

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

José Wesley de Souza Magalhães

**INSPEÇÃO AUTOMÁTICA DO ESTADO INTERNO DE PROGRAMAS EM
UM AMBIENTE NÃO COOPERATIVO**

Belo Horizonte
2021

José Wesley de Souza Magalhães

**INSPEÇÃO AUTOMÁTICA DO ESTADO INTERNO DE PROGRAMAS EM
UM AMBIENTE NÃO COOPERATIVO**

Versão Final

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Fernando Magno Quintão
Pereira

José Wesley de Souza Magalhães

**AUTOMATIC INSPECTION OF PROGRAM STATE IN AN
UNCOOPERATIVE ENVIRONMENT**

Final Version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Fernando Magno Quintão
Pereira

Belo Horizonte
2021

© 2021, José Wesley de Souza Magalhães.
. Todos os direitos reservados

Magalhães ,José Wesley de Souza

M188a Automatic inspection of program state in an uncooperative environment [manuscrito] / José Wesley de Souza Magalhães — 2021.
xvi, 58 f. il.

Orientador: Fernando Magno Quintão Pereira.
Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação
Referências: f.47-52

1. Computação – Teses. 2. Programas de computador – Inspeção – Teses. 3. Programas de computador – Verificação – Teses. 4. Compiladores (Programas de computador) – Teses. I. Pereira ,Fernando Magno Quintão. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. III.Título.

CDU 519.6*32 (043)

Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa
CRB 6ª Região nº 1510




UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO


FOLHA DE APROVAÇÃO

Automatic Inspection of Program State in an Uncooperative Environment

JOSÉ WESLEY DE SOUZA MAGALHÃES

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:


PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Orientador
Departamento de Ciência da Computação - UFMG


Prof. CHUNHUA LIAO
Centro de Computação Científica Aplicada - LNLL


PROF. RAFAEL DUEIRE LINS
Departamento de Informática - UFPE

Belo Horizonte, 15 de Outubro de 2021.

To my father, who taught me to be an honest person and who is in heaven looking after me

Acknowledgments

It is finally done. This work is the result of a great effort of research, seriousness, and perseverance, amidst the most varied emotions and tough times. I could not have done it alone, so I use this space to thank those who helped me.

First and foremost I thank God for having always been with me. Hearing me and blessing me, You gave me strength and did not let me disbelieve. I am also immensely grateful to my family, my mother Josefa and my brothers Thales and João Paulo. Thanks for the support, for the advice, for the fun, and for being my company in all those moments. You are the pillars of my world.

No matter how long or difficult be your journey, one needs to give the first step. I would like to thank Relva do Egypcto and Juliana for making my first steps pleasant and cozy. It is not easy to move from the countryside to a new and huge city, and you made me feel welcomed. I offer you my most sincere thanks.

I thank Fernando Magno for accepting me as one of your pupils and for being a better advisor than anyone could have thought. Always being helpful, present, willing to help, correcting me you taught me to be a real researcher. More than a Professor, you are a good human being. Thanks for your teachings, advice, and tips on good books. You were my Master Dohko. I would also like to thank Chunhua Liao for being my co-advisor during this work. Your expertise was essential for it to be completed well, and your tips taught me to perform my tasks more professionally.

I believe that being in a healthy and cooperative research environment is fundamental to perform a good job. I thank all the Dragons from The Nest who helped me. You were always willing to support and welcomed in the group. Besides, you are all experts in what you do. I wish we could have spent some more time together, but the pandemic did not allow it. But even remotely, your assistance was greatly appreciated by me.

Finally, to all those who helped me in some way and I did not name here, I feel grateful for your cooperation.

“We have just one world, but we live in different ones.”

(Mark Knopfler)

Resumo

O estado interno de um programa é formado pelos valores que tal programa manipula. Estes valores são armazenados na pilha de chamadas de funções, na *heap*, ou em memória estática. A habilidade de inspecionar o estado interno de um programa é útil para propósitos de depuração e verificação. Entretanto, não existe técnica geral para inserir pontos de inspeção em linguagens com um sistema de tipos não seguros, como C ou C++. A dificuldade vem da necessidade de percorrer o grafo de memória em um assim chamado ambiente não cooperativo. Nesta dissertação, uma técnica automática para lidar com esse problema é proposta. Nós introduzimos uma transformação estática de programa para reportar o seu estado interno. Nossa técnica foi implementada utilizando *Low Level Virtual Machine* (LLVM). É possível ajustar a granularidade dos pontos de inspeção, trocando precisão por desempenho. Nesta dissertação, nós demonstramos como utilizar pontos de inspeção para depurar otimizações de compiladores; para inserir código de verificação em benchmarks; e para visualizar estruturas de dados.

Palavras-chave: Pontos de Inspeção, Verificação, Estado Interno de Programa.

Abstract

The program state is formed by the values that the program manipulates. These values are stored in the stack, in the heap, or in static memory. The ability to inspect the program state is useful as a debugging or as a verification aid. Yet, there exists no general technique to insert inspection points in type-unsafe languages such as C or C++. The difficulty comes from the need to traverse the memory graph in a so-called uncooperative environment. In this dissertation, we propose an automatic technique to deal with this problem. We introduce a static code transformation approach that inserts in a program the instrumentation necessary to report its internal state. Our technique has been implemented in *Low Level Virtual Machine* (LLVM). It is possible to adjust the granularity of inspection points trading precision for performance. In this paper, we demonstrate how to use inspection points to debug compiler optimizations; to augment benchmarks with verification code; and to visualize data structures.

Palavras-chave: Inspection point, verification, program state.

List of Figures

1.1	An example program that builds two disjoint linked lists	19
2.1	Comparison of different techniques to find or prevent bugs in compilers. A formally verified compiler (C) is correct by construction [Leroy, 2009]. Program synthesizers such as <code>CSmith</code> [Yang et al., 2011] create programs (P) that are given as inputs to compilers to test them. Variations of P , e.g., P_1, \dots, P_n , can be created via program mutation [Le et al., 2014; Sun et al., 2016]. Translation validation [Lopes et al., 2021; Necula, 2000; Pnueli et al., 1998; Tristan et al., 2011a] checks that the compiler’s output, e.g., the compiled program $C(P)$, is correct. Our technique can be used to check the outcome of compiler optimizations (O). Given a compiled program $P_c = C(P)$, and its optimized version $P_o = O(C(P))$, inspection points let developers match the final state of P_c and P_o for any input I	23
3.1	A program with four static inspection points marked with letters A-D. . .	29
3.2	Dynamic contexts produced by the program in Figure 3.1, with the input “main a”.	30
3.3	Visible program states at two different dynamic inspection points from Figure 3.2. Symbol “ <code>retn</code> ” is the auxiliary variable that holds the return value of function <code>incAll</code> at Line 18 of Figure 3.2.	31
4.1	Global and stack tables for the program in Fig. 3.1. We show only the stack table for <code>main</code>	34
4.2	Auxiliary state at the seventh DIP in Figure 3.3. The keys in the type table are N = name; F = format; O = offset and B = base type.	35
4.3	A program with just one allocation of heap memory. <code>WHIRO</code> keeps a single entry in H and it always reports the same this same address regardless of how many program objects point to it.	36
4.4	A program with that defines a 2D matrix.	38

4.5	A program that allocates an array in the heap and creates pointers from the allocated address.	39
4.6	A program that reads a pointer past the last address of an array.	40
4.7	Program that defines a union type.	42
4.8	In this program, variable <code>b</code> would be shadowed. The value of the duplicated variable will be printed at the inspection point.	43
4.9	(Top) Program before scalarization. (Bottom) Program after scalarization.	44
4.10	A program with one heap allocation.	47
5.1	(a) Snapshot showing the two disjoint data structures in MIBENCH’s Patricia, after the first invocation of <code>pat_search</code> returns with the test input. (b-c) The two disjoint graphs in the heap of a program that copies a binary tree into a hash table. Collisions are stored in a linked list.	51
6.1	Time to instrument programs (in seconds). For reference, we provide the time of compiling each program with <code>clang -O0 -g</code>	54
6.2	Increase of execution time (with relation to the original program). Numbers in boxes denote the running time of the original program.	55
6.3	Increase in memory consumption (with relation to the original program). Numbers in boxes denote memory consumed by the original program, in Kilobytes.	56
6.4	Code growth (with relation to the original program). Numbers in boxes denote the number of LLVM instructions in the original program, compiled with <code>mem2reg</code> . Benchmarks are sorted by the size of the original program.	57
7.1	A program with type casts between a pointer and integer.	61
A.1	WHIRO framework overview	70

