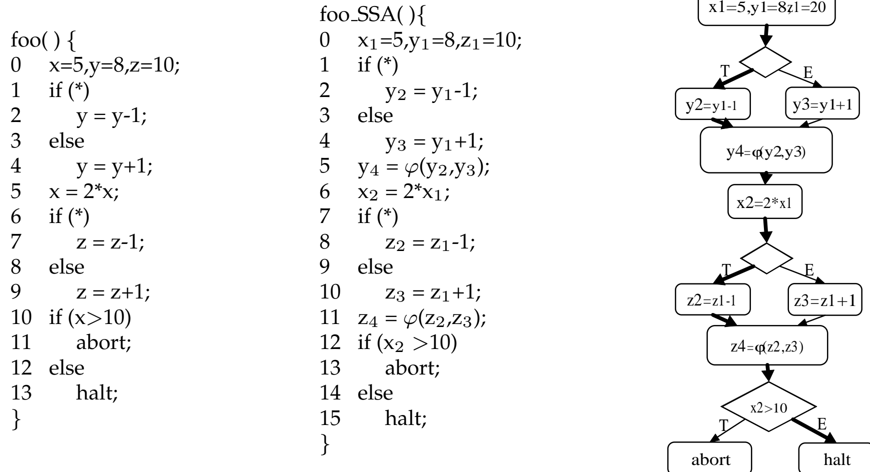


Figure 5.1: A sample C program (left), its SSA form (middle), and SSA graph representation (right).

`x = 2*x` falsify the predicate `x>10` which forces the third conditional to choose its else-branch. We also learn that none of the assignments within the branches of the nondeterministic conditionals can make the predicate true. This allows us to prune all the remaining paths from the search space. A DPR-based model checker would therefore stop after the first execution and report that `abort` is not reachable.

5.2 DPR-Based Model Checking Algorithm

In this section, we present DPR-MC, our bounded model-checking algorithm with dynamic path reduction. Our presentation is carried out in stages, starting with a simplified but transparent version of the algorithm, and with each stage incrementally improving the algorithm's performance. The model-checking algorithm we propose is tunable to run either as a randomized Las Vegas algorithm or as a guided-search algorithm.

As defined formally below, a k-oracle is a bit string of length k representing a sequence of nondeterministic choices a program might make during execution. Suppose we want to perform bounded model checking on a program up to search depth D, such that within this D-bounded search space, each execution path contains at most k nondeterministic choices. In this case, the DPR-MCalgorithm repeats the following three steps until the entire D-bounded search space has been explored: (1) Ask the constraint (SAT) solver to provide a k-oracle. (2) Execute the program on that oracle; stop if an `abort` statement is reached. (3) Use the constraint solver again to prune from the search space all paths that are equivalent to the one just executed.

5.2.1 Global Search Algorithm

The core language we use for analysis is a subset of C, extended with one statement type not present in C: nondeterministic conditionals. To simplify the analysis undertaken by

DPR-MC, we use the *static single assignment* (SSA) representation of programs. For example, the SSA representation of the C program of Figure 5.1 (left) is shown in Figure 5.1 (middle). By indexing (versioning) variables and introducing the so-called φ function at join points, this intermediate representation ensures that each variable is statically assigned only once. We leverage the SSA representation to interface with the satisfiability modulo theory (SMT) solver Yices [21]. In this context, every statement (excepting statements within loops) can be conveniently represented as a predicate. Looping statements are handled by unfolding them up so that every execution path has at most k nondeterministic choices; i.e., a k-oracle is used to resolve the choices. We refer to the SSA representation obtained after such a k-unfolding as the *dynamic single assignment* (DSA) representation.

Suppose the program C to be analyzed has at most k nondeterministic conditionals on every execution path. We call a resolution of these k conditionals a *k-oracle*. Obviously, each k-oracle uniquely determines a finite concrete execution path of C. Let $\mathcal{R}$ be the set of all k-oracles (resolvents) of C. $\mathcal{R}$ can be organized as a decision tree whose paths are k-oracles.

Algorithm 1 DPR-MC(PROGRAM C, INT k)

```
1: R = all k-oracles in C;
2: while R ≠ ∅ do
3:    Remove an oracle R = ⟨r1 r2 ... rk⟩ from R;
4:    ExecuteFollowOracle(R, R, k);
5: end while
6: exit("No bug found up to oracle-depth k");
```

Algorithm 1 is the main loop of our DPR-MCalgorithm. It repeatedly removes a k-oracle R from $\mathcal{R}$ and executes C as guided by R. The algorithm terminates if: (1) execution reaches `abort` within `ExecuteFollowOracle`, indicating that a bug is found; or (2) $\mathcal{R}$ becomes empty after all oracles have been explored, in which case the program is bug-free to oracle-depth k.

Note that Algorithm 1 employs a *global* search strategy. If the oracle removal is random, it corresponds to a randomized Las Vegas algorithm. If the oracle removal is heuristic, it corresponds to a guided-search algorithm. Obviously, the number of oracles is exponential in the depth k of the decision tree $\mathcal{R}$. Hence, the algorithm is unlikely to work for nontrivial programs. We subsequently shall show how to efficiently store the decision tree and how to prune oracles by learning from previous executions.

5.2.2 Weakest Precondition Computation

An execution path $\pi = \langle s_1, s_2, \ldots, s_n \rangle$ is a sequence of program statements, where each s_i is either an assignment or a conditional. We write c_T and c_E for the `then` and `else` branches respectively of a conditional statement c. For brevity, we sometimes refer to an execution path simply as a "path".

Definition Let `x` be a variable, `e` an expression, `c` a Boolean expression, P a predicate,

and P[e/x] the simultaneous substitution of x with e in P. The *weakest precondition* wp(π,P) of π with respect to P is defined inductively as follows:

Assignment: wp(x=e, P) = P[e/x].
Conditional: wp(if(c)$_T$, P) = $P \wedge$ c; wp(if(c)$_E$, P) = $P \wedge \neg$c.
Nondeterminism: wp(if(*)$_T$, P) = wp(if(*)$_E$, P) = P.
Sequence: wp(s_1; s_2, P) = wp(s_1, wp(s_2, P)).

Given an execution path $\pi = \langle s_1, s_2, \ldots, s_n \rangle$, we use $\pi^i = s_i$ to denote the i-th statement of π, and $\pi^{i,j} = s_i, \ldots, s_j$ to denote the segment of π between i and j. Assume now that π^n, the last statement of π, is either c_T or c_E. If $\pi^n = c_T$, then it is impossible for any execution path with prefix $\pi^{1,n-1}$ to take the else-branch at π^n. That is, any execution path that has ρ as a prefix, where $\rho^i = \pi^i (1 \leq i < n)$ and $\rho^n \neq \pi^n$, is infeasible. Because of this, we say that ρ is an *infeasible sub-path*.

Let ρ be an infeasible sub-path of length m where ρ^m is a conditional c. We use $wp(\rho)$ to denote $wp(\rho^{1,m-1}, c)$, and $wp(\rho) = \mathit{false}$ as ρ is infeasible. According to Definition 5.2.2, assuming that ρ contains $t < m$ conditionals in addition to c, we have:

$$wp(\rho) = c' \wedge (c'_1 \wedge c'_2 \ldots \wedge c'_t) = \mathit{false}$$

where c' is ρ^n transformed through transitive variable substitutions, and similarly each c'_l is a transformed deterministic predicate in s_l: $(c_l)_{T/E}$ $(1 \leq l \leq t)$. More formally, given a formula F, we use F' to denote the formula in wp that is transformed from F. The definition is transitive in that both $F' = F(e/v)$ and $F'(e_2/v_2)$ are *transformed formulae* from F.

5.2.3 Learning From Infeasible Sub-paths

Upon encountering a new execution path, the DPR-MCalgorithm collects information about infeasible sub-paths at deterministic predicates by using the weakest precondition computation presented in the previous section. We now analyze the reasons behind the infeasibility of such paths in order to provide useful information for pruning unexplored execution paths.

Since $wp(\rho)$ is unsatisfiable, there must exist an unsatisfiable subformula $wp_{us}(\rho)$ that consists of a subset of clauses $\{c', c'_1, c'_2 \ldots, c'_t\}$.

Definition A *minimally unsatisfiable subformula* of $wp(\rho)$, denoted by $mus(\rho)$, is a subformula of $wp(\rho)$ that becomes satisfiable whenever any of its clauses is removed. A *smallest cardinality MUS* of $wp(\rho)$, denoted by $smus(\rho)$, is an MUS such that for all $mus(\rho)$, $|smus(\rho)| \leq |mus(\rho)|$.

In general, any unexplored paths that contain $mus(\rho)$ are infeasible and can be pruned. $wp(\rho)$ can have one or more MUSes; as a matter of succinctness, we keep track of $smus(\rho)$ for pruning purposes.

Next, we need to identify which statements are responsible for ρ's infeasibility and thus $smus(\rho)$.

Definition A *transforming statement* of a predicate c is an assignment statement `s: v = e` such that variable v appears in the transitive support of c.

For example, the statement `s1:x = y+1` is a transforming statement of the condition $c : (x > 0)$, since $wp(s1, c)$ produces $c' : (y + 1 > 0)$. Statement `s2:y = z*10` is also a transforming statement of c, since $wp(s2, c')$ produces $(z * 10 + 1 > 0)$. During weakest precondition computations, only assignment statements can transform an existing conjunct c into a new conjunct c'. Branching statements can only add new conjuncts to the existing formulae, but cannot transform them. Given an execution path $\pi^{i,j} = s_i, \ldots, s_j$, we use $ts(\pi^{i,j}, c) \subseteq \{s_i, \ldots, s_j\}$ to denote the transforming statements for c.

Definition We define the *explanation* of the infeasibility of ρ to be the set of transforming statements $explain(\rho) = \{s \mid s \in ts(\rho, smus(\rho))\}$.

5.2.4 Pruning Unexplored Paths

In this section we use examples to illustrate how to prune the path search space based on information obtained after learning.

The SSA form of the program of Figure 5.1 is represented graphically to its right. Assume the first explored execution π (highlighted in the figure) takes the `then`-branches at the two nondeterministic `if` statements. We would like to learn from π to prove unexplored paths. In the example, $\pi = \langle x_1 = 5, y_1 = 8, z_1 = 20, *, y_2 = y_1 - 1, y_4 = y_2, x_2 = 2x_1, *, z_2 = z_1 - 1, z_4 = z_2, \neg(x_2 > 10), halt\rangle$, which implies the infeasible sub-path $\rho = \langle x_1 = 5, y_1 = 8, z_1 = 20, *, y_2 = y_1 - 1, y_4 = y_2, x_2 = 2x_1, *, z_2 = z_1 - 1, z_4 = z_2, x_2 > 10\rangle$. According to Definition 5.2.2, we have:

$$wp(\rho) = (x_1 = 5) \wedge (y_1 = 8) \wedge (z_1 = 20) \wedge (true) \wedge (true) \wedge (2x_1 > 10) = false$$

The first three conjuncts come from the initial variable assignments and the next two ($true$) come from the nondeterministic conditionals. The last conjunct $2x_1 > 10$ is due to the deterministic conditional $x_2 > 10$ and the assignment $x_2 = 2x_1$. With a decision procedure, we can decide $smus(\rho) = (2x_1 \leq 10) \wedge (x_1 = 5)$. The explanation for ρ's infeasibility is $explain(\rho) = \{x_1 = 5, x_2 = 2 * x_1\}$. Therefore, we learned that any path containing these two assignments will not satisfy $x_2 > 10$; that is, any execution that contains $explain(\rho)$ can only take the `else`-branch to the conditional $x_2 > 10$. Since all the four possible paths contain $explain(\rho)$, none can reach the `abort` statement, which requires a path through the `then`-branch of the conditional $x_2 > 10$. Therefore, with SMT-based learning, a proof is obtained after only one execution.

A question that naturally arises from the example is what happens if a variable assigned in $explain(\rho)$ is subsequently reassigned? The answer is that if a variable is reassigned at s_i, then s_i will be included in $explain(\rho)$ if it is considered part of the explanation to ρ's infeasibility. For example, consider the program `foo2` which is the same as program `foo` of Figure 5.1 except for an additional assignment $x = x + 1$. The SSA form of `foo2` and its graphical representation is shown in Figure 5.2.4. Due to the new assignment $x = x + 1$ on Line 11, we need to add $x_4 = \varphi(x_2, x_3)$ on Line 12 to decide which

```
foo2_SSA( ){
0   x1=5,y1=8,z1=10;
1   if (*)
2       y2 = y1-1;
3   else
4       y3 = y1+1;
5   y4 = φ(y2,y3);
6   x2 = 2*x1;
7   if (*)
8       z2 = z1-1;
9   else
10      z3 = z1+1;
11      x3= x2+1;
12  x4 = φ(x2,x3);
13  z4 = φ(z2,z3);
14  if (x4 >10)
15      abort;
16  else
17      halt;
}
```

Figure 5.2: A C program in SSA form (left), its graphical representation with a highlighted execution path (middle), and the remaining paths after learning from the highlighted path (right).

version of x to use on Line 14. Assume the first execution, as highlighted in Figure 5.2.4, is $\pi_2 = \langle 0 : x_1 = 5, y_1 = 8, z_1 = 10, 1_T : *, 2 : y_2 = y_1 - 1, 5 : y_4 = y_2, 6 : x_2 = 2x_1, 7_T : *, 8 : z_2 = z_1 - 1, 12 : x_4 = x_2, 13 : z_4 = z_2, 14_E : \neg(x_4 > 10), 15 : halt\rangle$. From this, we can infer the infeasible execution segment $\rho_2 = \langle 0 : x_1 = 5, y_1 = 8, z_1 = 10, 1_T : *, 2 : y_2 = y_1 - 1, 5 : y_4 = y_2, 6 : x_2 = 2x_1, 7_T : *, 8 : z_2 = z_1 - 1, 12 : x_4 = x_2, 13 : z_4 = z_2, 14_T : x_4 > 10\rangle$. Based on an analysis similar to that used in the previous example, we have:

$$wp(\rho_2) = ((x_1 = 5) \wedge (y_1 = 8) \wedge (z_1 = 10) \wedge (true) \wedge (true) \wedge (2x_1 > 10)) = false$$

Although it results in the same $smus(\rho_2) = (2x_1 \leq 10) \wedge (x_1 = 5)$ as $smus(\rho_1)$, the explanation to $smus(\rho_2)$ is different: $explain(\rho_2) = \{x_1 = 5, x_2 = 2x_1, x_4 = x_2\}$. As a result, we can prune fewer paths than in the previous example of Figure 5.1. Figure 5.2.4(right) shows the remaining paths after pruning. Both of the remaining paths take the `else`-branch at the second nondeterministic `if` statement, which will go through $x_4 = x_3$. They cannot be pruned because neither path contains the statement $x_4 = x_2$ of $explain(\rho_2)$.