List of Tables

6.1	Time to run the benchmarks (in seconds) with and without counting the time to print inspection traces.	55
6.2	Relations between program features and code growth. We let <code>fst = FAST</code> and <code>prc = PRECISE</code>	58
6.3	Bugs reported by WHIRO.	59
6.4	Summary of results. Cmp : compilation time (Sec. 6.1). This column is the geometric mean of 16 ratios comparing the instrumentation time with the compilation time (<code>clang -O0 -g</code>); Exe : runtime overhead (Sec. 6.2); Mem : Memory consumption (Sec. 6.3); CG : code growth (Sec. 6.4). These three columns report geometric means of 16 ratios between instrumented and original program. Size : number of LLVM instructions in the 16 benchmarks.	59

List of Acronyms

DDR Double Data Rate

DIP Dynamic Inspection Point

ELF Executable and Linkable File

GCC GNU Compiler Collection

GDB GNU Debugger

LLVM Low Level Virtual Machine

RAM Random Access Memory

RTL Register Transfer Level

SIP Static Inspection Point

SMT Satisfiability Modulo Theories

SPEC Standard Performance Evaluation Corporation

SSA Single Static Assignment

Contents

1	Introduction	17
1.1	Motivation	18
1.2	Solution.....	20
1.3	Summary of Experimental Results.....	21
1.4	Next Chapters.....	21
2	Literature Review	22
2.1	Compiler Correctness	22
2.2	Garbage Collection in Uncooperative Environments	25
2.3	Shape Analysis	26
3	Core Definitions	28
3.1	Program State.....	29
4	The Whiro Framework	32
4.1	Tracking Program State	32
4.2	Information Retrieval.....	37
4.3	Implementation Decisions	40
4.4	Properties of the Memory Monitor	44
4.5	Customizations.....	46

5	Applications of Inspection Points	49
5.1	Debugging Aid	49
5.2	Adding Verification Outputs to Benchmarks	50
5.3	Data Visualization	50
6	Experimental Results	52
6.1	Compilation Overhead	53
6.2	Running Time Overhead	53
6.3	Memory Overhead	56
6.4	Code-Size Overhead	57
6.5	Effectiveness	58
6.6	Summary of Results	59
7	Conclusion	60
7.1	Limitations & Future Work	60
	Bibliography	63
	Appendix A Whiro Documentation	69
A.1	Overview.....	69
A.2	Methods	69

Chapter 1

Introduction

Regardless of the programming paradigm or language, a program describes how values are handled in order to do computations. Said values are stored in distinct memory regions e.g., static memory, stack or heap, which can be reached through program structures like pointers or activation records. The ensemble of the values that a program manipulates is called the state of a program. This state is not immutable, as during the execution a program performs different operations that may alter the contents of variables.

The ability to inspect the program state is useful for reasons including debugging, verification, and visualization [Brusilovsky, 1993]. The identification of the memory regions that constitute the program state is a solved problem for type-safe languages. In a type-safe language, the type of each variable can be determined, either statically or dynamically [Aho et al., 2006, Sec.7.5.1]. More specifically, those languages distinguish pointers from other types, which means that it is possible to accurately determine whether a data segment may point to a different memory position. The capacity of discovering which values in a program access memory facilitates the identification of program state. As testimony to this fact, such identification is the basis of mark and sweep garbage collectors [Zorn, 1990; Wilson, 1992].

However, identifying program state is difficult in type-unsafe languages. These languages include not only C, C++ and mainstream assemblies, but also the unsafe parts of otherwise type-safe languages such as Java, C# and Racket [Mastrangelo et al., 2015]. C and C++ are commonly known in the garbage collection community as *uncooperative* [Boehm and Weiser, 1988]. They own this qualifier to a weak type system, that neither associates size information with memory regions, nor distinguishes pointers from scalars. Also, in type-unsafe languages, it is possible to create new pointers by applying arithmetic operations on existing pointers or by casting from integers, for

example. This implies that no memory fragment can be safely released since it could still be accessed by ambiguous program elements [Aho et al., 2006, Sec.7.5.1]. Although it is still possible to implement garbage collectors for languages like C or C++ [Boehm, 1993; Henderson, 2003; Rafkind et al., 2009; Lee et al., 2020; Banerjee et al., 2020], such implementations are not mainstream. The more reliable these garbage collectors are, the heavier the overhead they bring.

1.1 Motivation

The first motivation of this work was towards compiler correctness. There are plenty of techniques to debug compilers and code optimizations, but testing is the most widely used method. Testing-based approaches have already confirmed hundreds of bugs in mainstream compilers such as GCC and LLVM [Le et al., 2014; Yang et al., 2011]. However, testing requires benchmarks. When used to find bugs in compilers, these benchmarks are programs with known behavior. Benchmarks can be curated from mainstream applications, like SPEC CPU [Henning, 2006] and DaCapo [Blackburn et al., 2006]. Usually, these benchmark suites have few programs, which causes many researchers to resource to benchmarks generated automatically by tools like CSmith [Yang et al., 2011] and LDRGen [Barany, 2017].

Nevertheless, regardless of their origin, in order to be useful, benchmarks must provide verifiable outputs. The verification of a benchmark is commonly performed by comparing the output of the benchmark with a reference result. This constraint makes it harder the automatic construction of benchmarks or to adapt general programs to be used as such, as normally a program does not provide a way to the user check if its output is correct.

In this scenario, a technique able to automatically create outputs for any program would be extremely useful to the research community in general, since this would greatly increase the number of benchmarks for several tasks. However, we noticed that our ideas could be extended not just to add verifiable outputs, but to report the entire internal state of a program at different points during the execution. Such approach allows us, for instance, visualizing the state of the memory and data manipulated by a program. This also increases the effectiveness of the use of our solution as a bug-catching tool, since we inspect more variables.

As an example, see the program at Figure 1.1. This program implements a function that recursively traverses a list duplicating it with the values added by 1. A possible reference output for this program is the final values of the variables at each

```

1  int numNodes = 0;
2
3  struct Node {
4      int data;
5      struct Node* next;
6  };
7
8  struct Node* create(const int data, struct Node* next) {
9      struct Node* p = (struct Node*)malloc(sizeof(struct Node));
10     p->data = data;
11     p->next = next;
12     numNodes++;
13     return p;
14 }
15
16 struct Node* incAll(struct Node *head) {
17     if (head) {
18         return create(head->data + 1, incAll(head->next));
19     } else {
20         return NULL;
21     }
22 }
23
24 int main(int argc, char** argv) {
25     struct Node *n = NULL;
26     struct Node *m = NULL;
27     if(argc > 1){
28         for (int i = 0; i < argc; i++) {
29             n = create(i, n);
30         }
31     }
32     if(n){
33         m = incAll(n);
34     }
35     return 0;
36 }

```

Figure 1.1. An example program that builds two disjoint linked lists

function. It is straightforward to report variables of primitive types and its derived types just by calling standard printing functions available in the programming language. Nevertheless, there are variables in this program which have user-defined types that do not have a specific format specifier. Consequently, the functions responsible to print data do not know how to handle such variables. Besides that, the program contains pointers that can reach other memory blocks that keep valid data, hence such data should be reported. Our solution can report all the values manipulated by the program in Figure 1.1 that are part of the valid state at any stage during its execution.

There exist other tools capable of observing the state of the program at different points. An example is the debuggers, like GDB [Stallman et al., 2002]. Yet, debuggers

require a lot of user interaction to observe values of variables, whereas our solution is fully automatic. In addition to it, our solution enables the user to explore not just the *Reachable* but the *Visible* program state. We explain that difference in Chapter 3.