5.2.5 Path Reduction Algorithm

Algorithm 2 gives the pseudo-code that our DPR-MCalgorithm uses in order to drive the execution of the program under verification along the path determined by a given k-oracle R. If the current statement s_i is an `abort` statement (Lines 3-4), an execution with a bug is found and the algorithm terminates. If s_i is a `halt` statement (Lines 5-6), the current execution is completed. An assignment is performed if s_i is an assignment

Algorithm 2 EXECUTEFOLLOWORACLE(k-ORACLE R, SET $\mathcal{R}$, INT k)

1: $i = j = 0$;
2: **while** true **do**
3: **if** s_i== abort **then**
4: exit("report bug trace $\langle s_1, \ldots, s_i \rangle$");
5: **else if** s_i == halt **then**
6: break;
7: **else if** s_i is an assignment **then**
8: Perform the assignment;
9: **else if** s_i is a nondeterministic conditional **then**
10: if $j == k$ break;
11: follow oracle $R[j]$;
12: $j++$;
13: **else if** s_i is deterministic conditional c with value $true$ **then**
14: LearnToPrune($\langle s_1, \ldots, s_{i-1}, \neg c \rangle$, $\mathcal{R}$) if `then`-branch cannot reach `abort`;
15: **else if** s_i is deterministic conditional c with value $false$ **then**
16: LearnToPrune($\langle s_1, \ldots, s_{i-1}, c \rangle$, $\mathcal{R}$) if `else`-branch cannot reach `abort`;
17: **end if**
18: $i++$;
19: **end while**
20: $\mathcal{R} = \mathcal{R} - \{R\}$;

statement (Lines 7-8). If s_i is a nondeterministic conditional (Lines 9-12), the algorithm checks if the threshold k has already been reached. If not, the algorithm follows the branch specified by oracle $R[j]$ and increase the value of j by 1; otherwise the algorithm breaks from the loop. If s_i is a deterministic conditional c (Lines 13-17), the value of c is computed and the corresponding branch is taken. Meanwhile, SMT-based learning is performed on the branch not taken, as shown in Algorithm 3, if the taken branch cannot reach the `abort` statement. Finally, the completed execution is removed from the unexplored oracle set (Line 20).

The SMT-based learning procedure is given in Algorithm 3. The meaning of, and reason for, each statement, i.e., weakest-precondition computation, SMUS and transforming statements, have been explained in previous sections.

Algorithm 3 LEARNTOPRUNE(INFEASIBLESUBPATH ρ, SET $\mathcal{R}$)

1: $w = wp(\rho)$; // Perform weakest precondition computation
2: $s = smus(w)$ // Compute smallest cardinality MUS
3: $e = explain(s)$; // Obtain transforming statements
4: $\mathcal{R} = prune(\mathcal{R}, e)$; //Remove all oracles in $\mathcal{R}$ that define paths containing e

5.3 Implicit Oracle Enumeration using SAT

One problem with Algorithm 1 is the need to save in $\mathcal{R}$ all k-oracles when model checking commences, the number of which can be exponential in k. In order to avoid this complexity, we show how Boolean formulae can be used to symbolically represent k-oracles.

Our discussion of the symbolic representation of k-oracles will be centered around loop-unrolled control flow graphs (CFGs), which can be viewed as directed acyclic graphs whose nodes are program statements and whose edges represent the control flow among statements. We shall assume that every loop-unrolled CFG has a distinguished root node. The *statement depth* of a loop-unrolled CFG is the maximum number of statements along any complete path from the root. The *oracle depth* of a loop-unrolled CFG is the maximum number of nondeterministic conditional nodes along any complete path from the root.

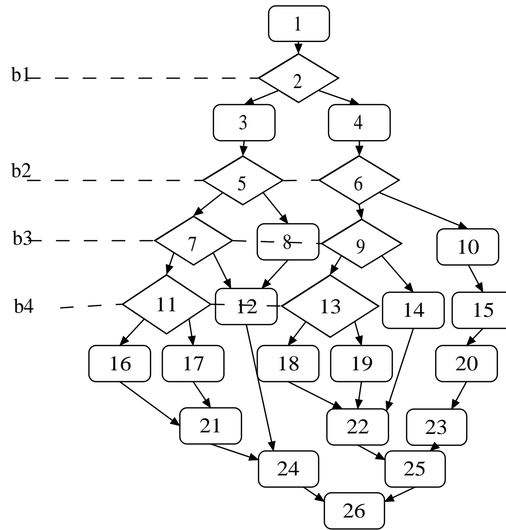


Figure 5.3: An example control flow graph.

Figure 5.3 depicts a typical loop-unrolled CFG, where each node in the CFG has a unique index. Diamond-shaped nodes correspond to nondeterministic conditionals and rectangles are used for other statement types. The statement depth of this CFG is 10. As for its oracle depth, there are 7 nondeterministic conditionals divided into 4 levels (indicated by dotted lines); i.e., its oracle depth is 4.

To encode the choice made along a particular execution path at each level, we introduce the Boolean variables b_1, b_2, b_3 and b_4, with positive literal b_i indicating the `then`-branch and negative literal $\neg b_i$ indicating the `else`-branch. For example, path $\langle 1, 2, 4, 6, 9, 13, 19, 22, 25, 26\rangle$ is captured by $\neg b_1 \wedge b_2 \wedge b_3 \wedge \neg b_4$.

In general, a loop-unrolled CFG will have k levels of nondeterministic conditionals, and we will use k-oracles to explore its path space, with each k-oracle represented as a bit vector of the form $R = \langle b_1, b_2, \ldots, b_k\rangle$. As such, the valuation of Boolean variable b_i indicates an oracle's choice along an execution path at level i, and we call b_i an *oracle choice variable* (OCV). Such considerations lead to a symbolic implementation of the oracle

space in which we use Boolean formulae over $b_i (1 \leq i \leq k)$ to encode k-oracles. For example, the Boolean formula $b_1 b_2 b_3 b_4 + \neg b_1 b_2 \neg b_3$ encodes two paths through the CFG of Figure 5.3: $\langle 1, 2, 3, 5, 7, 11, 16, 21, 24, 26 \rangle$ and $\langle 1, 2, 4, 6, 9, 14, 22, 25, 26 \rangle$. In order to use modern SAT solvers, we maintain such Boolean formulae in conjunctive normal form (CNF).

Algorithm 4 presents a SAT-based implementation of Algorithm 1. It maintains a CNF $\mathcal{B}$ over k OCVs $\{b_1, b_2, \ldots, b_k\}$. Initially, $\mathcal{B}$ is a tautology; the while-loop continues until $\mathcal{B}$ becomes unsatisfiable. Inside the while-loop, we first use a SAT solver to find a k-oracle that is a solution of $\mathcal{B}$, and then perform the program execution determined by the oracle. Algorithm 4 is essentially the same as Algorithm 1 except that: 1) oracle R and set $\mathcal{R}$ are represented symbolically by $\widehat{b}$ and $\mathcal{B}$, respectively; and 2) function calls to `LearnToPrune` (in algorithm `ExecuteFollowOracle`) are replaced by function calls to `SATLearnToPrune`, whose pseudo-code is given in Algorithm 5.

Algorithm 4 DPR-SATMC(PROGRAM C, INT k)

1: Let $b_i (1 \leq i \leq k)$ be k OCV variables, where k is C's oracle depth;
2: CNF $\mathcal{B} = true$;
3: **while** $\mathcal{B}$ is satisfiable **do**
4: Obtain a k-oracle $\widehat{b} = \langle \widehat{b}_1 \widehat{b}_2 \ldots \widehat{b}_k \rangle$ which is a solution of $\mathcal{B}$;
5: ExecuteFollowOracle($\widehat{b}, \mathcal{B}, k$);
6: **end while**
7: exit("No bug found up to oracle-depth k");

Let s be an assignment statement in an infeasible sub-path ρ. We define OCV_s to be the conjunction of those signed (positive or negative) OCVs within whose scope s falls. Also, given ρ's set of transforming statements $explain(\rho) = \{s_1, \ldots, s_t\}$, $OCV(explain(\rho)) = \wedge_{i=1}^{t} OCV_{s_i}$. Note that $OCV(\rho) \neq false$ as all statements in $explain(\rho)$ are along a single path. Further note that $explain(\rho)$ and $OCV(explain(\rho))$ can be simultaneously computed with one traversal of ρ: if a transforming statement s in $explain(\rho)$ is within the scope of a nondeterministic conditional, then the conditional's associated OCV variable is in OCV_s.

To illustrate these concepts, assume $explain(\rho) = \{1, 4, 22\}$ in the loop-unrolled CFG of Figure 5.3. Since node 1 can be reached from root node without going through any conditional branches, $OCV_1 = true$. Node 4 on the other hand is within the scope of the `else`-branch of nondeterministic conditional node 2 and thus $OCV_4 = \neg b_1$. Similarly, $OCV_{22} = \neg b_1 b_2$. Notice that the scopes of b_3 and b_4 close prior to node 22 and are therefore not included in OCV_{22}. Finally, $OCV(\rho) = OCV_1 \wedge OCV_4 \wedge OCV_{22} = \neg b_1 b_2$.

Algorithm 5 is our SAT-based implementation of Algorithm 3. $OCV(e)$ determines the set of paths containing all statements in $explain(\rho)$, and thus all paths that can be pruned. Let $OCV(e) = l_1 \wedge l_2 \wedge \ldots \wedge l_m$, where l_i is a literal denoting b_i or $\neg b_i$. Adding $\neg OCV(e) = \neg l_1 \vee \neg l_2 \vee \ldots \vee \neg l_m$ to the CNF formula $\mathcal{B}$ will prevent the SAT solver from returning any solution (k-oracle) that has been pruned. We refer to $\neg OCV(e)$ as a *conflict clause*.

Algorithm 5 SATLEARNTOPRUNE(INFEASIBLESUBPATH ρ, CNF $\mathcal{B}$)

1: $w = wp(\rho)$; // Perform weakest precondition computation
2: $s = smus(w)$; // Compute smallest cardinality MUS
3: $e = explain(s)$; // Obtain transforming statements
4: $b = OCV(e)$; //Obtain OCV on which e depends
5: let $b = l_1 \wedge l_2 \wedge \ldots \wedge l_m$ where l_i is a literal for b_i or $\neg b_i$;
6: $\mathcal{B} = \mathcal{B} \wedge (\neg l_1 \vee \neg l_2 \vee \ldots \vee \neg l_m)$;

Note that the added conflict clause may prune multiple oracles, including the one just executed. Further note that when exploring a path by virtue of a given k-oracle, not all OCVs may be executed. For example, if the k-oracle in question is $b_1 \neg b_2 b_3 b_4$ in Figure 5.3, the actual execution path terminates after $\neg b_2$. In this case, the added conflict clause is $(\neg b_1 \vee b_2)$ instead of $(\neg b_1 \vee b_2 \vee \neg b_3 \vee \neg b_4)$.

To further illustrate Algorithms 4 and 5, consider once again the program of Figure 5.1. Suppose that the first path π_1 to be explored is the highlighted one in the figure. In this case, the infeasible sub-path ρ_1 to be considered is the same as π_1 except that the then-branch of the final deterministic conditional is taken leading to the `abort` statement. We then have that $smus(\rho_1) = (2x_1 \leq 10) \wedge (x_1 = 5)$ and the explanation for ρ_1's infeasibility is $explain(\rho_1) = \{x_1 = 5, x_2 = 2{*}x_1\}$. Moreover, $OCV(e_1) = true$ as neither of the statements in $e_1 = explain(\rho)$ are in the scope of a nondeterministic conditional. The resulting conflict clause is $false$ and adding (conjoining) it to $\mathcal{B}$ renders $\mathcal{B}$ unsatisfiable; i.e., all remaining paths can be pruned.

Consider next the program of Figure 5.2.4 and its highlighted execution π_2. As explained in Section 5.2, $smus(\rho_2) = smus(\rho_1)$, where ρ_2 is the infeasible sub-path corresponding to π_2. However, the explanation for $smus(\rho_2)$, $explain(\rho_2) = \{x_1 = 5, x_2 = 2x_1, x_4 = x_2\}$, is different. Furthermore, $OCV(e_2) = b_2$, where $e_2 = explain(\rho_2)$, since the assignment $x_4 = x_2$ is within the scope of the `then`-branch of the second nondeterministic conditional. We thus add conflict clause $\neg b_2$ to $\mathcal{B}$, which results in the two remaining paths after pruning illustrated in Figure 5.2.4(right), both of which take the `else`-branch at the second nondeterministic conditional.

Theorem 5.3.1. *(Soundness and Completeness). Let C be a CFG that is loop-unrolled to statement depth D, and let ϕ be a safety property, the violation of which is represented by an* `abort` *statement in C. Then algorithm DPR-MCreports that the* `abort` *statement is reachable if and only if C violates ϕ within statement depth D.*

5.4 Experimental Evaluation

In order to assess the effectiveness of the DPR technique in the context of bounded model checking, we conducted several case studies involving well-known randomized algorithms. All results were obtained on a PC with a 3 GHz Intel Duo-Core processor with 4 GB of RAM running Fedora Core 7. We set a time limit of 500 seconds for each program execution.

In the first case study, we implemented a randomized algorithm for the MAX-3SAT

vars	clauses	paths	explored	pruned	time w DPR(s)	time w/o DPR(s)
9	349	512	44	468	5.44	3.86
10	488	1024	264	760	13.77	7.61
11	660	2048	140	1908	9.67	15.58
12	867	4096	261	3835	14.53	30.59
13	1114	8192	1038	7154	49.61	70.10
14	1404	16384	965	15419	54.05	150.32
15	1740	32768	337	32431	25.58	300.80
16	2125	65536	2369	63167	49.32	Timeout
17	2564	131072	2024	129048	184.91	Timeout
18	3060	262144	1344	260800	175.34	Timeout
19	3615	524288	669	523619	110.14	Timeout

Table 5.1: Bounded model checking with DPR of Randomized MAX-3SAT

problem. Given a 3-CNF formula (i.e., with at most 3 variables per clause), MAX-3SAT finds an assignment that satisfies the largest number of clauses. Obtaining an exact solution to MAX-3SAT is NP-hard. A randomized approximation algorithm independently sets each variable to 1 with probability 0.5 and to 0 with probability 0.5, and the number of satisfied clauses is then determined. In our implementation, we inserted an unreachable `abort` statement; as such, all paths have to be explored to prove the absence of any reachable `abort` statement. Table 5.1 presents our experimental results for the randomized MAX-3SAT algorithm. Each row of the table contains the data for a randomly generated CNF instance, with Columns 1 and 2 listing the number of variables and clauses in the instance, respectively. Columns 3-5 respectively show the total number of execution paths, the number explored paths, and the number of pruned paths, with the sum of the latter two equal to the former. Finally, Columns 6-7 present the run time with DPR and the run time of executing all paths without DPR. From these results, we can observe that DPR is able to prune a significant number of the possible execution paths. Furthermore, the larger the CNF instance, the more effective dynamic path reduction is.

Benchmark			With DPR			Without DPR	
length	valid	paths	explored	pruned	time(s)	explored	time(s)
13	yes	8192	22	8166	0.707	2741	0.085
14	yes	16384	28	16356	0.845	10963	0.144
18	yes	262144	39	262105	2.312	175403	7.285
20	yes	1048576	29	1048542	4.183	350806	6.699
21	yes	2097152	26	2097097	4.202	175403	4.339
11	no	2048	15	2033	1.69	2048	10.027
13	no	4096	13	4083	0.52	4096	16.607
14	no	16384	8	16376	0.84	16384	53.358
20	no	1048576	28	1048548	3.32	-	Timeout

Table 5.2: Bounded model checking with DPR of NFA for floating-point expressions.

In our second case study, we implemented an algorithm that uses a Nondeterministic Finite Automaton (NFA) to recognize regular expressions for floating-point values of the form $[+]?[0-9]+\backslash.[0-9]+$. We encoded the accept state as an `abort` statement and verified whether it is reachable. Table 5.2 contains our experimental results on nine input sentences, among which five are valid floating-point expressions and four are not. Columns 1 and 2 give the length of the input and whether or not it is accepted by the NFA. Column 3 lists the total number of execution paths Columns 4-6 contain the results using DPR, i.e. the number explored paths, the number of pruned paths and the run time. Columns 7 and 8 list the number of explored paths and run time without DPR. Note that in the case of a valid floating-point expression, the number of explored paths without DPR may not be the same as the number of total paths since the accept state is reached before exploring the remaining paths. As in the MAX-3SAT case study, we can again observe a very high percentage of pruned paths, a percentage that grows with the instance size.

Chapter 6

Software Monitoring with Controllable Overhead

The *controller design problem* is the problem of devising a controller Q that regulates the input v to a process P (henceforth referred to as the *plant*) in such a way that P's output y adheres to a *reference input* x with good dynamic response and small error; see the architecture shown in Figure 6.1.