1.2 Solution

We demonstrate that ideas previously used in the implementation of tracing-based garbage collectors for type-unsafe languages can be used to inspect program state in type-unsafe languages. With this goal in mind, we bring, from the systems community into the software engineering community techniques to create *Program Inspection Points*, a notion that we define in Chapter 3. Several useful applications emerge from this perspective: program synthesizers, debuggers and visualizers, to name a few.

Our approach is highly customizable. Whoever uses it can configure the granularity of inspection points, to encompass, for instance, either the return statement of every function, or the last statement of the program that is visible to the compiler. Similarly, state can be configured to include values stored in global, stack-allocated and heap-allocated variables, or any combination of them. The possibility to customize the amount of program state that is tracked is key to producing practical applications. For instance, to debug compiler transformations, we need to inspect every memory location affected by the program, regardless of its usage e.g., as a pointer or as a scalar, or its location. To visualize data structures, in turn, we need to traverse the graph formed by pointers to locations in the heap. And to synthesize verification code for benchmarks, we need to preserve their performance; for instance, printing only the state of local variables at the end of program execution.

We show in Chapter 4 that this customization is achievable within a unified framework, henceforth called WHIRO. Said framework relies on the compiler only—it does not depend on the operating system or the architecture. Thus, WHIRO can be used even in embedded devices that lack support of inspection tools like Valgrind [Nethercote and Seward, 2007] or GDB [Stallman et al., 2002]. To preserve performance, we move to compilation time as much computation as possible. To this end, WHIRO injects local variables in the program to track the state of values. To make it useful, we rely on the meta-information that the compiler creates to match memory locations with source-code names. To ensure soundness, the program is augmented with a global hash-table to track values whose address cannot be estimated at compilation time.

1.3 Summary of Experimental Results

We have implemented a pass that inserts inspection points in the *Low Level Virtual Machine* (LLVM) intermediate representation. Chapter 5 discusses three usages of this implementation: to debug compiler optimizations (Sec. 5.1), to visualize data structures (Sec. 5.3), and to insert verification code into benchmarks (Sec. 5.2). Chapter 6 evaluates the precision and the overhead of our implementation. At its maximum precision, our implementation keeps track of every memory address that has a corresponding allocation point in the program's source code: static, local and heap allocated variables, including aggregates such as arrays and structs. At this level, we observe an average slowdown of 1.34x on MIBENCH programs Guthaus et al. [2001], an increase of memory usage of 1.63x, and an increase of static code size of 1.48x. However, precision is customizable. For instance, when used to synthesize verification code for benchmarks, we observe no performance regression, an increase of memory usage of 1.07x, and an increase in code size of 1.06x.

1.4 Next Chapters

This dissertation is structured as follows. Chapter 2 explores some previous works from areas in which our work relates to. Chapter 3 brings some definitions that are important for the understanding of our approach. In chapter 4 we describe WHIRO, the framework that implements our technique. Chapter 5 provides three examples of possible applications of the proposed solution. Chapter 6 presents our experimental evaluation and in chapter 7 we conclude our work. We also provide some documentation about the main methods implemented in this work and an overview of our framework in Appendix A

Chapter 2

Literature Review

This work relates to some different research areas. In terms of purpose, the ability of inspecting program state supports the implementation of lightweight forms of program verification. In terms of implementation, the techniques discussed in this paper are heavily inspired by previous work on tracing-based garbage collection for type-unsafe languages. As we will state in section 4.5 (property 7), our technique can also be viewed as a dynamic version of shape analysis. In the rest of this chapter, we discuss how our work fits in these different subfields of the programming languages literature.

2.1 Compiler Correctness

The area of program verification is immense. Although our work does not deliver formal guarantees about the behavior of programs, it helps developers to debug compilers in at least three ways: (i) automatically producing outputs for benchmarks; (ii) revealing program state to developers; and (iii) checking the behavior of compiler optimizations. Figure 2.1 outlines where such purposes fit in the broader area of program verification.

Solutions to assert the correctness of a compiler and code transformations range from formal methods to testing, each one using different principles towards correctness. Formal verification methods have been used to prove the correctness of an optimizing compiler. The most notable work in this field is `CompCert` [Leroy, 2009], which translates code from a subset of C to PowerPC assembly code. [Mullen et al., 2016] have extended that work to verify peephole optimizations. Another example of formal verification is `CakeML` [Kumar et al., 2014], a compiler which supports a subset of Standard ML and uses first-order logic to check if an input program meets the semantic specifications of the compiler.

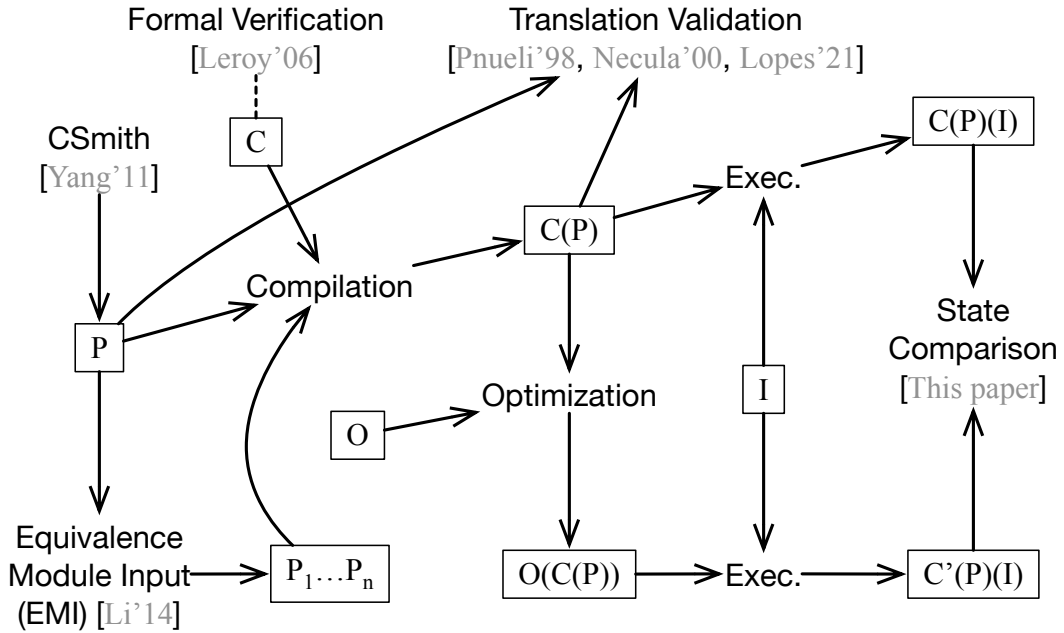


Figure 2.1. Comparison of different techniques to find or prevent bugs in compilers. A formally verified compiler (C) is correct by construction [Leroy, 2009]. Program synthesizers such as `CSmith` [Yang et al., 2011] create programs (P) that are given as inputs to compilers to test them. Variations of P , e.g., P_1, \dots, P_n , can be created via program mutation [Le et al., 2014; Sun et al., 2016]. Translation validation [Lopes et al., 2021; Necula, 2000; Pnueli et al., 1998; Tristan et al., 2011a] checks that the compiler’s output, e.g., the compiled program $C(P)$, is correct. Our technique can be used to check the outcome of compiler optimizations (O). Given a compiled program $P_c = C(P)$, and its optimized version $P_o = O(C(P))$, inspection points let developers match the final state of P_c and P_o for any input I .