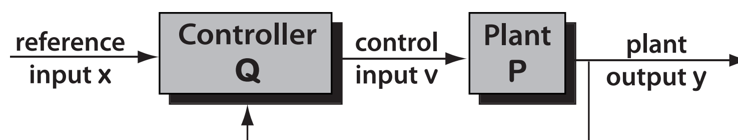


Figure 6.1: Plant (P) and Controller (Q) architecture.

Runtime monitoring with controllable overhead can beneficially be stated as a controller design problem: The controller is a feedback controller that observes the monitoring overhead, the plant comprises the runtime monitor and the application software, and the reference input x to the controller is given by the user-specified *target overhead* o_t. This structure is depicted in Figure 6.2. To ensure that the plant is *controllable*, one typically *instruments* the application and the monitor so that they emit *events* of interest to the controller. The controller catches these events, and controls the plant by *enabling* or *disabling* monitoring and event signaling. Hence, the plant can be regarded as a *discrete event process*.

In runtime monitoring, overhead is the measure of how much longer a program takes to execute because of monitoring. If an unmodified and unmonitored program executes in time R and executes in total time $R+M$ with monitoring, we say that the monitoring has overhead $M\,/\,R$.

Instead of controlling overhead directly, it is more convenient to write the SMCO control laws in terms of *monitoring percentage*: the percentage of execution time spent monitoring events, which is equal to $M/(R+M)$. Monitoring percentage m is related to the traditional definition of overhead o by the equation $m = o\,/\,(1+o)$. The *user-*

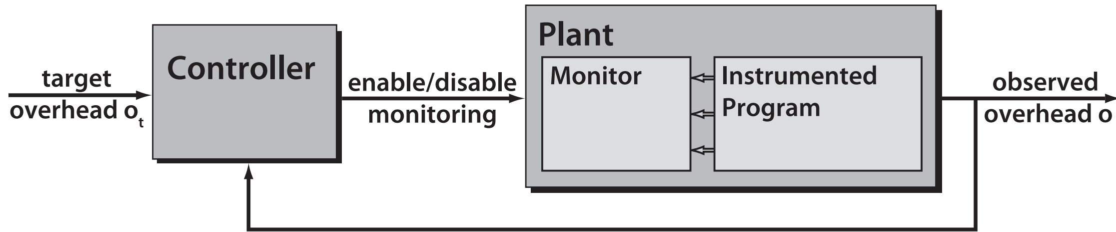


Figure 6.2: Generic SMCO Architecture.

specified target monitoring percentage (UTMP) m_t is derived from o_t in a similar manner; i.e., $m_t = o_t \,/\, (1 + o_t)$.

The classical theory of digital control [29] assumes that the plant and the controller are linear systems. This assumption allows one to semi-automatically design the controller by applying a rich set of design and optimization techniques, such as the Z-transform, fast Fourier transform, root-locus analysis, frequency response analysis, proportional-integrative-derivative (PID) control, and state-space optimal design. For nonlinear systems, however, these techniques are not directly applicable, and various linearization and adaptation techniques must be applied as pre- and post-processing, respectively.

The problem we are considering is nonlinear, because of the enabling and disabling of interrupts. Intuitively, the interrupt signal is multiplied by a control signal which is 1 when interrupts are enabled and 0 otherwise. Although linearization is one possible approach for this kind of nonlinear system, automata theory suggests a better approach, recasting the controller design (synthesis) problem as one of *supervisory controller design* [63, 1].

The main idea of supervisory control we exploit to enable and disable interrupts is the synchronization inherent in the *parallel composition* of state machines. In this setting, the plant P is a state machine, the desired outcome (tracking the reference input) is a language L, and the controller design problem is that of designing a controller Q, which is also a state machine, such that the language $L(Q\|P)$ of the composition of Q and P is included in L. This problem is decidable for *finite* state machines [63, 1].

Monitoring percentage depends on the timing (frequency) of events and the monitor's per-event processing time. The specification language L therefore consists of *timed words* $a_1, t_1, \ldots, a_l, t_l$ where each a_i is an (access) event that occurs at time t_i. Consequently, the state machines used to model P and Q must also include a notion of time. Previous work has shown that supervisory control is decidable for *timed automata* [3, 73] and for *timed transition models* [51].

Modeling overhead control requires however, the use of more expressive, extended timed automata (see Section 6.2), and for such automata decidability is lost. The lack of decidability means that a controller cannot be automatically synthesized. This however, does not diminish the usefulness of control theory. On the contrary, this theory becomes an indispensable guide in the design of a controller that satisfies a set of constraints. In particular, we use control theory to develop a novel combination of supervisory and PID

control. As in classical PID control, the error from a given setpoint (and the integral and derivative of the error) is employed to control the plant. In contrast to classical PID control, the computation of the error and its associated control happens in our framework on an event basis, instead of a fixed, time-step basis.

To develop this approach, we must reconcile the seemingly incompatible worlds of event- and time-based systems. In the time-based world of discrete time-invariant systems, the input and the output signals are assumed to be known and available at every multiple of a fixed sampling interval Δt. Proportional control (P) continually sets the current control input $v(n)$ as proportional to the current error $e(n)$ according to the equation $v(n) = ke(n)$, where n stands for $n\Delta t$ and $e(n) = y(n) - x(n)$ (recall that v, x, and y are depicted in Figure 6.1). Integrative control (I) sums the previous and current error and sets the control input to $v(n) = k \sum_{i=0}^{n} e(n)$.

In contrast, in the event-based world, time information is usually abstracted away, and the relation to the time-based world, where controller design is typically done, is lost. However, in our setting the automata are timed, that is, they contain clocks, ticking at a fixed clock interval Δt. Thus, events can be assumed to occur at multiples of Δt, too. Of course, communication is event based, but all the necessary information to compute the proper control value $v(t)$ is available, whenever an event is thrown at a given time t by the plant.

We present two controller designs with different trade-offs and correspondingly different architectures. Our *global controller* is a single controller responsible for all objects of interest in the monitored software; for example, these objects may be functions or memory allocations, depending on the type of monitoring being performed. The global controller features relatively simple control logic and hence is very efficient: its calculations add little to the observed overhead. It does not, however, attempt to be fair in terms of monitoring infrequently occurring events. Our *cascade controller*, in contrast, is designed with fairness in mind, as the composition of a *primary controller* and a set of *secondary controllers*, one for each monitored plant.

Both controller architectures temporarily disable interrupts to control overhead. One must therefore consider the impact of events missed during periods of non-monitoring on the monitoring results. The two applications of SMCO we consider are integer range analysis and the detection of under-utilized memory. For under-utilized memory detection, when an event is thrown, we are certain that the corresponding object is not stale. We can therefore ignore interrupts for a definite interval of time, without compromising soundness and at the same time lowering the monitoring percentage.

Similarly, for integer range analysis, two updates to an integer variable that are close to each other in time (e.g., consecutive increments to a loop variable) are often near each other in value as well. Hence, processing the interrupt for the first update and ignoring the second, is often sufficient to accurately determine the variable's range, while also lowering monitoring percentage. For example, in the benchmarking experiments described in Section 7.3, we achieve high accuracy (typically 90% or better) in our integer range analysis with a target overhead of just 10%.

6.1 Target Specification

The target specification for a single controlled plant is given as a timed language L, containing timed words of the form $a_1, t_1, \ldots, a_l, t_l$, where a_i is an event and t_i is the time at which a_i has occurred. Each plant has a local target monitoring percentage m_{lt}, which is effectively that plant's portion of the UTMP m_t. Specifically, L contains timed words $a_1, t_1, \ldots, a_l, t_l$ that satisfy the following conditions:

1. The average monitoring percentage $\overline{m} = (l\,p_a)\,/\,(t_l - t_1)$ is such that $\overline{m} \leq m_{lt}$, where p_a is the average time taken by the monitor and controller to process an event.
2. If the strict inequality $\overline{m} < m_{lt}$ holds, then the monitoring-percentage undershoot is due to time intervals with low activity during which all events are monitored.

The first condition bounds only the *mean monitoring percentage* $\overline{m}$ within a timed word $w \in L$. Hence, various policies for handling monitoring percentage, and thus enabling and disabling interrupts, are allowed. The second condition is a *best-effort* condition which guarantees that if the target monitoring percentage is not reached, this is only because the plant does not throw enough interrupts. As our benchmarking results of Section 7.3 demonstrate, we designed the SMCO global and cascade controllers (described in Section 6.3) to satisfy these conditions.

When considering the target specification language L and the associated mean monitoring percentage $\overline{m}$, it is important to distinguish plants in which all interrupts can be disabled (as in Figure 6.3) from the other (as in Figure 6.4). Hardware-based execution platforms (e.g., CPU and MMU) and virtual machines such as the JVM belong to the former category. (The JVM supports disabling of software-based interrupts through just-in-time compilation.)

Software plants written in C, however, typically belong to the latter category, because code inserted during instrumentation is not removed at run-time. In particular, as discussed in Section 6.2.2, when function calls are instrumented, the instrumented program always throws function-call interrupts a_{fc}. Consequently, for such plants, in addition to $\overline{m}$, there is also an unavoidable *base monitoring percentage* $m_b = k\,p_{fc}$, where k is the number of function calls.

6.2 Plant Models

This section specifies the behavior of the above plant types in terms of *extended timed automata* (introduced below). For illustration purpose, each *hardware plant* is controlled by a secondary controller, and the unique *software plant* is controlled by the global controller.

6.2.1 Hardware Plant

Timed automata (TA) [3] are finite-state automata extended with a set of clocks, whose values are positive reals. Clock predicates on transitions are used to model timing behavior, while clock predicates appearing within locations (states) are used to enforce progress properties. Clocks may be reset by transitions. *Extended TA* are TA with local variables and a more expressive clock predicate/assignment language.

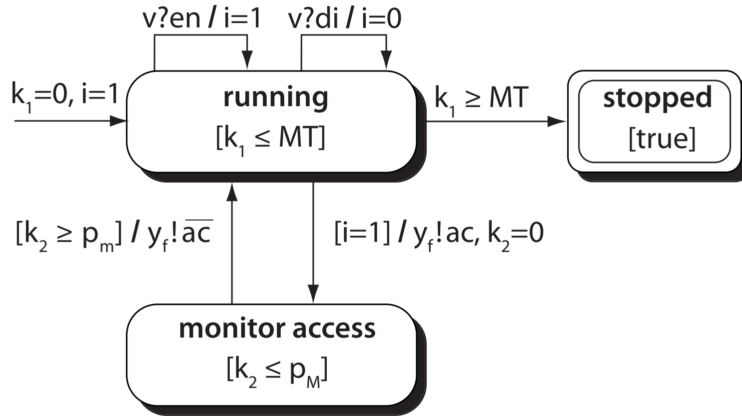


Figure 6.3: Automaton for the hardware plant P of one monitored object.

The hardware plant P is modeled by the extended TA in Figure 6.3. Its alphabet consists of input and output events. The clock predicates labeling its locations and transitions are of the form $k \sim c$, where k is a *clock*, c is a natural number or variable, and $\sim$ is one of $<$, $\leq$, $=$, $\geq$, and $>$. For example, the predicate $k_1 \leq MT$ labeling P's state *running* is a clock constraint, where k_1 is a clock and MT is the maximum-monitoring-time parameter discussed below.

Transition labels are of the form `[guard] In / Cmd`, where `guard` is a predicate over P's variables; `In` is a sequence of input events of the form $v?e$ denoting the receipt of value e (written as a pattern) on channel v; and `Cmd` is a sequence of output and assignment events. An output event is of the form $y!a$ denoting the sending of value a on channel y; an assignment event is simply an assignment of a value to a local variable of the automaton. All fields in a transition label are optional. The use of $?$ and $!$ to denote input and output events is standard, and first appeared in Hoare's paper on CSP [47].

A transition is *enabled* when its guard is true and the specified input events (if any) have arrived. A transition is not *forced* to be taken unless letting time flow would violate the condition (invariant) labeling the current location. For example, the transition out of the state *monitor access* in Figure 6.3 is enabled as soon as $k_2 \geq p_m$, but not forced until $k_2 \geq p_M$. The choice is nondeterministic, and allows to succinctly capture any transition in the interval $[p_m, p_M]$. This is a classic way of avoiding overspecification.

P has an input channel v where it may receive *enable* and *disable* commands, denoted en and di, respectively, and an output channel y_f where it may send begin and end of *access* messages, denoted ac and $\overline{ac}$, respectively. Upon receipt of di, interrupt bit i is set to 0, which prevents the plant from sending further messages. Upon receipt of en, i is set to 1, which allows the plant to send an access message ac at arbitrary moments in time. Once an access message is sent, P resets the clock variable k_2 and transitions to a new state. At any time in the interval $[p_m, p_M]$, P can leave this state and send an end of access message $y_f!\overline{ac}$ to the controller.

P terminates when the maximum monitoring time MT, a parameter of the model, is reached, i.e., when clock k_1 reaches value MT. Initially, $i = 1$ and $k_1 = 0$.

A running program can have multiple hardware plants, with each plant a source

of monitored events. For example, a program running under our NAP detection tool for finding under-utilized memory has one hardware plant for each monitored memory region. The NAP detector's controller can individually enable or disable interrupts for each hardware plant.

6.2.2 Software Plant

In a software plant P, the application program is instrumented to handle, together with the monitor, the interrupt logic readily available to hardware plants (see Figure 6.4).

A software plant represents a single function that can run with interrupts enabled or disabled. In practice, the function toggles interrupts by choosing between two copies of the function body each time it is called: one copy that is instrumented to send event interrupts and one that is left unmodified.

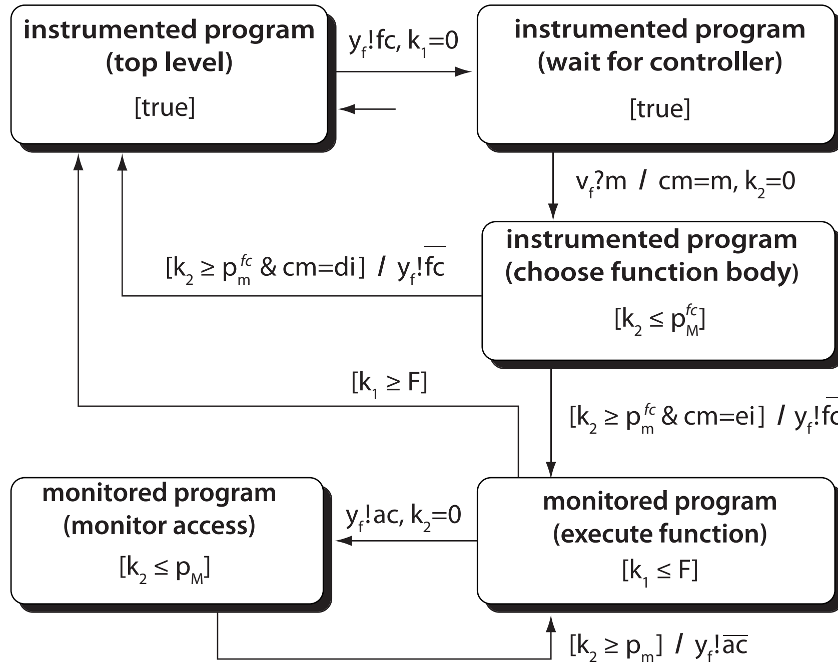


Figure 6.4: Automaton for the software plant P of all monitored objects.

Whenever a function call happens at the *top level* state of P, the instrumented program resets the clock variable k_1, sends the message fc on y_f to the controller and *waits* for its response. If the response on v_f is di, indicating that interrupts are disabled, then the unmonitored version of the function body is called. This is captured in P by returning to the top level state at any time in the interval $[p_m^{fc}, p_M^{fc}]$. This interval represents the time required to implement the call logic.