Another widely used approach to debug compilers is testing. Hundreds of bugs have been reported by testing-based approaches in mainstream compilers, like GCC and LLVM [Le et al., 2014; Yang et al., 2011]. Testing requires benchmarks, and in order to be useful such benchmarks must present known behavior. Besides benchmarks from mainstream applications, like SPEC CPU ([Henning, 2006] and DaCapo [Blackburn et al., 2006]), researchers also resort to synthetically generated programs. Program generation for testing has been the focus of many works and several techniques have emerged to automate this process. Such programs can be generated strictly based on the grammar of a language [Amodio et al., 2017; Lindig, 2005], or by adding some context to that grammar [Yang et al., 2011; Alipour et al., 2016; Morisset et al., 2013]. Other solutions do not based on the grammar of the language, e.g. the works by [Eide and Regehr, 2008] and [Nagai et al., 2012], which first create constructs of the language and then

generate the remaining of the program, and the work by [Zhang et al., 2017], which introduces skeletal program enumeration. Testing benchmarks can also be obtained by program mutation, whether preserving the semantics of the original programs [Le et al., 2014; Sun et al., 2016] or not [Nagai et al., 2014].

There exist debugging techniques which use a combination of domain-specific languages to encode optimizations and an Satisfiability Modulo Theories (SMT) solver to prove them correct [Lopes et al., 2015; Menendez et al., 2016]; and techniques which try to catch and insert code to prevent an program to stop running in case of an error occurs [Sidiroglou-Douskos et al., 2015], however the latter is restrict to some categories of faults, e.g., out-of-bound access and integer overflows.

Verifying the correctness of a compiler as a whole, i.e., checking whether the compiler correctly optimizes every possible input program is a difficult and time-consuming task. The technique of translation validation [Pnueli et al., 1998] consists in certifying the correctness of each module produced by a compiler individually by comparing both the original and the optimized program. This approach is similar to this work in that it compares different versions of each program after it has been transformed. Translation validation has already been proved useful and has been applied in different domains, including mainstream compilers, such as GCC [Necula, 2000; Sewell et al., 2013] and LLVM [Tristan et al., 2011b; Lopes et al., 2021], high-level synthesis [Tun Li et al., 2013; Li et al., 2013; Kundu et al., 2010], code generation [Ryabtsev and Strichman, 2009; Leung et al., 2015], and software security [Zhang et al., 2011]. Our work is similar to translation validation in the sense that it performs individually for each program, and the states are compared for each transformation applied.

Verification of Benchmarks' Outputs With the rise of automated benchmark generators, program checking is increasingly the target of research. In this scenario, such generators must provide some method for validate its benchmarks. CSmith [Yang et al., 2011] uses a checksum mechanism to verify the output of the benchmarks generated by it. This checksum consists of the values of global non-pointer variables at the end of execution, and it must be unique for a same valid input, regardless of the compiler or optimizations used on a program. It roughly corresponds to the lowest level of granularity that we provide: inspection at the end of the program based on statically allocated variables. The work by Richards et al. [2011] also inserts verification code into synthesized JavaScript programs. Their instrumentation enables the recording of program states at each stage of its execution. In contrast to our work, these techniques are tightly coupled with program synthesis. Verification is inserted onto synthetic programs, at the time these programs are produced; not in general code, like we are able to

accomplish. Yang et al. [2015] developed an approach which automatically instruments Register Transfer Level (RTL) modules to assist in debugging of high-level synthesis tools. Their technique is based on just-in-time traces to gather the expected values for the operations in the program and use this trace to insert verification code during the generation of the RTL module, including information about the equivalence between the RTL and the LLVM IR of the program. However, the set of programs that can be successfully verified by that technique is a way simpler than ours, given that the authors restrict themselves to a specific set of instructions that they can verify. Our work improves upon said approach, since we are able to handle a larger universe of programs.

2.2 Garbage Collection in Uncooperative Environments

C and C++ do not contain a standard garbage collector. The aspect that most hinders the existence of native garbage collection for those languages is the fact that they are unsafe languages: the type systems of these languages do not prevent values of numeric type from being treated as pointers. Hence we can, for example, compute addresses with arithmetic operations. Nevertheless, different research groups have designed and implemented garbage collectors for them, and this work was inspired by such implementations. Conservative garbage collectors do not have full information about pointer locations, therefore they assume that any pattern in memory that could be a pointer is in fact a pointer [Boehm, 1993], thus it can reach an object. The most well-known conservative technique for garbage collection for C is the work by Boehm et. al [1988; 1993]. It is a tracing-based algorithm that records any invalid pointer discovered during the collection. Those ideas have been extended along different directions more recently. Henderson [2003] keeps a list of variables that might contain pointers within to achieve a more accurate image of the C stack. The work by Rafkind et al. [2009] describes a garbage collector for C that is precise, in the sense that it is able to distinguish integers from pointers. Current research efforts are still performed in this area. For example, the work by Lee et al. [2020] proposes a new garbage collector focusing on non volatile memory heap objects; and Banerjee et al. [2020] present a technique to perform this collection in a sound manner. The key idea behind their solution is to track the provenance of the pointers, to accurately compute the set of all program locations from which a pointer can derive.

It is not trivial to verify heap-allocated data, since they may contain pointers

to other data, and pointers in C do not keep information that describes the data being pointed to, unlike other languages, e.g., Fortran, in which one can access such information through a pointer [Chen et al., 2016]. The heap-checking performed by our tool is similar to those tracing-base garbage collectors. A tracing-based garbage collector starts from a set of root objects and scans all the objects which are reachable from this root set. *Heap Tracing* is also how we collect heap state. This work keeps a image of the heap at every function call, which contains addresses allocated in the heap. Using this image, we perform a depth-first-search going over the chain of references reachable from this image, printing the contents of each variable found. Our approach is neither over-conservative, nor relies on static analyses, and it improves the conservative garbage collectors techniques, since we can identify precisely every allocated pointer. However, the price for this precision is efficiency: our tracing technique would not be a viable alternative to garbage collection in an uncooperative environment, because its global table imposes a prohibitively large overhead on programs, as we shall see further in this dissertation (section 6.2).

2.3 Shape Analysis

The usage of pointers is one important aspect of imperative languages. Pointers increase the performance of operations such as iterating and are useful to pass large data structures as function arguments. However, this usage is error-prone, and making massive use of pointers may hinder program understanding and debugging. Besides that, the possibly far reachability of pointers makes program dependence analyses harder, which causes possible loss of opportunities for optimizing. Shape analysis is a static program analysis that aims to provide some properties of the heap-allocated data that a program handles. The results bring valuable information for program debugging, parallelization, and optimization [Wilhelm et al., 2000]. For example, a shape analysis algorithm may assert the type of data structure that constitutes the output of a program [Sagiv et al., 1998] or confirm that a given program is free of some types of memory management errors [Dor et al., 2000].

Nevertheless, static analyses need to summarize the execution of a program and consequently they do not depict the real extension of pointer's reachability. For recursive programs, summarizing procedure calls is also necessary [Rinetzky and Sagiv, 2001]. Similar to the idea of shape analysis, our work can provide a view of the format of the heap-allocated data in a program, but we generate said view dynamically and at different points during the execution. Due to the heap tracing, we can analyze recursive

data structures and validate information about the heap state without the limitation of static bounds.

Chapter 3

Core Definitions

The goal of this dissertation is to let users observe the state of the memory manipulated by a program at different points of its execution. To illustrate how our contribution works, we will be using the program seen in Figure 1.1. In this example, the nodes of the tree are allocated in the heap, while the variable that controls the loop is in the stack of function *main*, and the variable *numNodes* is static.

In the rest of this chapter, we define the concepts that serve as the basis for our solution, starting with the notion of a *Static Inspection Point*.