If the response on v_f is ei, indicating that interrupts are enabled, then the monitored version of the function body is called. This is captured in P by transitioning to the state *execute function* within the same interval $[p_m^{fc}, p_M^{fc}]$.

Within the monitored function body, the monitor may send on y_f a begin of access

event ac to the controller, whenever a variable is accessed, and transition to the state *monitor access*. The time spent by monitoring this access is expressed with a transition back to *execute function* that happens at any time in the interval $[p_m, p_M]$. This transition sends an end of access message $\overline{ac}$ on y_f to the controller.

P terminates processing function f when the maximum monitoring time F, a parameter of the model, is reached; that is, when clock $k_1 \geq F$.

6.3 Controllers

6.3.1 Global Controller

Integrative control uses previous behavior of the plant to control feedback. Integrative control has the advantage that it has good overall statistical performance for plants with consistent behavior and is relatively immune to *hysteresis*, in which periodicity in the output of the plant produces periodic, out-of-phase responses in the controller. Conversely, proportional control is highly responsive to changes in the plant's behavior, which makes it appropriate for long-running plants that exhibit change in behavior over time.

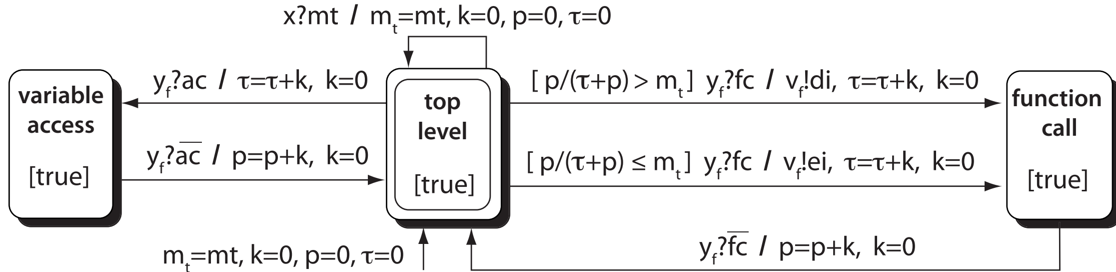


Figure 6.5: Automaton for global controller.

We have implemented an integral-like global controller for plants with consistent behavior. Architecturally, the global controller is in a feedback loop with a single plant representing all objects of interest to the runtime monitor. The architecture of the global controller is thus exactly that of Figure 6.2, which is identical to the classical plant-controller architecture of Figure 6.1, except that in Figure 6.2, the plant is decomposed into the runtime monitor and the software it is monitoring.

In presenting the extended TA for the global controller, we assume it is in a feedback loop with a software-oriented plant whose behavior is given by the extended TA of Figure 6.4. This is done without loss of generality, as the global controller's state machine is simpler in the case of a hardware-oriented plant. The global controller thus assumes the plant emits events of two types: *function-call events* and *access events*, where the former corresponds to the plant having entered a C function, and the latter corresponds to updates to integer variables, in the case of integer range analysis.

The global controller's automaton is given in Figure 6.5 and consists of three locations: *top level*, the *function-call* processing location, and the *variable-access* processing location. Besides the UTMP m_t, the automaton for the global controller makes use of the

following variables: clock variable k, a running total τ of the program's execution time, and a running total p of the instrumented program's observed processing time p. Variable τ keeps the time the controller spent in total (over repeated visits) in its top-level location, whereas variable p keeps the time the controller spent in total in its function-call and access-event processing locations. Hence, at every moment in time, the observed overhead is $o = p / \tau$ and the observed monitoring percentage is $m = p / (\tau + p)$.

In the top-level location, the controller can receive the UTMP on channel x. The controller transitions from the top-level to the function-call processing location whenever a function-call event occurs. In particular, when function f is called, the plant emits an fc signal to the controller along y_f (regardless of whether access event interrupts are enabled for f), transitioning the controller to the function-call processing location along one of two edges. If the observed monitoring percentage for the entire program execution is above the UTMP m_t, the edge taken sends the di signal along v_f to disable monitoring of interrupts for that function call. Otherwise, the edge taken enables these interrupts. Thus, the global controller decides to enable/disable monitoring on a per function-call basis. Moreover, since the enable/disable decision depends on the sign of the *cumulative* error $e = m - m_t$, the controller is *integrative*.

The time taken in the function-call processing location, which the controller determines by reading clock k's value upon receipt of an $\overline{fc}$ signal from the plant, is considered monitoring time; the transition back to the initial state thus adds this time to the total monitoring time p.

The controller transitions from the top-level to the variable-access processing location whenever a function f sends the controller an access event ac and interrupts are enabled for f. Upon receipt of an $\overline{ac}$ event signaling the completion of event processing in the plant, the controller measures the time it spent in its variable-access location, and adds this quantity to p. To keep track of the plant's total execution time τ, each of the global controller's transitions exiting the initial location updates τ with the time spent in the top-level location.

Note that all of the global controller's transitions are event-triggered, as opposed to time-triggered, as it interacts asynchronously with the plant. This aspect of the controller model reflects the discrete-event-based nature of our PID controllers.

6.3.2 Cascade Controller

As per the discussion of monitoring-percentage undershoot in Section 6.1, some plants (functions or objects in a C program) might *not* generate interrupts at a high rate, and therefore might not make use of the target monitoring percentage available to them. In such situations it is desirable to redistribute such unused UTMP to more active plants, which are more likely to make use of this monitoring percentage. Moreover, this redistribution of the unused UTMP should be performed fairly, so that less-active plants are not ignored.

This is the rationale for the SMCO cascade controller (see Figure 6.6), which consists of a set of secondary controllers Q_i, each of which directly control a single plant P_i, and a

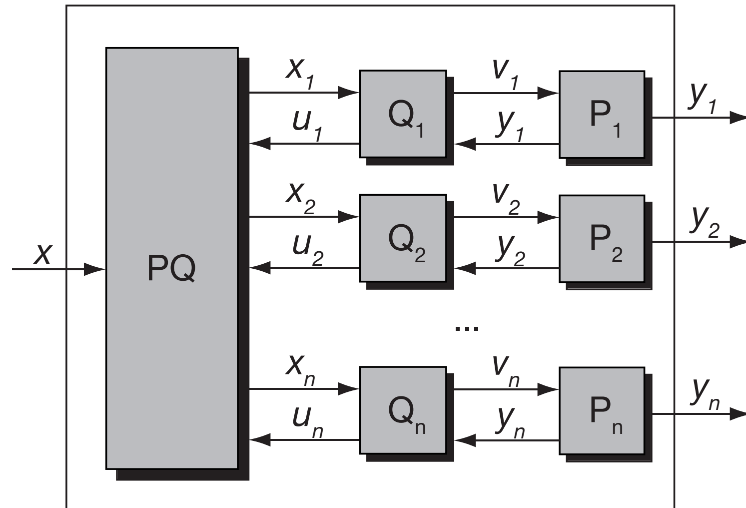


Figure 6.6: Overall cascade control architecture.

primary controller PQ that controls the reference inputs x_i to the secondary controllers. Thus, in the case of cascade control, each monitored plant has its own secondary controller that enables and disables its interrupts. The primary controller adjusts the *local target monitoring percentage* (LTMP) m_{lt} for the secondary controllers.

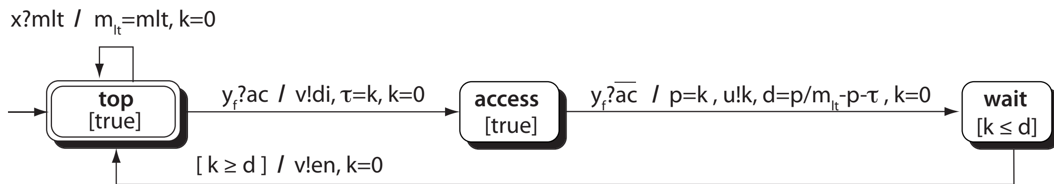


Figure 6.7: Automaton for secondary controller Q.

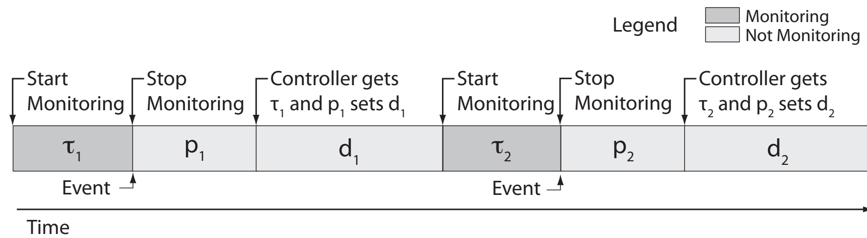


Figure 6.8: Timeline for secondary controller.

6.3.2.1 Secondary Controllers.

Each monitored plant P has a secondary controller Q, the state machine for which is given in Figure 6.7. Within each iteration of its main control loop, Q disables interrupts by sending message di along v upon receiving an access event ac along y, and subsequently enables interrupts by sending en along v. Consider the i-th execution of Q's control loop, and let τ_i be the *time monitoring is on* within this cycle; i.e., the time between events $v!en$

and $y?ac$. Let p_i be the time required to *process* event $y?ac$, and let d_i be the *delay time* until monitoring is restarted; i.e., until event $v!en$ is sent again. See Figure 6.8 for a graphical depiction of these intervals. Then the overhead in the i-th cycle is $o_i = p_i \,/\, (\tau_i + d_i)$ and accordingly, the *monitoring percentage* of the i-th cycle is $m_i = p_i \,/\, (p_i + \tau_i + d_i)$.

To ensure that $m_i = m_{lt}$ whenever the plant is throwing access events at a high rate, Q computes d_i as the least positive integer greater than or equal to $p_i/m_{lt} - (\tau_i + p_i)$. Choosing d_i this way lets the controller extend the total time spent in the i-th cycle so that its m_i is exactly the target m_{lt}.

To see how the secondary controller is like a proportional controller, regard p_i as a constant (p_i does not vary much in practice), so that p_i/m_{lt}—the desired value for the cycle time—is also a constant. The equation for d_i becomes now the difference between the desired cycle time (which we take to be the controller's reference value) and the actual cycle time measured when event i is finished processing. The value d_i is then equal to the proportional error for the i-th cycle, making the secondary controller behave like a proportional controller with proportional constant 1.

If plant P throws events at a low rate, then all events are monitored and $d_i = 0$. When processing of ac is finished, which is assumed to occur within the interval $[p_m, p_M]$, Q sends the processing time k to the primary controller along channel u.

6.3.2.2 Primary Controller.

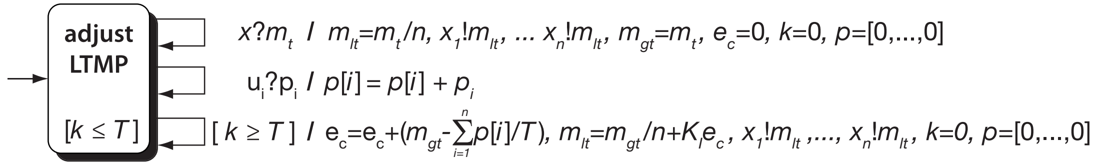


Figure 6.9: Automaton for the primary controller.

Secondary controller Q achieves its LTMP m_{lt} only if plant P throws events at a sufficiently high rate. Otherwise, its mean monitoring percentage $\overline{m}$ is less than m_{lt}. When monitoring a large number of plants P_i simultaneously, it is possible to take advantage of this under-utilization of m_{lt} by increasing the LTMP of those controllers Q_i associated with plants that throw interrupts at a high rate. In fact, we can adjust the m_{lt} of *all* secondary controllers Q_i by the same amount, as the controllers Q_j of plants P_j with low interrupt rates will not take advantage of this increase. Furthermore, we do this every T seconds, a period of time we call the *adjustment interval*. The periodic adjustment of the LTMP is the task of the primary controller PQ.

Its extended TA is given in Figure 6.9. After first inputting the UTMP m_t on x, PQ computes the initial LTMP to be m_t/n, thereby partitioning the global target monitoring percentage evenly among the n secondary controllers. It assigns this initial LTMP to the local variable m_{lt} and outputs it to the secondary controllers. It also assigns m_t to local variable m_{gt}, the *global target monitoring percentage* (GTMP). PQ also maintains an array p of total processing time, initially zero, such that $p[i]$ is the processing time used by secondary controller Q_i within the last adjustment interval of T seconds. Array entry $p[i]$

is updated whenever Q_i sends the processing time p_j of the most recent event a_j; i.e., $p[i]$ is the sum of the p_j that Q_i generates during the current adjustment interval.

When the time bound of T seconds is reached, PQ computes the error $e = m_{gt} - \sum_{i=1}^{n} p[i]/T$, as the difference between the GTMP and the observed monitoring percentage during the current adjustment interval. PQ also updates a cumulative error e_c, which is initially 0, such that $e_c = e_c + e$, making it the sum of the error over all adjustment intervals. To correct for the cumulative error, PQ computes an offset $K_I e_c$ that it uses to adjust m_{lt} down to compensate for over-utilization, and up to compensate for under-utilization. The new LTMP is set to $m_{lt} = m_{gt}/n + K_I e_c$ and sent to all secondary controllers, after which array p and clock k are reset.

Because the adjustment PQ makes to the LTMP m_{lt} over a given adjustment interval is a function of a cumulative error term e_c, primary controller PQ behaves as an *integrative controller*. In contrast, each secondary controller Q_i alone maintain no state beyond p_i and τ_i. They are therefore a form of *proportional controller*, which respond directly as the plant output changes. The controller parameter K_I in PQ's adjustment term $K_I e_c$ is known in control theory as the *integrative gain*. It is essentially a weight factor that determines to what extent the cumulative error e_c affects the local monitoring percentage m_{lt}. The larger the K_I value, the larger the changes PQ will make to m_{lt} during the current adjustment interval to correct for the observed overhead.

The *target specification language* L_P is defined in a fashion similar to the one for the secondary controllers, except that the events of the plant P are replaced by the events of the parallel composition $P_1 \parallel P_2 \parallel \ldots \parallel P_n$ of all plants.

6.4 Integer Range Analysis

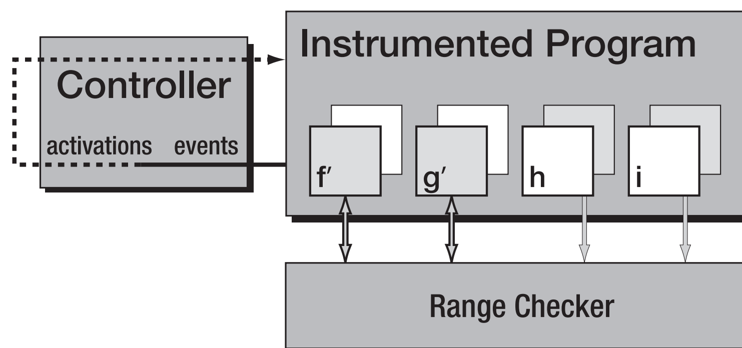


Figure 6.10: SMCO architecture for `range-checker`.

We have implemented several SMCO applications, including an integer range-analysis tool (`range-checker`), a memory staleness monitor (`Non-Accessed Period Detector`), an object assignment tracker (`C-DIDUCE`), and a pointer validity checker (`Bounds Checker`). In this thesis, we mainly discuss the integer range analysis application.

Integer range analysis [28] determines the range (minimum and maximum value) of each integer variable in the monitored execution. These ranges are useful for finding

program errors. For example, analyzing ranges on array subscripts may reveal bounds violations.

Figure 6.10 is an overview of `range-checker`, our integer range-analysis tool, which is implemented by our CAI tools described in Chapter 2. Our `range-checker` instrumentation plug-in adds range-update operations after assignments to global, function-level static, and stack-scoped integer variables. The *Range Checker* module (shown in Figure 6.10) consumes these updates and computes ranges for all tracked variables. Range updates are enabled or disabled on a per-function basis. In Figure 6.10, monitoring is enabled for functions f and g; this is reflected by the instrumented versions of their function bodies, labeled f' and g', appearing in the foreground.