Definition 1 (Static Inspection Point – SIP) *Given a program P written as a sequence of statements, a static inspection point is a point following a statement of P .*

Example 1. At maximum granularity, we recognize four static inspection points in the program in Figure 1.1, which, have been marked as A, B, C and D. The points are shown in Figure 3.1

WHIRO observes the variables at the different calling contexts that might exist during the execution of a program. At runtime, each time the program traverses a SIP, values are reported at *Dynamic Inspection Points*, which we define as follows:

Definition 2 (Dynamic Inspection Point – DIP) *A dynamic inspection point is a static inspection point, plus a calling context when that point executes. The calling context of a given invocation of a function f is determined by the list of functions active when that activation of f took place.*

Example 2. If we execute the program in Figure 3.1 as “main a”, then we shall observe the eight dynamic inspection points seen in Figure 3.2.

```

1  int numNodes = 0;
2
3  struct Node {
4      int data;
5      struct Node* next;
6  };
7
8  struct Node* create(const int data, struct Node* next) {
9      struct Node* p = (struct Node*)malloc(sizeof(struct Node));
10     p->data = data;
11     p->next = next;
12     numNodes++;
13     return p;
14 }
15
16 struct Node* incAll(struct Node *head) {
17     if (head) {
18         return create(head->data + 1, incAll(head->next));
19     } else {
20         return NULL;
21     }
22 }
23
24 int main(int argc, char** argv) {
25     struct Node *n = NULL;
26     struct Node *m = NULL;
27     if(argc > 1){
28         for (int i = 0; i < argc; i++) {
29             n = create(i, n);
30         }
31     }
32     if(n){
33         m = incAll(n);
34     }
35     return 0;
36 }

```

Figure 3.1. A program with four static inspection points marked with letters A-D.

Notice that from Definition 2, the same static inspection point might yield a large number of different dynamic inspection points. In face of recursion, a potentially infinite number of different DIPs is possible.

3.1 Program State

At each one of the dynamic points, WHIRO produces an image of the program state, relating memory locations with the names of variables in the source code of the program. We define as *state* the set of values of local, static and heap-allocated variables

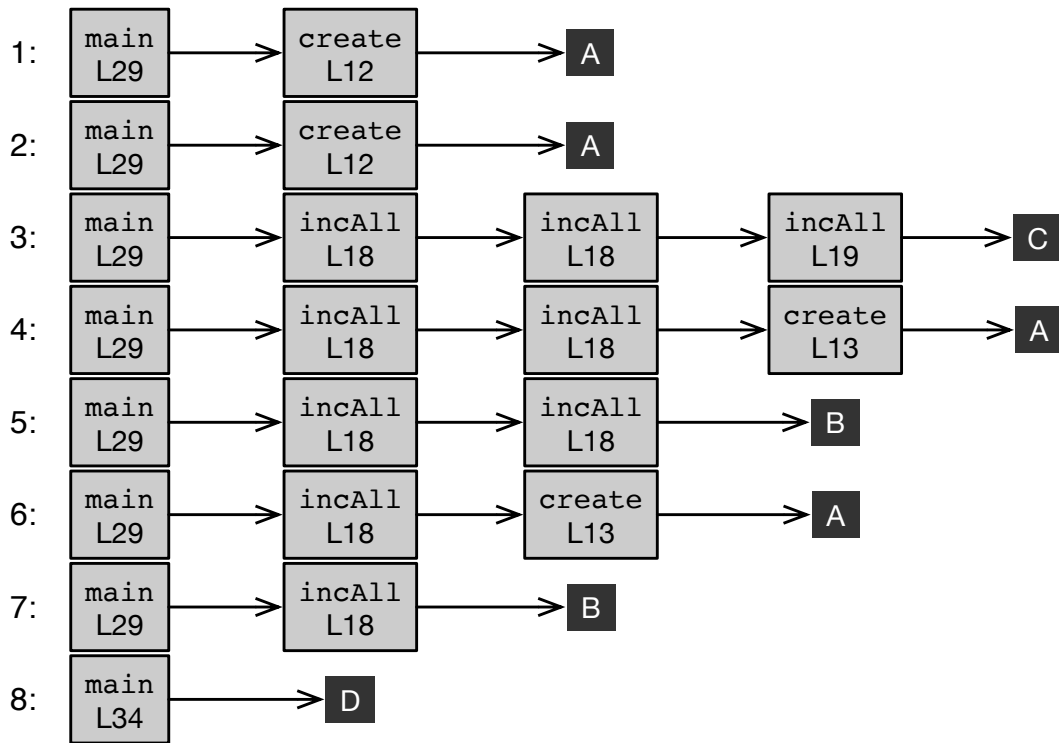


Figure 3.2. Dynamic contexts produced by the program in Figure 3.1, with the input “main a”.

of a program at a given dynamic inspection point. The *Visible Program State* is the subset of the program state that is reachable from variables visible at the scope of that dynamic inspection point:

Definition 3 (Visible Program State) *The visible state of a program at a dynamic inspection point p is a map from the program symbols visible at the scope of p to values. Program Symbols are the names of program variables.*

The notion of visible state differs from the notion of *Reachable State* commonly adopted in the description of tracing-based garbage collectors. The reachable state of a program (see, for instance, Aho et al. [2006, Sec.4.6.6]) is formed by memory allocated statically or on the stack, plus the memory in the heap referenced by any value in the reachable state (including the heap itself). The visible state differs with regards to stack allocated variables. Only variables in the scope of the active function (the function on the top of the call stack) are considered visible.

Example 3. Figure 3.3 shows the visible program state at two dynamic inspection points during the execution of the “main” function from Figure 3.1, with string “a” as input.

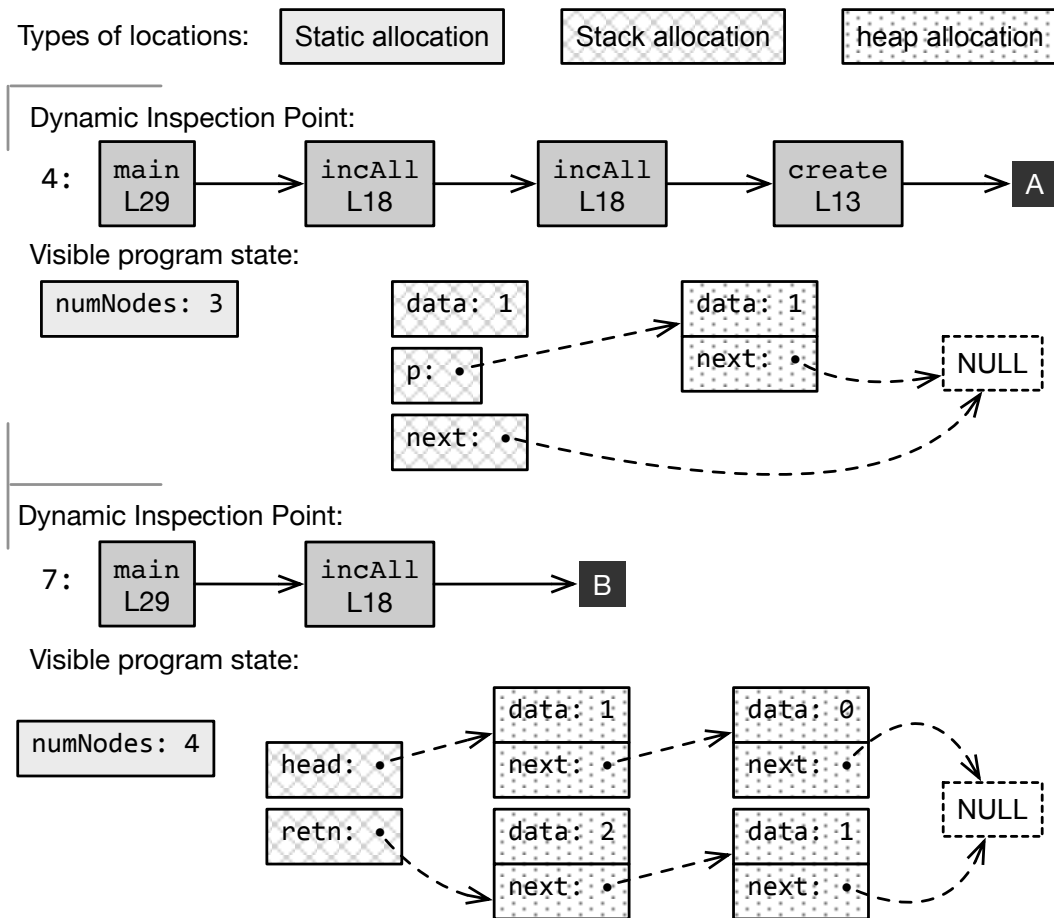


Figure 3.3. Visible program states at two different dynamic inspection points from Figure 3.2. Symbol “retn” is the auxiliary variable that holds the return value of function `incAll` at Line 18 of Figure 3.2.