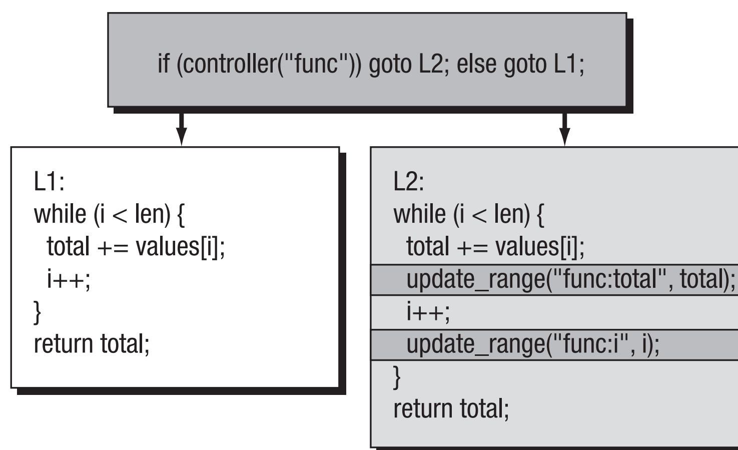


Figure 6.11: `range-checker` adds a distributor with a call to the SMCO controller. The distributed passes control to either the original, uninstrumented function body shown on the left, or the instrumented copy shown on the right.

To allow efficient enabling and disabling of monitoring, the plug-in creates a copy of the body of every function to be instrumented, and adds instrumentation only to the copy. A distributor block at the beginning of the function calls the SMCO controller to determine whether monitoring for the function is currently enabled. If so, the distributor jumps to the instrumented version of the function body; otherwise, control passes to the original, unmodified version. Figure 6.11 shows a function modified by the `range-checker` plug-in to have a distributor block and a duplicate instrumented function body. Functions without integer updates are not duplicated and always run with monitoring off.

Because monitoring is enabled or disabled at the function level, the instrumentation notifies the controller of function-call events. As shown in Figure 6.10, the controller responds by activating or deactivating monitoring for instrumented functions. With the global controller, there is a single "on-off" switch that affects all functions: when monitoring is off, the uninstrumented versions of all function bodies are executed. The cascade controller maintains a secondary controller for each instrumented function and can switch monitoring on and off for individual functions.

The controller and the `range-checker` monitor are implemented as external functions in shared object libraries which need to be linked with the instrumented program. To do so we added compiling flags in target program's makefile to specify the location of library files. However the make process can remain intact. Also, note that the monitoring functions defined in `range-checker` library does not actually take variable names and scope as input, as shown in Figure 6.11. In fact at compilation time the `range-checker` plug-in assigns each integer variable a unique id. The monitoring function takes the id of integer variables as input, instead of their name string. `range-checker` plug-in saves the name–id mapping for later analysis.

Loading a share object allows the program to register *at_exit* functions. When program exits, the *at_exit* function in `range-checker` library will be invoked to save the collected range information into a disk file. We also developed a statistic tool to plot the `range-checker` output and compute that accuracy of range-checking.

6.5 Clock Thread

As our controller logic relies on measurements of monitoring time, `range-checker` queries the system time whenever it makes a control decision. The RDTSC instruction is known as the fastest and most precise timekeeping mechanism on the x86 platform. It returns the CPU's timestamp counter (*TSC*), which stores a processor cycle timestamp (with sub-nanosecond resolution), without an expensive system call.

However, we found that even RDTSC can be too slow for our purposes. On our testbed, we measured the RDTSC instruction to take 45 cycles on average, more than twenty times longer than an arithmetic instruction. With time measurements necessary on every function call for our `range-checker`, this was too expensive. Our first `range-checker` implementation called RDTSC inline for every time measurement, resulting in a 23% overhead even with all monitoring turned off.

To reduce the high cost of timekeeping, we modified the `range-range-checker` to spawn a separate "clock thread" to handle its timekeeping. The clock thread periodically calls RDTSC and stores the result in a memory location that `range-checker` uses as its clock. `range-checker` can read this clock with a simple memory access. This is not as precise as calling RDTSC directly, but it is much more efficient.

The frequency that clock thread calls RDTSC is adjustable and implemented as an input parameter for `range-checker`. Apparently the more frequently that clock thread calls RDTSC, the more precise the artificial clock will be. However experiments show high clock thread frequency does not mean improvement on either the precision of SMCO's overhead-control or the accuracy of range-checking. Choice of a good clock thread frequency and its effect on range-checking accuracy is discussed in Section 7.4.1.

6.6 Controller Design

To implement and test the `range-checker` tool described in Section 6.4, we developed two versions of controller, the global controller and the cascade controller, as described in Chapter 6. The global controller consists of 180 lines of C code. The cascade controller,

whose logic is relatively more complex than the global controller, consists of 300 lines of C code.

For differ controllers, the `range-checker` library varies correspondingly. For the global controller, `range-checker`'s monitoring function only needs integer variables' id and current value as input, because the global controller thinks the whole program is a plant and treats all variable equally. In contrast, the cascade controller assigns a secondary controller to every function. All integer variables that belongs to one function are supervised by same secondary controller. To keep track of controlling/monitoring cost for every secondary controller, besides variable id and value, the function (secondary controller) id is also required to be passed into cascade controller's `range-checker` monitoring function.

Both controllers will spawn the clock thread for efficient timekeeping. When the controller library is detached from target program, a flag is set to notify the clock thread to gracefully exit itself.

Chapter 7

SMCO Experimental Evaluation

This chapter describes a series of benchmarks that together show that SMCO fulfills its goals: it closely adheres to the specified target overhead, allowing the user to specify a precise trade-off between overhead and monitoring effectiveness. In addition, our cascade controller apportions the target overhead to all sources of events, ensuring that each source gets its fair share of monitoring time.

Our results highlight the difficulty inherent in achieving these goals. The test workloads vary in behavior considerably over the course of an execution, making it impractical to predict sources of overhead. Even under these conditions, SMCO is able to control observed overhead fairly well.

To evaluate SMCO's versatility, we tested it on two workloads, one CPU-intensive and one I/O-intensive, and with our two different runtime monitors. Section 7.1 discusses our experimental testbed. Section 7.2 describes the workloads and profiles them in order to examine the challenges involved in controlling monitoring overhead. In Section 7.3, we benchmark SMCO's ability to control the overhead of our integer range analysis monitor using both of our control strategies. Section 7.4 explains how we optimized certain controller parameters.

7.1 Testbed

Since controlling overhead is most important for long-running server applications, we chose a server-class machine for our testbed. Our benchmarks ran on a Dell PowerEdge 1950 server with two quad-core 2.5GHz Intel Xeon processors, each with 12MB L2 cache, and 32GB of memory. The server was equipped with a pair of Seagate Savvio 15K RPM SAS 73GB disks in a mirrored RAID. We configured the server with 64-bit CentOS Linux 5.3, using a CentOS-patched 2.6.18 Linux kernel.

For our observed overhead benchmark figures, we averaged the results of ten runs and computed 95% confidence intervals using Student's t-distribution. Error bars represent the width of a measurement's confidence interval.

We tested our SMCO approach on two applications: the CPU-intensive `bzip2` and an I/O-intensive `grep` workload. The `bzip2` benchmark is a data compression workload

from the SPEC CPU2006 benchmark suite, which is designed to maximize CPU utilization [44]. This benchmark uses the `bzip2` utility to compress and then decompress a 53MB file consisting of text, JPEG image data, and random data.

Our `range-checker` monitor, described in Section 6.4, found 80 functions in `bzip2`, of which 61 contained integer assignments, and 445 integer variables, 242 of which were modified during execution. The integer-update events were spread very unevenly among these variables. The least-updated variables were assigned only one or two times during a run, while the most updated variable was assigned 2.5 billion times.

7.2 Workloads

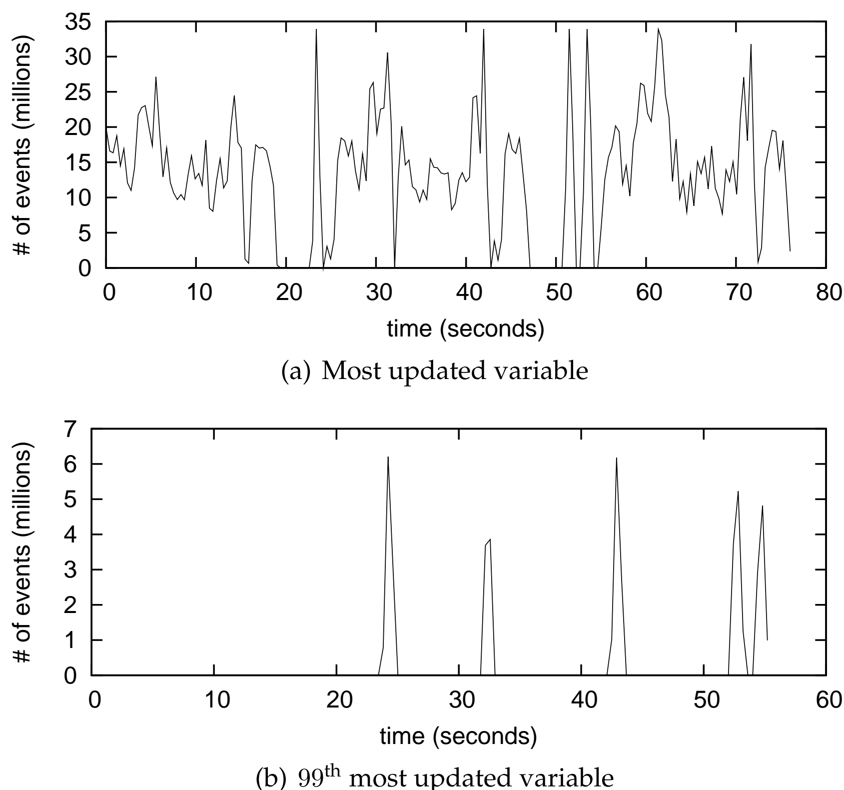


Figure 7.1: Event distribution histogram for the most updated variable (a) and 99th most updated variable (b) in `bzip2`. Execution time (x-axis) is split into 0.4 second buckets. The y-axis shows the number of events in each time bucket.

Figure 7.2 shows the frequency of accesses to two different variables, the most updated variable and the 99th most updated variable, over time. The data was obtained by instrumenting `bzip2` to monitor a single specified variable with unbounded overhead. The monitoring runs for these two variables took 76.4 seconds and 55.6 seconds, respectively. The two histograms show different extremes: the most updated variable is constantly active, while accesses to the 99th most updated variable are concentrated

in short periods of high activity. Both variables, however, experience heavy bursts of activity that make it difficult to predict monitoring overhead.

Our I/O-intensive workload uses GNU grep 2.5, the popular Linux regular expression search utility. In our benchmarks, grep searches the entire GCC 4.5 source tree (about 543MB in size) for an uncommon pattern. When we tested the workload with the Unix time utility, it reported that these runs typically used only 10–20% CPU time. Most of each run was spent waiting for read requests, making this an I/O-heavy workload. Because the grep workload repeats the same short tasks, we found that its variable accesses were distributed more uniformly than in bzip2. Our range-checker reported 489 variables, with 128 actually updated in each run, and 149 functions, 87 of which contained integer assignments. The most updated variable was assigned 370 million times.

7.3 Range Checker

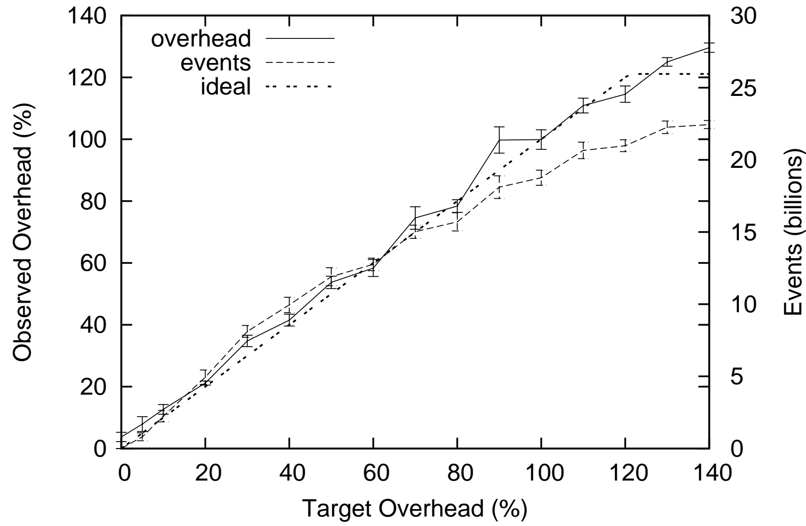
We benchmarked the range-checker monitor discussed in Section 6.4 on both workloads using both of the controllers in Section 6.3.1 and 6.3.2. Sections 7.3.1 and 7.3.2 present our results for the global controller and cascade controller, respectively. Section 7.3.3 compares the results from the two controllers. Section 7.3.4 discusses range-checker's memory overhead.

7.3.1 Global Controller

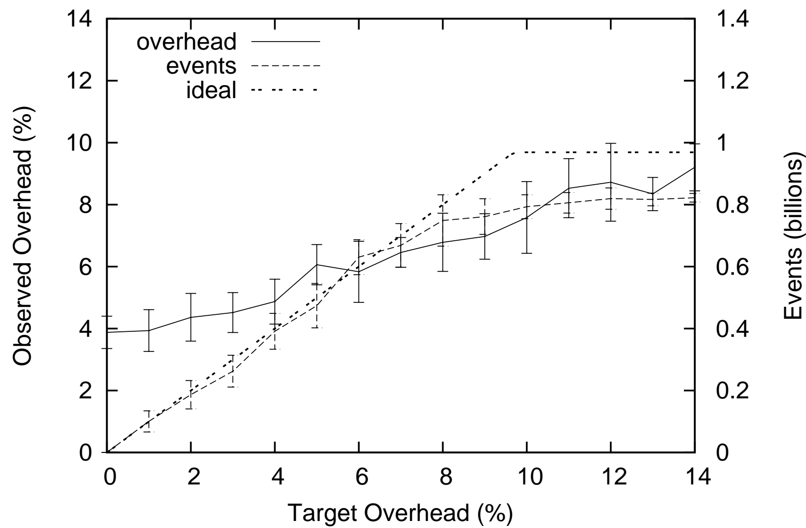
Figure 7.3.1 shows how the global controller performs on our workloads for a range of target overheads (on the x-axis), with the observed overhead on the y-axis and the total number of events monitored on the $y2$-axis for each target-overhead setting. With target overhead set to 0%, both workloads ran with an actual overhead of 4%, which is the controller's *base overhead*. The base overhead is due to the controller logic and the added complexity from unused instrumentation.

The dotted line in each plot shows the ideal result: observed overhead equals target overhead up to an ideal maximum. We computed the ideal maximum to be the observed overhead from monitoring all events in the program with all control turned off. Any observed overhead above the ideal maximum is the result of overhead incurred by the controller.

At target overheads of 10% and higher, Figure 7.2(a) shows that the global controller tracked the specified target overhead all the way up to 140% in the bzip2 workload. The grep workload (Figure 7.2(b)) showed a general upward trend for increasing target overheads, but never exceeded 9% observed overhead. In fact, at 9% overhead, range-checker is already at nearly full coverage, with 99.7% percent of all program events being monitored. The grep workload's low CPU usage imposes a hard limit on range-checker's ability to use overhead. The controller has no way to exceed this limit. Confidence intervals for the grep workload were generally wider than for bzip2, because I/O operations are noisier than CPU operations, making run times less consistent.



(a) bzip2

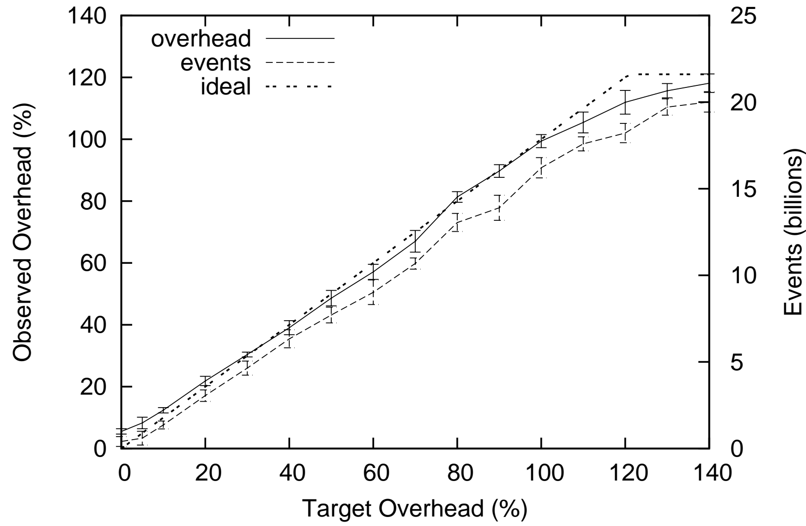


(b) grep

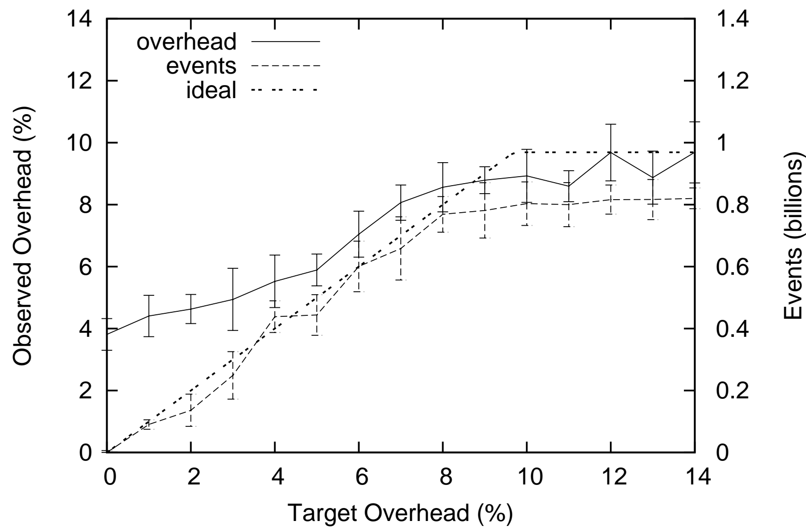
Figure 7.2: Global controller with range-checker *observed overhead (y-axis) for a range of target overhead settings (x-axis) and two workloads.*

7.3.2 Cascade Controller

Figure 7.3.2 shows results from experiments that are the same as those for Figure 7.3.1 except using the cascade controller instead of the global controller. The results were similar. On the `bzip2` workload, the controller tracked the target overhead well from 10% to 100%. With targets higher than 100%, the observed overhead continued to increase, but the controller was not able to adjust overhead high enough to reach the target because observed overhead was already so close to the 120% maximum. On the `grep` workload, we saw the same upward trend and eventual saturation with 9% observed overhead monitoring 99.5% of events.



(a) bzip2



(b) grep

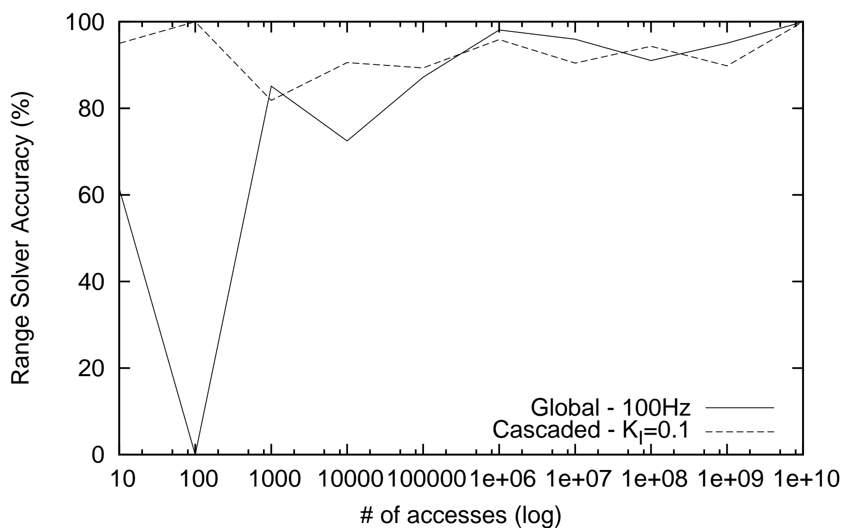
Figure 7.3: Cascade controller with range-checker *observed overhead (y-axis) for a range of target overhead settings (x-axis) and two workloads.*

7.3.3 Controller Comparison

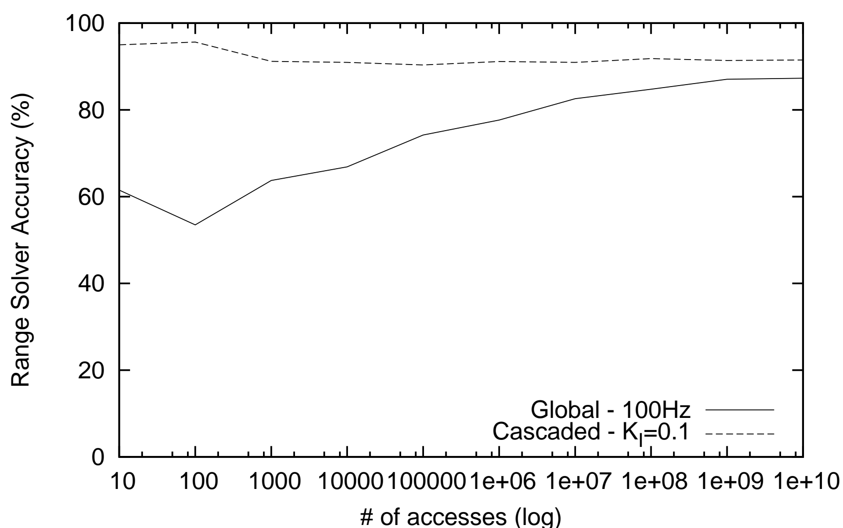
The global and cascade controllers differ in the distribution of overhead across different event sources. To compare them, we developed an accuracy metric for the results of a bounded-overhead `range-checker` run. We measured the accuracy of a bounded-overhead run of the `range-checker` against a reference run with full coverage of all variables (allowing unbounded overhead). The reference run determined the actual range for every variable.

In a bounded-overhead run, the accuracy of a range computed for a single variable is the ratio of the computed range size to the range's actual size (which is known from the

reference run). Missed updates in a bounded-overhead run can cause `range-checker` to report smaller ranges, so this ratio is always in the interval $[0, 1]$. For a set of variables, the accuracy is the average accuracy for all variables in the set.



(a) Non-cumulative



(b) Cumulative

Figure 7.4: Comparison of range-checker accuracy *for both controllers with `bzip2` workload. Variables are grouped by total number of updates.*

Figure 7.3.3 shows a breakdown of `range-checker`'s accuracy by how frequently variables are updated. We grouped variables into sets with geometrically increasing bounds: the first set containing variables with 1–10 updates, the second group containing variables with 10–100 updates, etc. Figure 7.4(a) shows the accuracy for each of these sets, and Figure 7.4(b) shows the cumulative accuracy, with each set containing the variables from the previous set.

	Exe Size	*VSZ*	*RSS*
bzip2 (unmodified)	68.6KB	213KB	207KB
bzip2 (global)	262KB	227KB	203KB
bzip2 (cascade)	262KB	225KB	201KB
grep (unmodified)	89.2KB	61.4MB	1260KB
grep (global)	314KB	77.1MB	1460KB
grep (cascade)	314KB	78.2MB	1470KB

Table 7.1: `range-checker` memory usage, including executable size, virtual memory usage (VSZ), and physical memory usage (RSS).

We used 10% target overhead for these examples, because we believe that low target overheads represent the most likely use cases. However, we found similar results for all other target overhead values that we tested.

The cascade controller's notion of fairness results in better coverage, and thus better accuracy, for rarely updated variables. In this example, the cascade controller had better accuracy than the global controller for variables with fewer than 100 updates. As the global controller does not seek to fairly distribute overhead to these variables, it monitored a smaller percentage of their updates. Most dramatically, Figure 7.4(a) shows that the global controller had 0 accuracy for all variables in the 10–100 updates range, meaning it did not monitor more than one event for any variable in that set. The 3 variables in the workload with 10–100 updates were used while there was heavy activity, causing their updates to get lost in the periods when the global controller had to disable monitoring to reduce overhead.

However, with the same overhead, the global controller was able to monitor many more events than the cascade controller, because it did not spend time executing the cascade controller's more expensive secondary controller logic. These extra events gave the global controller much better coverage for frequently updated variables. Specifically, it had better accuracy for variables with more than 10^6 updates.

Between these two extremes, i.e., for variables with 100–10^6 updates, both approaches had similar accuracy. The cumulative accuracy in Figure 7.4(b) shows that overall, considering all variables in the program, the two controllers achieved similar accuracy. The difference is primarily in where the accuracy was distributed.

7.3.4 Memory Overhead

Although `range-checker` does not use SMCO to control memory overhead, we measured memory use of our controllers for both workloads. Table 7.3.4 shows our memory-overhead results. Here *Exe Size* is the size of the compiled binary after stripping debugging symbols (as is common in production environments). This size includes the cost of the SMCO library, which contains the compiled controller and monitor code. *VSZ* is the total amount of memory mapped by the process, and *RSS* (Resident Set Size) is the total amount of that virtual memory stored in RAM. We obtained the peak VSZ and RSS for each run using the Unix `ps` utility.

Both binaries increased in size by 3–4 times. Most of this increase is the result of function duplication, which at least doubles the size of each instrumented function. Duplicated functions also contain a distributor block and instrumentation code. The 17KB SMCO library adds a negligible amount to the instrumented binary's size. As few binaries are more than several megabytes in size, we believe that even a 4x increase in executable size is acceptable for most environments; this is more true these days, with increasing amounts of RAM in popular 64-bit systems.

The worst-case increase in virtual memory use was only 27.4%, for the `grep` workload with the cascade controller. The additional virtual memory is allocated statically to store integer variable ranges and per-function overhead measurements (when the cascade controller is used). This extra memory scales linearly with the number of integer variables and functions in the monitored program, not with runtime memory usage. The `bzip2` workload uses more memory than `grep`, so we measured in this case a much lower 6.6% virtual memory overhead.

7.4 Controller Optimization

This section describes how we chose values for several of our control parameters in order to get the best performance from our controllers. Section 7.4.1 discusses our choice of clock precision for time measurements. Section 7.4.2 explains how we chose the integrative gain and adjustment interval for the primary controller. Section 7.4.3 discusses optimizing the adjustment interval for an alternate primary controller.

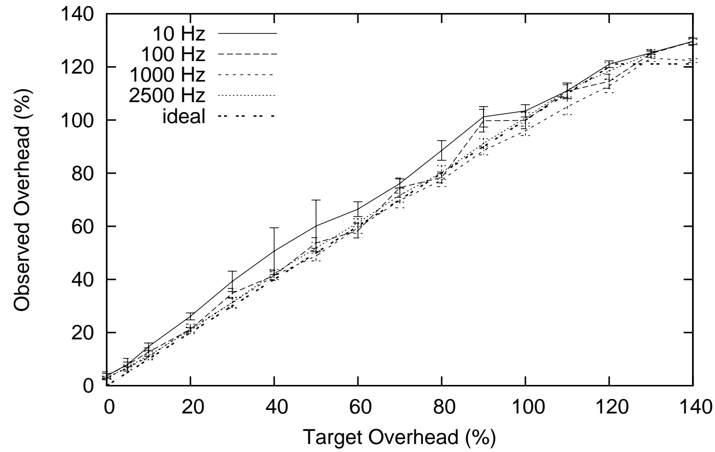
7.4.1 Clock Frequency

The control logic for our `range-checker` implementation uses a *clock thread*, as described in Section 6.5, which trades off precision for efficiency. This thread can keep more precise time by updating its clock more frequently, but more frequent updates result in higher timekeeping overhead. Recall that the RDTSC instruction is relatively expensive, taking 45 cycles on our testbed.

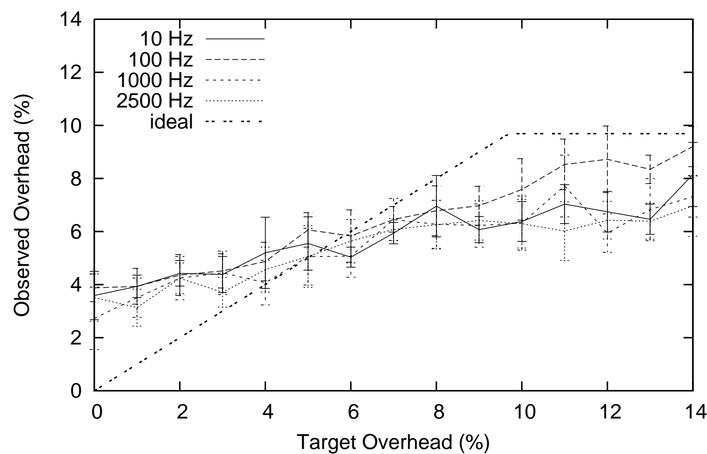
We performed experiments to determine how much precision was necessary to control overhead accurately. Figure 7.4.1 shows the `range-checker` benchmark in Figure 7.3.1 repeated for four different clock frequencies. The *clock frequency* is how often the clock thread wakes up to read the TSC.

At only 10Hz, the controller's time measurements were not accurate enough to keep the overhead below the target. With infrequent updates, most monitoring operations occurred without an intervening clock tick and therefore appeared to take 0 seconds. The controller ended up with an under-estimate of the actual monitoring overhead and thus overshot its goal.

At 100Hz, however, controller performance was good and the clock thread's impact on the system was still negligible, incurring the 45-cycle RDTSC cost only one hundred times for every 2.5 billion processor cycles on our test system. More frequent updates did not perform any better and wasted resources, so we chose 100Hz for our clock update frequency.