The reason to distinguish visible from reachable state is pragmatic. The difference between visible and reachable states does not compromise WHIRO’s ability to inspect values created by recursive functions. Variables that are not in scope cannot be modified (within the defined semantics of C, for instance [Hathorn et al., 2015]). Once these invisible variables become active, e.g., when their activation record reaches the top of the calling stack, they can be inspected. Nevertheless, the static memory and the entire heap (including parts reachable only via invisible variables) are still tracked.

Chapter 4

The Whiro Framework

The solution proposed in this dissertation is presented as a framework called WHIRO. In this chapter, we describe the main features of said framework. In section 4.1 we define the main data structure used by WHIRO to track the program state. Section 4.2 details how we inspect variables in inspection points. Section 4.3 exemplifies some of the engineering decisions that have been taken during the implementation of our solution. Section 4.4 discuss the properties that arise from our implementation. Finally section 4.5 shows the different modes of usage of WHIRO.

4.1 Tracking Program State

To track the visible program state, we use a data structure henceforth called *memory monitor*, defined as follows:

Definition 4 (Memory Monitor) *The memory monitor is a data structure given by a tuple (G, S, H, T) , such that:*

G, S : Maps of global or stack SYMBOLS to (METAVAR, TRACE)

H : Set of HEAP ADDRESSES

T : Set of TYPE DESCRIPTORS

Where:

METAVAR *consists of name and type of a variable;*

TRACE *is List of (PROGRAMLOC, SSADEF);*

SSADEF is the definition point of some expression in an Single Static Assignment (SSA)-form program;

PROGRAMLOC is a program point, i.e., a program instruction or declaration in the LLVM intermediate representation.

TYPE DESCRIPTOR is a metadata that specifies the name and the format specifier of a type in the source code.

In Definition 4, G maps global variables to debugging information. S is analogous, except that it maps automatic variables to debugging information. There exists one table S per program function—each table stores information related to variables in the scope of a function. H is the set of addresses of memory blocks allocated in the heap. T holds metadata describing every type in the target program.

Static Components of the Memory Monitor. The G (global) and S (stack) components of the memory monitor exist only at compilation time. They do not exist when the target program executes. Their goal is to guide the insertion of instrumentation code in the program. Said code serves two purposes: (i) report program data to the user; and (ii) update the heap map H . The following parsing events cause updates of these structures (at instrumentation time):

1. Declaration of global variable v at location l : an entry for v is created in G . The pair $(l, def(v))$ is appended at $\text{Trace}(v)$, in the entry $G[v]$. Here $def(v)$ is the SSA expression that encompasses the address of v .
2. Declaration of a local variable v within function f : an entry for v_f is created in S .
3. Assignment $v = u$ to a variable v local to function f at location l . The pair $(l, def(u))$ is appended at $S_f[v]$.

As mentioned before, $def(v)$ for a global variable v represents its address. The addresses of global variables are determined at compilation time. Such addresses are visible in every program point, thus the assignments to global variables in the program are not tracked. In fact, $(l, def(v))$ is the only element in the $\text{Trace}(v)$ in case of static variables. To inspect a variable statically allocated the Memory Monitor always uses the address definition.

Example 4. Figure 4.1 shows S and G for the program in Figure 3.1. G contains data for global variable `numNodes`. Table S_{main} refers to function `main`. We omit the other

two stack tables, e.g., S_{create} and S_{incAll} , to save space. The variable i is not kept in the S_{main} map because it does not exist at the inspection point of `main`.

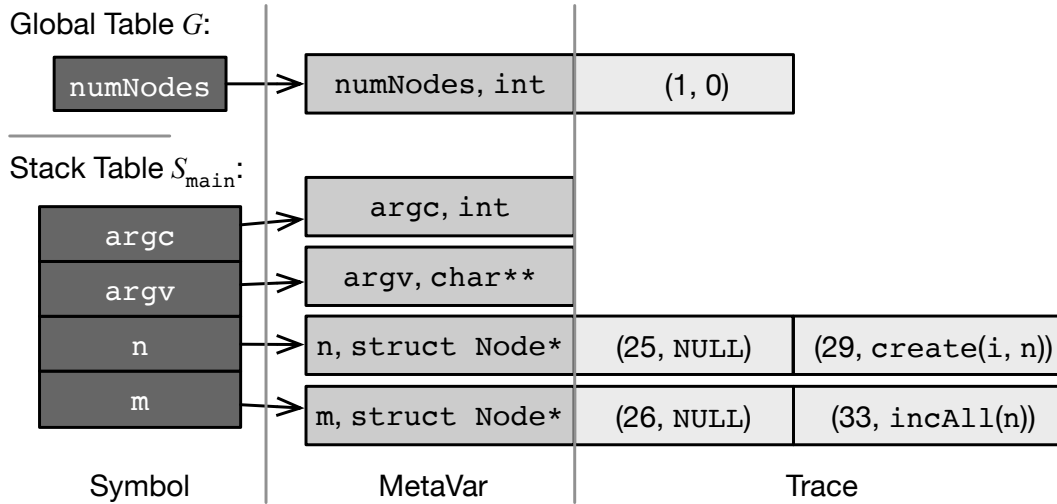


Figure 4.1. Global and stack tables for the program in Fig. 3.1. We show only the stack table for `main`.

Dynamic Components of the Memory Monitor. The heap table H and the type table T exist during program execution. We call these two tables the *Auxiliary Program State*. Table T is constructed statically and is immutable. The instrumented version of the program reads T and maintains it during the entire execution. It contains *type descriptors*. Descriptors are *flyweights*, meaning that there exists one per type in the program’s source code. T lets WHIRO print non-scalar variables, for it associates every type with an output format. It also allows WHIRO to navigate the program’s visible state, indicating fields of aggregate types that are pointers. For product types (e.g., C-like `structs`) the descriptor consists of the name, format, and the offset of each field. For pointers and qualified types (`const`, `volatile`, etc), the descriptor also contains an index to access the descriptor of the base type. This recursive nature can represent any composite type.

Example 5. Figure 4.2 shows the type table for the program earlier seen in Figure 3.3. The table contains seven entries, one for each type used in Figure 3.3, namely: `int`, `struct Node*`, `struct Node`, `const int`, `char**`, `char*` and `char`.

The *Heap Table* H , in contrast to the type table, changes during the execution of the program. Instrumentation inserted in the target program updates H . Each entry in the heap table contains an index to a descriptor in the type table. H also keeps

Type Table T :

0	1	2	3	4	5	6
int	pointer	Node	const	pointer	pointer	char
N: ""	N: ""	N: "data"	N: ""	N: ""	N: ""	N: ""
F: %d	F: %p	F: %d	F: %d	F: %p	F: %p	F: %c
O: 0	O: 0	O: 0	O: 0	O: 0	O: 0	O: 0
B: 0	B: 2	B: 0	B: 0	B: 5	B: 6	B: 6
		N: "next"				
		F: %p				
		O: 8				
		B: 2				

Heap Table H :

B1	B2	B3	B4
Type: 2	Type: 2	Type: 2	Type: 2
Size: 1	Size: 1	Size: 1	Size: 1
Free: 0	Free: 0	Free: 0	Free: 0
Vis: 0	Vis: 0	Vis: 0	Vis: 0

Figure 4.2. Auxiliary state at the seventh DIP in Figure 3.3. The keys in the type table are N = name; F = format; O = offset and B = base type.

track of freed heap addresses, which are considered unreachable data. If a freed address is re-allocated by the memory manager, the memory monitor sets the corresponding entry in H as reachable again, and updates the reference to T according to the type being allocated. The following events cause the heap table to be updated (at running time):

1. Memory is allocated, e.g., $v = \text{malloc}(\text{Tp})$ ¹: the monitor creates an entry $H[v]$ (if one does not exist), sets it as reachable and associates this descriptor with $T(\text{Tp})$.
2. Memory block is resized, e.g., $v_2 = \text{realloc}(v_1, \text{size})$: the monitor updates the size of $H[v_1]$ to size .
3. Memory is freed: the monitor sets $H[v]$ as unreachable.