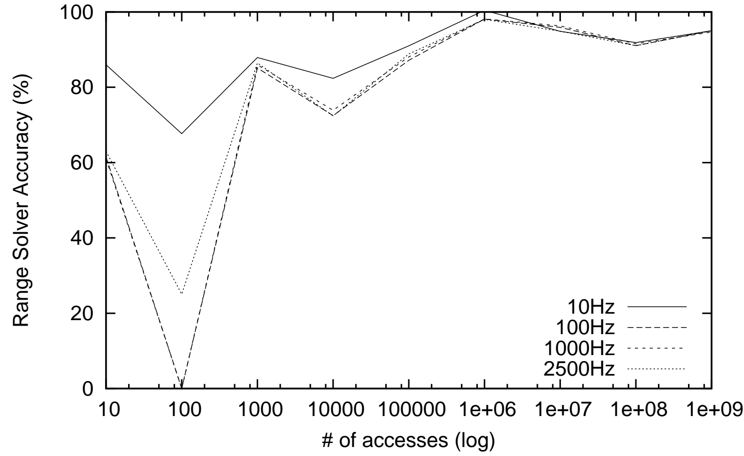
(a) bzip2



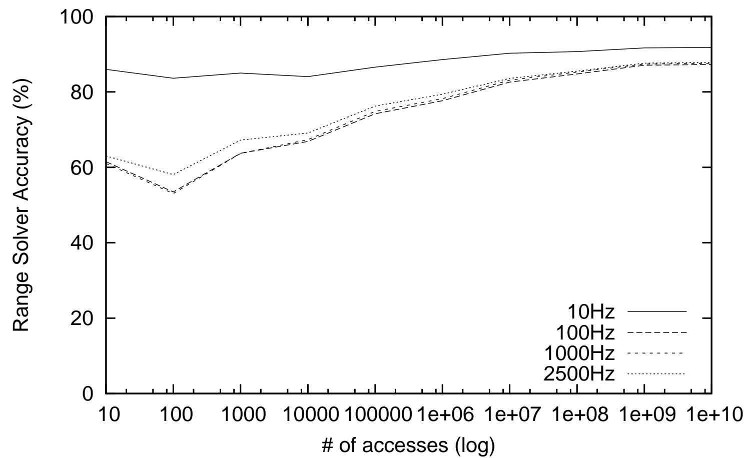
(b) grep

Figure 7.5: Observed overhead for global controller clock frequencies *with 4 different clock frequencies and 2 workloads using* `range-checker` *instrumentation.*

In choosing the clock frequency, we wanted to ensure that we also preserved SMCO's effectiveness. Figure 7.4.1 shows the accuracy of the `range-checker` at 10% target overhead using the four clock frequencies we tested. We used the same accuracy metric as in Section 7.3.3 and plotted the results as in Figure 7.3.3. The `range-checker` accuracy is similar for 100Hz, 1000Hz, and 2500Hz clocks. These three values resulted in similar observed overheads in Figure 7.4.1. It therefore makes sense that they achieve similar accuracy, since SMCO is designed to support trade-offs between effectiveness and overhead. The 10Hz run has the best accuracy, but this result is misleading because it attains that accuracy at the cost of higher overhead than the user-requested 10%. Testing these clock frequencies at higher target overheads showed similar behavior. Note that the 100Hz curves in Figure 7.4.1 are the same as the global controller curves from the controller comparison in Figure 7.3.3.



(a) Non-cumulative



(b) Cumulative

Figure 7.6: Accuracy of global controller clock frequencies *with 4 different clock frequencies using* `range-checker` *instrumentation. Variables are grouped by total number of updates.*

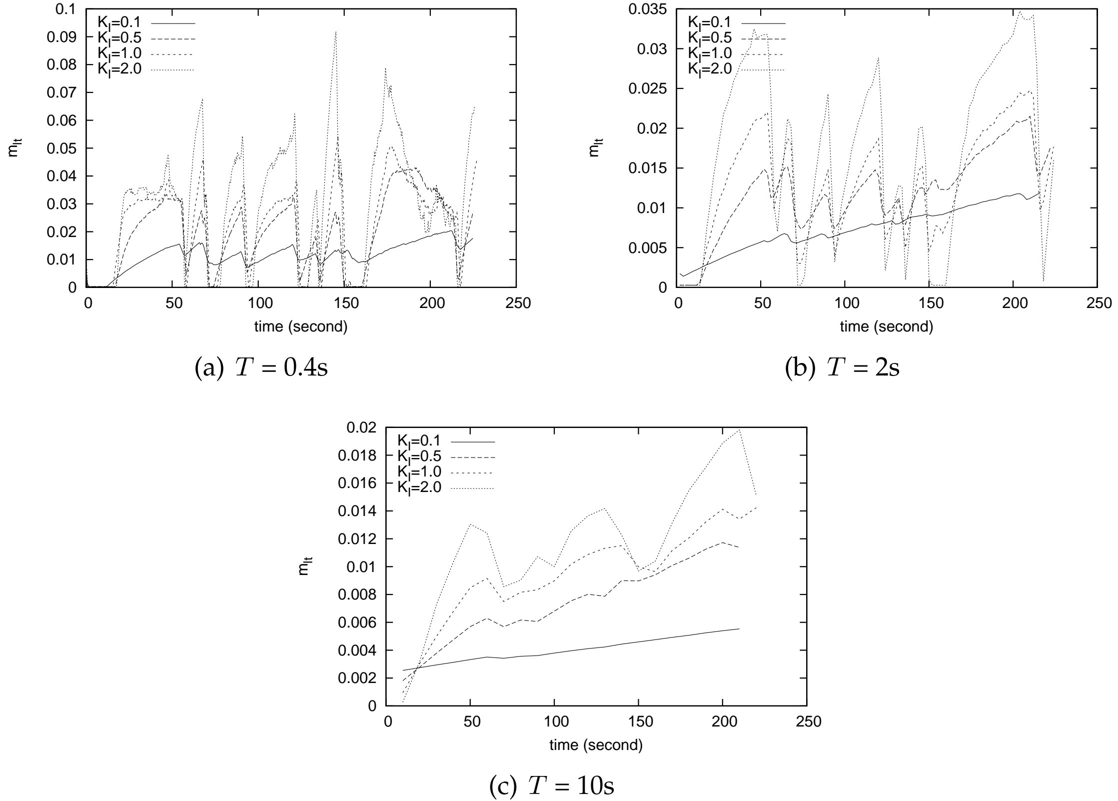
7.4.2 Integrative Gain

The primary controller component of the cascade controller discussed in Section 6.3.2 has two parameters: the integrative gain K_I and the adjustment interval T. The integrative gain is a weight factor that determines how much the cumulative error e_c (the deviation of the observed monitoring percentage from the target monitoring percentage) changes the local target monitoring percentage m_{lt} (the maximum percent of execution time that each monitored plant is allowed to use for monitoring). When K_I is high, the primary controller makes larger changes to m_{lt} to correct for observed deviations.

The adjustment interval is the period of time between primary controller updates. With a low T value, the controller adjusts m_{lt} more frequently.

There are processes for choosing good control parameters for most applications of

control theory, but our system is tolerant enough that good values for K_I and T can be determined experimentally.



(a) T = 0.4s

(b) T = 2s

(c) T = 10s

Figure 7.7: Local target monitoring percentage (m_{lt}) over time *during* `bzip2` *workload for* `range-checker` *with cascade control. Results shown with target overhead set to 20% for 4 different values of K_I and 3 values of T.*

We tuned the controller by running `range-checker` on an extended `bzip2` workload with a range of values for K_I and T and then recording how the primary controller's output variable, m_{lt}, stabilized over time. Figure 7.4.2 shows our results for four K_I values and three T values with target overhead set to 20% for all runs. These results revealed trends in the effects of adjusting the controller parameters.

In general, we found that higher values of K_I increased controller responsiveness, allowing m_{lt} to more quickly compensate for under-utilization of monitoring time, but at a cost of driving higher-amplitude oscillations in m_{lt}. This effect is most evident with T = 0.4s (see Figure 7.7(a)). All the values we tested from $K_I = 0.1$ to $K_I = 2.0$ successfully met our overhead goals, but values greater than 0.1 oscillated wildly enough that the controller had to sometimes turn monitoring off completely (by setting m_{lt} to almost 0) to compensate for previous spikes in m_{lt}. With $K_I = 0.1$ however, m_{lt} oscillated stably for the duration of the run after a 50-second warm-up period.

When we changed T to 2s (see Figure 7.7(b)), we observed that 0.1 was no longer the

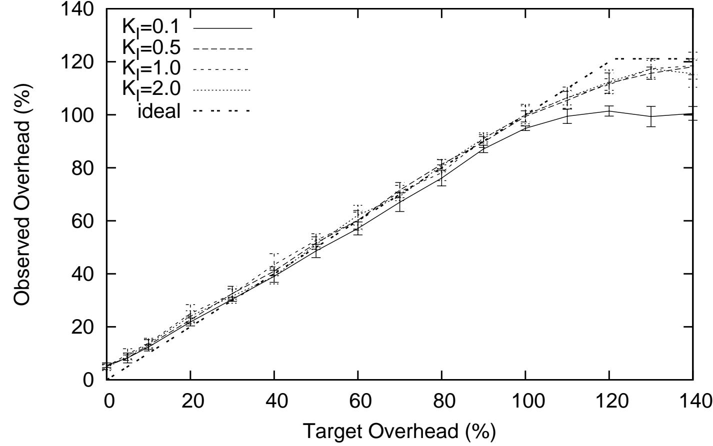


Figure 7.8: Observed overhead for primary controller K_I values *using* `range-checker` *with T = 400ms and four different K_I values.*

optimal value for K_I. But with $K_I = 0.5$, we were able to obtain performance as good as the optimal performance with T = 0.4s: the controller met its target overhead goal and had the same 50-second warmup time. We do see the consequences of choosing too small a K_I, however: with $K_I = 0.1$, the controller was not able to finish warming up before the benchmark finished, and the system failed to achieve its monitoring goal.

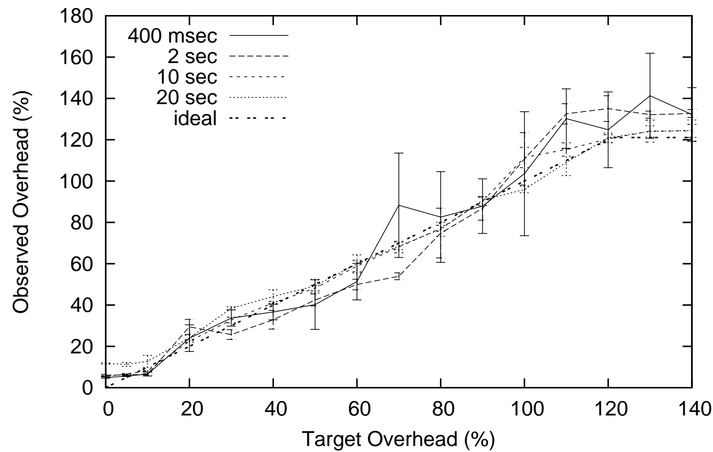
The same problem occured when T = 10s (see Figure 7.7(c)): the controller updated so infrequently that it only completed its warm-up for the highest K_I value we tried. Even with the highest K_I, the controller still undershot its monitoring percentage goal.

Because of its stability, we chose $K_I = 0.1$ (with T = 0.4s) for all of our cascade controlled `range-checker` experiments with low target overhead. Figure 7.4.2 shows how the controller tracked target overhead for all the K_I values we tried. Although $K_I = 0.1$ worked well for low overheads, we again observed warmup periods that were too long when target overhead was very high. The warmup period took longer with very high overhead because of the larger gap between target overhead and the initial observed overhead. To deal with this discrepancy, we actually use two K_I values, $K_I = 0.1$ for normal cases, and a special high-overhead K_I^H for cases with target overhead greater than 70%. We chose $K_I^H = 0.5$ using the same experimental procedure we used for K_I.

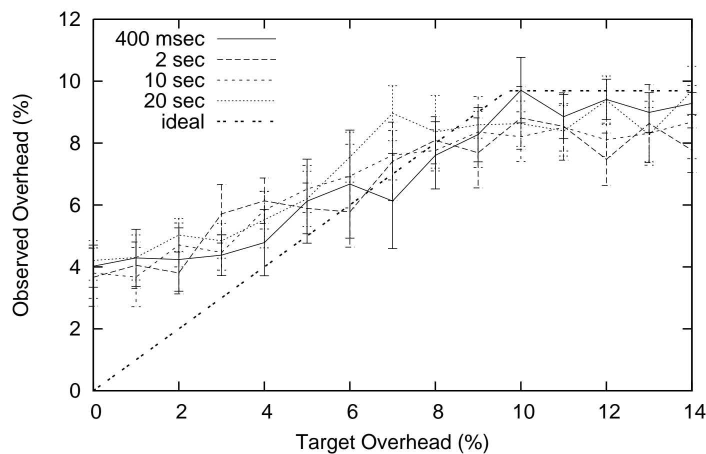
7.4.3 Adjustment Interval

Before settling on the integrative control approach that we use for our primary controller, we attempted an ad hoc control approach that yielded good results for the `range-checker`, but was not stable enough to control the NAP Detector. We also found the ad hoc primary controller to be more difficult to adjust than the integrative controller. Though we no longer use the ad hoc approach, the discussion of how we tuned it illustrates some of the properties of our system.

Rather than computing error as a difference, the ad hoc controller computes error e_f fractionally as the Global Target Monitoring Percentage (GTMP, as in Section 6.3.2) di-



(a) bzip2



(b) grep

Figure 7.9: Observed overhead for an ad hoc cascade controller's T values *with 4 different values of T and two* `range-checker` *workloads.*

vided by the observed global monitoring percentage for the entire program run so far. The value of e_f is greater than one when the program is under-utilizing its monitoring time and less than one when the program is using too much monitoring time. After computing the fractional error, the controller computes a new value for m_{lt} by multiplying it by e_f.

The ad hoc primary controller has only one parameter to adjust. Like the integrative controller, it has an interval time T between updates to m_{lt}.

Figure 7.4.2 shows the `range-checker` benchmark in Figure 7.3.2 repeated for four values of T. For the `bzip2` workload (Figure 7.9(a)), the best results were obtained with a T interval of 10 seconds. Smaller T values led to an unstable primary controller: the observed overhead varied from the target and results became more random, as shown by the wider confidence intervals for these measurements.

This result contradicts the intuition that a smaller T should stabilize the primary con-

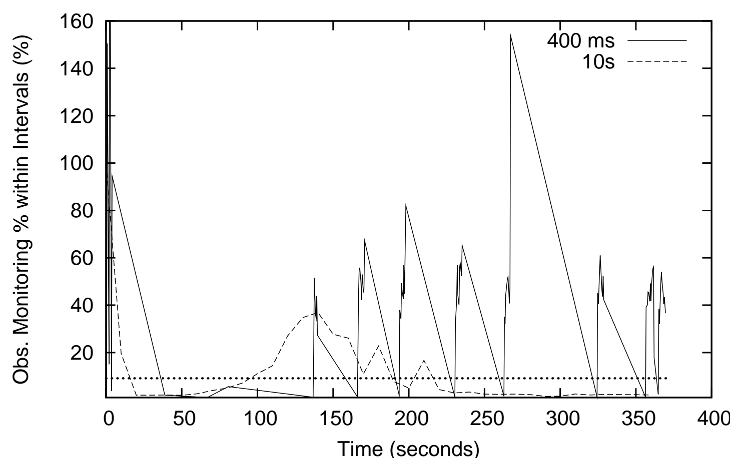


Figure 7.10: Observed monitoring percentage over bzip2 `range-checker` execution. *The percent of each adjustment interval spent monitoring for 2 values of T. The target monitoring percentage is shown as a dotted horizontal line.*

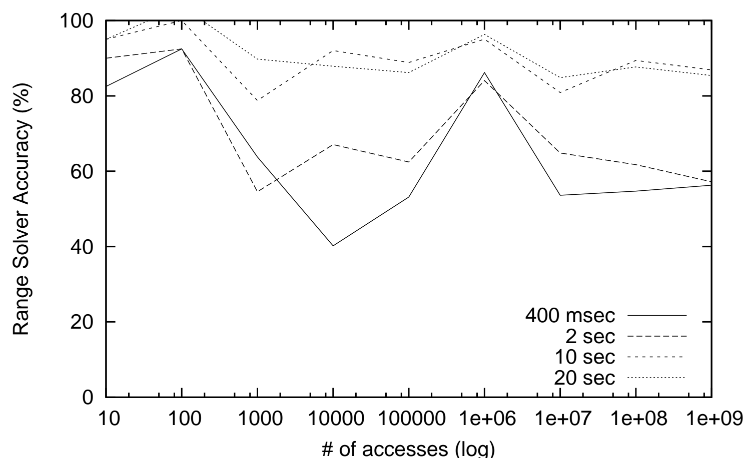
troller by allowing it to react faster. In fact, the larger T gives the controller more time to correctly observe the trend in utilization among monitoring sources. For example, with a short T, the first adjustment interval of `bzip2`'s execution used only a small fraction of all the variables in the program. The controller quickly adjusted m_{lt} very high to compensate for the monitoring time that the unaccessed variables were failing to use. In following intervals, the controller needed to adjust m_{lt} down sharply to offset the overhead from many monitoring sources becoming active at once. The controller spent the rest of its run oscillating to correct its early error, never reaching equilibrium during the entire run.