¹We use the same assumption as Banerjee *et al.* [2020]: heap-allocated addresses only come out as the return value of particular functions (e.g malloc in C).

Only objects initialized with memory allocation functions have their info stored in the Heap Table. WHIRO exposes the same object in case there are many program symbols pointing to it.

```

1 int main{
2   int* a = malloc(16);
3   int* b = a;
4   float* c = (float*)a;
5   int* d = a+1;
6   return 0;
7 }
```

Figure 4.3. A program with just one allocation of heap memory. WHIRO keeps a single entry in H and it always reports the same this same address regardless of how many program objects point to it.

Example 6. In the program in Figure 4.3, there will be only one entry in H , referring to an array with 16 bytes. The same value reported when inspecting a or b , i.e., a hash code value built from the contents of an array with 16 bytes, as section 4.2 explains. If heap tracking is disabled, then the contents of $a[0]$ and $c[0]$ are printed. In this case, the former is printed as an integer, and the latter as a single-precision floating point number. Variable d will be reported as a pointer to a single byte.

Currently, WHIRO has special treatment for functions from the C standard library that manage memory (`calloc`, `free`, `malloc`, `realloc`, `reallocf`, `valloc` and `aligned_alloc`). It is important to emphasize that these functions have not been modified: WHIRO only understands that they perform memory allocation and deallocation. To use a different allocator, this semantic would have to be encoded into the implementation of WHIRO. Only one of the source files would have to be modified, but this modification requires knowledge of WHIRO’s implementation.

Example 7. Figure 4.2 shows the auxiliary state after the program in Figure 3.1 executes the inspection point B for the second time. H contains four heap-allocated blocks corresponding to the nodes of two linked lists. Each block contains an index to access the type table; hence, WHIRO is able to find the type of memory chunks. The “Vis” column indicates whether that block was visited when traversing the heap graph, as section 4.2 shall explain.

4.2 Information Retrieval

WHIRO inserts code at inspection points to retrieve the visible program state. Data is printed as either a textual log, or as a DOT graph (see Section 5.3). The rest of this section explains how WHIRO retrieves this information.

Inspection Traces. WHIRO instruments SSA-form programs [Cytron et al., 1989]; hence, for each source variable v there might exist several SSA definitions, henceforth called $Ins(v)$. When the inspection point is logged, only one $Ins(v)$ is valid. Given a function f , for each entry v in the global map G or in the function map S_f , WHIRO finds the valid $Ins(v)$ as follows:

1. If a definition $d(v) \in \mathbf{Trace}(v)$ is an address in the stack of f , $Ins(v) = d(v)$
2. If a definition $d(v) \in \mathbf{Trace}(v)$ resides in the same basic block as the current inspection point being created, $Ins(v) = d(v)$. If there are multiple definitions that meet this criterion, we choose the last.
3. If a definition $d(v) \in \mathbf{Trace}(v)$ dominates the inspection point, $Ins(v) = d(v)$. In case of multiple dominators, we choose the most immediate.

In SSA-programs it is also possible that a single definition is associated with different source code variables. We rely on the debugging information generated by the compiler to build this relation between source variables and SSA expressions. However, such associations might disappear, for instance, after some optimization. In this case, we will not be able to report information about some variable in the source code. We have faced such problem due to scalarization. In this case, we had to preserve meta-information ourselves. Example 13 explains how WHIRO deals with this.

Extending Live Ranges. An inspection point reports the state of all the automatic variables declared in the function where the SIP (Def. 1) exists. A problem ensues if a variable v is not alive at the SIP, and the SIP can be reached through multiple SSA definitions of v . In this case, WHIRO would not know which $Ins(v)$ to use. To deal with this issue, WHIRO inserts, at the SIP, a ϕ -function joining all the definitions of v that reach that program point.

Example 8. In Figure 3.1, n and m are promoted from the stack to virtual registers. Two definitions of n can reach point D, coming from lines 25 and 29. Similarly, definitions of m can reach D coming from lines 26 and 33. WHIRO shall create two ϕ -functions in the basic block that contains D, each redefining one of these variables.

Function Counters and Calling Context. The names reported in an inspection point are defined statically. Nevertheless, DIPs (Def 2) run in different calling contexts. Hence, the same name may be associated with different values during program execution. WHIRO distinguishes data at DIPs by the calling context. A static counter is associated with every instrumented function. Said counter is incremented each time the function is called. Thus, when reporting the program state, the current value of the function counter is also informed.

```

1  int** foo(int M, int N) {
2  int **x = (int**)malloc(M * sizeof(int*));
3  for (int i = 0; i < M; i++) {
4  x[i] = (int*)malloc(N * sizeof(int));
5  for (int j = 0; j < N; j++) {
6  x[i][j] = j;
7  }
8  }
9  // Inspection point here
10 return x;
11 }

```

Figure 4.4. A program with that defines a 2D matrix.

Dealing with Multi-Dimensional Arrays. WHIRO can print the contents of arrays of primitive types; however, in its default mode, WHIRO produces a hashcode that summarizes the data stored within them. For arrays of aggregate types, WHIRO inspects each cell individually. To find the hash of an array v of primitive types, WHIRO still traverses v entirely, even if the array is multidimensional. It is to note that a change in any array position yields a different hashcode; hence, hashing does not harm the use of WHIRO as a debugging tool. If the dimensions of the array are determined by constants known at compilation time, then no additional instrumentation is inserted in the program to permit this traversal: the necessary loop is hardcoded at compilation time. Otherwise, WHIRO inserts instructions to compute those values at runtime. These values are kept in the S table.

For arrays of pointers, WHIRO treats each contiguous block of memory independently, so as long as the type of an array cell is declared with a pointer qualifier, WHIRO can track it. This is possible because size information is stored in the Heap Table H , and type information is preserved in the table T . The combination of these two tables also lets us handle arrays that contain structure types, even if said structures contain

pointers to other arrays.

Example 9. The program in Figure 4.4 defines a 2D matrix whose root is declared as `int **`. WHIRO would print individual hash codes for `x[0]` and `x[1]`, because array `x` has cells that are pointers themselves.

Traversing the Heap Graph. The blocks allocated in the heap form a graph. Edges exist whenever a block contains a pointer to another heap address. One of the inspection modes of WHIRO traverses this graph (see Section 4.5) in Depth-First-Search fashion. To avoid cycles, WHIRO adds a bit to every node in the heap table H , indicating if that node has been visited. Tracing-based garbage collectors implement similar approach [Wilson, 1992; Zorn, 1990]. To deal with aliasing, we set every node as unvisited right after the traversal ends. In short, WHIRO guarantees that every byte that has been allocated (and not freed) is traversed at least once if this memory is reachable.

Reporting Example. Here we show an example of report generated by WHIRO. Consider the program in Figure 4.5:

```

1 int main() {
2   char* c = malloc(16);
3   long *d0 = (long*)(c+8);
4   *d0 = 42;
5   long d1 = (long)*(c+8);
6   char *c1 = (char*)&d1;
7   printf("%ld\n", d1);
8   return 0;
9 }
```

Figure 4.5. A program that allocates an array in the heap and creates pointers from the allocated address.

Using maximum precision, WHIRO prints, for this program:

```

*c main 1: 1536899735 // Hash of a 16-byte array
*d0 main 1: 42 // Value stored in an array of type long[1]
d1 main 1: 42 // Value of a variable of type long
```

When inspecting variable `c`, WHIRO checks the Heap Table and sees that the size of the corresponding entry is 16 and the type is a scalar, so it computes a hashcode. It does not find `d0` in the Heap Table and there's no size information associated with it, so it inspects `d0` according to its source type, i.e., a pointer to `long`. WHIRO does not

print anything related to `c1`, as this pointer is never used, after being initialized. It is worth to emphasize that when reporting floating pointer values WHIRO rounds them up to two decimal places.

4.3 Implementation Decisions

While developing WHIRO, we took a series of engineering decisions (also known as “hacks”) to ensure that the framework could handle general C/C++ programs. These decisions, in our opinion, are not necessarily of scientific interest; however, from a coding standpoint, they might hold some consequence. In this section, we go over some of these decisions.