Figure 7.4.3 shows how the observed monitoring percentage for each adjustment interval fluctuates during an extended `range-checker` run of the `bzip2` workload as a result of the primary controller's m_{lt} adjustments. With $T = 0.4$ seconds, the observed monitoring percentage spikes early on, so observed overhead is actually very high at the beginning of the program. Near 140 seconds, the controller overreacts to a change in program activity, causing another sharp spike in observed monitoring percentage. As execution continues, the observed monitoring percentage sawtooths violently, and there are repeated bursts of time when the observed percentage is much higher than the target percentage (meaning observed overhead is much higher than the user's target overhead).

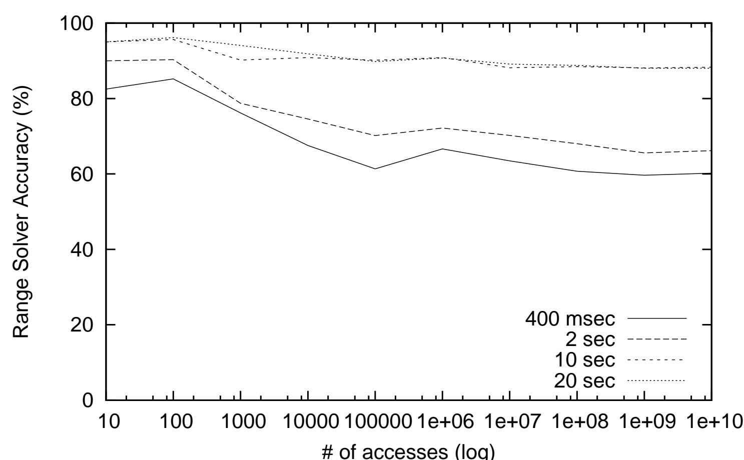
With $T = 10$ seconds, the observed monitoring percentage still fluctuates, but the extremes do not vary as far from the target. As execution continues, the oscillations dampen, and the system reaches stability.

In our `bzip2` workload, the first few primary controller intervals were the most critical: bad values at the beginning of execution were difficult to correct later on. The more reasonable 10 second T made its first adjustment after a larger sample of program activity, so it did not overcompensate. Overall, we expect that a primary controller with a longer T is less likely to be misled by short bursts or lulls in activity.

There is a practical limit to the length of T, however. In Figure 7.9(a), a controller with $T = 20$ seconds overshot its target overhead. Because the benchmark runs for only about one minute, the slower primary controller was not able to adjust m_{lt} often enough to converge on a stable value before the benchmark ended.



(a) Non-cumulative



(b) Cumulative

Figure 7.11: Accuracy of cascade controller T values *with 4 values of T on the* `bzip2` *workload using* `range-checker` *instrumentation. Variables are grouped by total number of updates.*

As in Section 7.4.1 we also tested how the choice of T affects the `range-checker`'s accuracy, using the same accuracy metric as in Section 7.3.3. Figure 7.4.3 shows the accuracy for the `bzip2` workload using four different values for T with a 10% target overhead. The accuracy results confirm our choice of 10 seconds. Only the 20 second value for T yields better accuracy, but it does so because it consumes more overhead than the primary controller with $T = 10$ seconds. Tests with higher target overheads gave similar results.

7.5 Conclusion

Our results show that in all the cases we tested, SMCO was able to track the user-specified target overhead for a wide range of target overhead values. Even when there were too few events during an execution to meet the target overhead goal, as was the case for the `grep` workload with target overheads greater than 10%, SMCO's controller was able to achieve the maximum possible observed overhead by monitoring nearly all events. We also showed that the overhead trade-off is a useful one: higher overheads allowed for more effective monitoring. These results are for challenging workloads with unpredictable bursts in activity.

Although our results relied on choices for several parameters, we found that it was practical to find good values for all of these parameters empirically. As future work, we plan to explore procedures for automating the selection of optimal controller parameters, which can vary with different types of monitoring.

Chapter 8

Conclusion and Future Work

8.1 Compiler-Assisted Techniques Conclusion

In this dissertation we have presented compiler-assisted techniques including the *GLua* as a program instrumentation tool and the GVM as a software concrete execution engine. Traditionally, program instrumentation requires a tedious work of rewriting of a new program parser. Some instrumenters seek to modify compiler source code, but only for a specific purpose, like profiling or memory debugging. Based on the GCC plug-in architecture, CAI provides an easy means of program instrumentation for general purpose. It also allows user s to directly utilize compiler's static analysis results. We further present *GLua*, a scripted program instrumentation tool, to save users from dealing with GCC internal data structures. Upon *GLua*, we developed several auxiliary tools to facilitate program instrumentation.

GVM is another example of how programmers can benefit from compilers. Based on GIMPLE intermediate representation, GVM does not require an instruction set and takes least effort to implement due to GIMPLE's simplicity. On GVM a user can have full access to VM's internal state. and control of interpretation of target programs.

We applied these techniques into our research projects. We use GVM as the concrete execution engine in two execution-based software model checking approaches, GMC^2 and DPR-MC. GMC^2 is the application of Monte Carlo model checking algorithm on C program. It traverses program CFGs and interprets GIMPLE statements according to their semantics. DPR uses GVM to compute concrete execution paths, and its results will be passed to a symbolic solver. DPR learns from symbolic analysis of concrete path informations and prunes unexplored paths as early as possible. Results show DPR prunes a significant percentage of execution path and this percentage increases with case size.

In SMCO we use CAI to instrument controller and monitor facilities into target program. Benchmark shows that SCMO controls overhead across a wide rage of target-overhead levels; its accuracy is also satisfactory.

8.2 Future Work

In the future we will work to further improve *GLua* and GVM. There could be many wonderful applications that *GLua* can make, e.g., something similar as Python's *decorator* syntax. GVM does not support many POSIX system calls and its multi-process scheduler is still instable. Although GVM works well on most model checking target programs, it is still problematic with general C programs, especially those with frequent pointer manipulations.

Bibliography

[1] A. Aziz, F. Balarin, R.K. Brayton, M.D. Dibenedetto, A. Sladanha, and A.L. Sangiovanni- Vincentelli. Supervisory Control of Finite State Machines. In P. Wolper, editor, *7th International Conference On Computer Aided Verification*, volume 939, pages 279–292, Liege, Belgium, 1995. Springer Verlag.

[2] B. Adams, C. Herzeel, and K. Gybels. cHALO, stateful aspects in C. In *ACP4IS '08: Proceedings of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software*, pages 1–6, New York, NY, USA, 2008. ACM.

[3] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[4] AT&T Research Labs. Graphviz, 2009. `http://www.graphviz.org`.

[5] T. Ball, A. Podelski, and S. K. Rajamani. Relative Completeness of Abstraction Refinement for Software Model Checking. In Joost-Pieter Kaoen and Perdita Stevens, editors, *Proceedings of TACAS02: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 158–172, Grenoble, France, April 2002. Springer-Verlag.

[6] T. Ball and S.K. Rajamani. The SLAM toolkit. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 260–264. Springer-Verlag, 2001.

[7] A. Bauer, M. Leucker, and C. Schallhart. Runtime Verication for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2009.

[8] A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL Semantics for Runtime Verication. *Journal of Logic and Computation*, 20(3):651–674, 2010.

[9] D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker Blast. In *International Journal on Software Tools for Technology Transfer (STTT)*, pages 505–525, 2007.

[10] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207. Springer, 1999.

[11] J. Burnim and K. Sen. Heuristics for Scalable Dynamic Test Generation. In *ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society.

[12] S. Callanan, D. J. Dean, and E. Zadok. Extending GCC with Modular GIMPLE Optimizations. In *Proceedings of the 2007 GCC Developers' Summit*, Ottawa, Canada, July 2007.

[13] S. Callanan, R. Grosu, X. Huang, S. A. Smolka, and E. Zadok. Compiler-Assisted Software Verification Using Plug-Ins. In *Proceedings of the 2006 NSF Next Generation Software Workshop, in conjunction with the 2006 International Parallel and Distributed Processing Symposium (IPDPS 2006)*, Rhodes Island, Greece, April 2006. DOI 10.1109/IPDPS.2006.1639579.

[14] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 15–28, 2004.

[15] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM TOPLAS*, 8(2), 1986.

[16] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 1999.

[17] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezze. Using Symbolic Execution for Verifying Safety-Critical Systems. In *ESEC/FSE-9: Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 142–151, New York, NY, USA, 2001. ACM Press.

[18] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzè. Using Symbolic Execution for Verifying Safety-Critical Systems. In *ESEC/FSE-9: Proc. 8th European Software Engineering Conference*, pages 142–151. ACM Press, 2001.

[19] S. Colin and L. Mariani. *Run-Time Verication*. Springer-Verlag LNCS 3472, 2005.

[20] The GCC Community. *GCC Internals*. 1988-2010.

[21] Computer Science Laboratory, SRI International. *Yices, an SMT Solver*. `http://yices.csl.sri.com`.

[22] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[23] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

[24] M. D'Amorim and K. Havelund. Runtime Verification for Java. In *Workshop on Dynamic Program Analysis (WODA'05)*, March 2005.

[25] D. J. Dean, S. Callanan, and E. Zadok. The Visual Development of GCC Plug-ins. In *Proceedings of the 2009 GCC Developers' Summit*, Montreal, Canada, June 2009.

[26] E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[27] Y. Falcone, J.-C. Fernandez, and L. Mounier. Runtime Verification of Safety-Progress Properties. In S. Bensalem and D. Peled, editors, *Proc. of the 9th International Workshop on Runtime Verification (RV'09)*, volume 5779 of *LNCS*, pages 40–59. Springer, 2009.

[28] L. Fei and S.P. Midkiff. Artemis: Practical Runtime Monitoring of Applications for Errors. Technical Report TR-ECE-05-02, Electrical and Computer Engineering, Purdue University, 2005. `docs.lib.purdue.edu/ecetr/4/`.

[29] G.F. Franklin, J.D. Powell, and M. Workman. *Digital Control of Dynamic Systems, Third Edition*. Addison Wesley Longman, Inc., 1998.

[30] V. Ganesh and D.L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *Computer Aided Verification (CAV '07)*, Berlin, Germany, July 2007. Springer-Verlag.

[31] G.C. Necula, S. McPeak, S.P. Rahul and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, England, 2002. Springer-Verlag.

[32] GCC 4.5 Release Series Changes, New Features, and Fixes. `http://gcc.gnu.org/gcc-4.5/changes.html`.

[33] The GCC team. *GCC Online Documentation*, December 2005. `http://gcc.gnu.org/onlinedocs/`.

[34] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[35] P. Godefroid. Model Checking for Programming Languages Using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[36] P. Godefroid and S. Khurshid. Exploring Very Large State Spaces Using Genetic Algorithms. In *TACAS*, pages 266–280. Springer, 2002.

[37] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. *SIGPLAN Not.*, 40(6):213–223, 2005.

[38] S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: A Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 120–126, June 1982.

[39] R. Grosu, X. Huang, S. Jain, and S.A. Smolka. Open Source Model Checking. In *Proc. of SoftMC'05, the 3rd Workshop on Software Model Checking*, July 2005.

[40] R. Grosu and S. A. Smolka. Monte Carlo Model Checking. In *Proceedings of the 11th Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2005.

[41] K. Havelund. Runtime Verification of C Programs. In *Proc. of the 1st TestCom/FATES conference*, volume 5047 of *LNCS*, Tokyo, Japan, June 2008. Springer.

[42] K. Havelund and A. Goldberg. Verify your runs. 2008.

[43] K. Havelund and T. Pressburger. Model Checking Java Programs Using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[44] J. L. Henning. SPEC CPU2006 Benchmark Descriptions. *Computer Architecture News*, 34(4):1—17, September 2006.

[45] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with Blast. In *Proceedings of the 10th SPIN Workshop on Model Checking Software (SPIN 2003)*, page 235239. Springer-Verlag, 2003.

[46] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

[47] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21:666–677, August 1978.

[48] X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S. A. Smolka, S. D. Stoller, and E. Zadok. Software monitoring with controllable overhead. *International Journal on Software Tools for Technology Transfer (STTT)*, 2010.

[49] J. Hubicka. Profile Driven Optimisations in GCC. In *Proceedings of the GCC Developers' Summit*, volume 216, pages 107–124. The GCC Community, 2005.

[50] J. Burnim and K. Sen. *CREST, an Automatic Test Generation Tool for C*. `http://http://code.google.com/p/crest/`.

[51] P. J.Ramadge and W.M. Wonham. Supervisory Control of Timed Discrete-Event Systems. *IEEE Transactions on Automatic Control*, 38(2):329–342, 1994.

[52] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–355. LNCS, Vol. 2072, 2001.

[53] G. Lowe. An Attack on the Needham-Schroeder Public-Key Authentication Protocol. *Information Processing Letters*, pages 131–133, 1995.

[54] lua.org. *The Programming Luaguage Lua*. http://www.lua.org.

[55] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[56] J. Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In *GCC Developers Summit*, 2003.

[57] R. Moore. Dynamic Probes and Generalised Kernel Hooks Interface for Linux. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 135–146, Atlanta, GA, October 2000. USENIX Association.

[58] R. Moore. A Universal Dynamic Trace for Linux and other Operating Systems. In *Proceedings of the 2001 USENIX Annual Technical Conference*, June 2001.

[59] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI '02)*, pages 75–88, Boston, MA, December 2002. USENIX Association.

[60] G. Necula. CIL - Infrastructure for C Program Analysis and Transformation, 2007. http://manju.cs.berkeley.edu/cil.

[61] D. Novillo. TreeSSA: A New Optimization Infrastructure for GCC. In *Proceedings of the 1st GCC Developers' Summit*, Ottawa, Canada, May 2003.

[62] A. Pnueli. Temporal Semantics of Concurrent Programs. *Theoretical Computer Science*, 13(1):44–60, 1981.

[63] P. J. Ramadge and W.M. Wonham. Supervisory Control of a Class of Discrete Event Systems. *SIAM J. Control and Optimization*, 25(1):206–230, 1987.

[64] G. Rothermel and M. J. Harrold. Empirical Studies of a Safe Regression Test Selection Technique. *Software Engineering*, 24(6):401–419, 1998.

[65] J. Seyster, K. Dixit, X. Huang, R. Grosu, K. Havelund, S. A. Smolka, S. D. Stoller, and E. Zadok. Aspect-Oriented Instrumentation with GCC. In *Proc. of the 1st International Conference on Runtime Verification (RV 2010)*, Lecture Notes in Computer Science. Springer, November 2010.

[66] J.E. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, Jun, 2005.

[67] O. Spinczyk and D. Lohmann. The Design and Implementation of AspectC++. *Know.-Based Syst.*, 20(7):636–651, 2007.

[68] the wxTeam. wxwidgets, 2010. http://www.wxwidgets.org.

[69] Stony Brook University. GCC Open-Source Software Model-Checking Tool Kit. http://www.cs.sunysb.edu/~gmc.

[70] The LLVM compiler infrastructure. http://llvm.org.

[71] M. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 332–344, 1986.

[72] Q.-G. Wang, Z. Ye, W.-J. Cai, and C.-C. Hang. *PID Control For Multivariable Processes*. Lecture Notes in Control and Information Sciences, Springer, March 2008.

[73] H. Wong-Toi and G. Hoffmann. The Control of Dense Real-Time Discrete Event Systems. In *Proc. of 30th Conf. Decision and Control*, pages 1527–1528, Brighton, UK, 1991.

[74] Z. Yang, B. Al-Rawi, K. Sakallah, X. Huang, S.A. Smolka, and R. Grosu. Dynamic Path Reduction for Software Model Checking. In *Proceedings of iFM'09, the 7th International Conference on Integrated Formal Methods*, pages 322–336. Springer, February 2009.