Reading one element past the end of an array The C Standard allows a comparison between a pointer and the first address past the end of an array ([Schildt, 1990, 6.5.6/8]). However, this element cannot be dereferenced. Referring to the next address past the last address of an array is a common approach to implement C++ iterators, for instance. Example 10 shows a program that uses this trick. This possibility poses a problem to WHIRO, for our implementation cannot know if a pointer is valid or not. If the pointer is mentioned in the source code, this pointer will be printed as part of the program’s visible state, once information is retrieved from the inspection point. In this case, undefined behavior might ensue.

Example 10. Figure 4.6 shows a program that reads one element past the last address of an array to iterate through the array. Pointer `e` cannot be dereferenced; however, it can be compared against another pointer.

```
1 int main() {
2     int v[4];
3     int *i;
4     int *e;
5     for (i = v, e = v + 4; i < e; ++i) {
6         *i = 1;
7     }
8     return 0;
9 }
```

Figure 4.6. A program that reads a pointer past the last address of an array.

WHIRO does not keep a list of valid memory blocks. Keeping this list would be overly expensive at runtime, for every memory address would have to be tagged. Thus, if an invalid address is mentioned in the source code, it will be printed as part of the data in a dynamic inspection point. Two problems emerge from this shortcoming. First, debuggers, like the one discussed in Section 5.1, might try to compare these pointers, and the result of this comparison is meaningless. Second, when tracking heap data, WHIRO might jump to whatever address it recognizes in such a pointer—possibly incurring into a segmentation fault. Our implementation does not try to deal with the first problem: it is up to WHIRO’s users to handle false positives. To deal with the second problem, WHIRO only prints pointers within specific Executable and Linkable Format (ELF) program segments: `stack`, `heap`, `bss`, `data`, and `text`. The boundaries of these segments are given by global variables. This solution works for ELF binaries, but it is not portable across different formats.

Dealing with Union Types Programming languages like C and C++, or the LLVM intermediate representation, provide users with “union types”. These types are sets formed by the union of two other types. Union types in C and C++ are defined by the `union` key word. Unfortunately, these unions are not tagged, as it happens, for instance, with datatypes in functional programming languages like Haskell and SML/NJ. Therefore, at runtime, WHIRO cannot know what is the intended meaning of a union type: any of its composing types could be a valid representation. Example 11 illustrates this issue.

Example 11. Figure 4.7 shows a program written in C that defines a union of two types; integers and 32-bit floating point numbers. A static reaching definitions analysis will inform us that two definitions of the union can reach Line 9: either the one at Line 5 or the one at Line 7. Yet, dynamically, only the definition at Line 5 ever reaches Line 9.

To allow the comparison between union types across two different versions of the same program, WHIRO prints them as bitmaps. In other words, the inspection of a value declared as a union type always yields the bitmap representation of that type, using its largest composing part. Similarly, whenever a union type is inserted into the heap table H , it is stored as its largest constituent, even if that is not its intended meaning.

Shadowing Stack Variables As mentioned in section 4.2, WHIRO reports all the variables local to the function where the inspection point is defined. For variables

```

1 union element { int i; float f; };
2 int main(int argc, char** argv) {
3     union element e;
4     if (argc) {
5         e.i = 4;
6     } else {
7         e.f = 0.25;
8     }
9     return e.i;
10 }

```

Figure 4.7. Program that defines a union type.

which are not alive at said point, we extend their live range by injecting ϕ -functions in the program. However, in some cases there are no definitions of a variable v from a function f in the predecessors of the inspection block. In this case, extending the live range of v might involve the creation of many ϕ -functions from the available definitions of v until the inspection point—some of them with UNDEF values. To simplify this process, WHIRO *shadows* v to correctly track its value.

Shadowing a variable means duplicating in memory the value of that variable. The duplicate must be updated whenever the original variable suffers an assignment. To implement shadowing, we allocate a slot $shadow(v)$ in f 's activation record. This slot has the same type as v . Then the trace of v is traversed and the memory monitor injects code in f to store all the definitions in $shadow(v)$ right after they are computed. Since $shadow(v)$ is created at the entry point of f with a special "undefined" value, the shadow value is visible in all the basic blocks of that function. Therefore, that value can be read at any point within f . Example 12 illustrates this modus operandi.

Example 12. Variable `b` in Figure 4.8 belongs into the local scope of function `main`. This variable is assigned at two different program points which reach the static inspection point at Line 11. Furthermore, its declaration point, with an undefined value, can also reach Line 11. Inspecting the value of `b` at Line 11 would require the insertion of three ϕ -functions in the program (between Lines 6-7, 8-9 and 10-11). Instead, WHIRO duplicates the value of `b` in memory.

We shadow variables whenever undefined values can reach an inspection point. A special **undef** token is stored in the shadow location. This token is unique for every occurrence of what, in LLVM jargon, is known as an "immediate undefined behavior" [Lopes et al., 2021, Sec.2]. In this way, when comparing two different versions

```

1 int main(int argc, char** argv){
2   int a = 10;
3   int b; alloca(shadow_b)
4   for(int i = 0; i < 5; i++){
5     if(argc > 1){
6       b = argc; *shadow_b = arc
7       while(b < a)
8         b++; *shadow_b += *shadow_b
9     }
10  }
11  SIP print(*shadow_b);
12  return 0;
}

```

Figure 4.8. In this program, variable `b` would be shadowed. The value of the duplicated variable will be printed at the inspection point.

of the same program, WHIRO will match two instance of the same **undef** token. We opted for this strategy for simplicity, as it makes it unnecessary to update the SSA-form representation of the program. A more mature version of WHIRO probably will not resort to shadowing. Instead, keeping variables in SSA-like virtual registers for as long as it is possible.

Scalarized Aggregate Types WHIRO needs to deal with program transformations when reporting the state of variables at inspection points. Many code optimizations implemented in LLVM preserve the metadata associated with program symbols. Therefore, they pose no problems to WHIRO. However, a few optimizations invalidate this metadata. One of these optimizations is *array scalarization*. Array scalarization replaces a cell within a array with a temporary register. Readings and updates meant to happen over the array are diverted to the register. Example 13 shows an example of this optimization in action.

Example 13. Figure 4.9 (Top) shows a program that performs updates on an array `r`. The update depends on the values of arrays `a` and `b`. It is possible to scalarize `r[i]`. Figure 4.9 (Bottom) shows the program after scalarization takes place.

Scalarization renders the state of an array invalid at some program regions. As an example, the contents of array `r` differ within the conditionals in Figure 4.9. If an inspection point exists in a region that contains scalarized arrays, then the contents of these arrays cannot be compared. To deal with this problem, WHIRO adds metadata

```

1 void sum0(int* a, int* b, int*restrict r, int N) {
2   int i;
3   for (i = 0; i < N; i++) {
4     r[i] = a[i];
5     if (!b[i]) {
6       r[i] = b[i];
7     }
8   }
9 }
10
11 void sum1(int* a, int* b, int*restrict r, int N) {
12   int i;
13   for (i = 0; i < N; i++) {
14     int tmp = a[i]; scalarized:tmp/r
15     if (!b[i]) {
16       tmp = b[i]; scalarized:tmp/r
17     }
18     SIP inconsistent_hash:r
19     r[i] = tmp; scalarized:tmp/r
20   }
21   SIP consistent_hash:r
22 }

```

Figure 4.9. (Top) Program before scalarization. (Bottom) Program after scalarization.

to the program, to indicate the places in the code that contain scalarized arrays. When traversing the data at a inspection point that exists at such a place, WHIRO still prints the hash of the scalarized array, albeit with a message to the user, indicating that the hash was taken from stale memory.

4.4 Properties of the Memory Monitor

A number of properties ensue from our implementation. We provide here sketches of proofs of each property. We have verified them experimentally, as we explain in chapter 6.

Property 1.(Non-interference) The memory monitor does not alter any value originally computed by the program.

Proof (Sketch): WHIRO does not update memory originally allocated by the program. Notice that WHIRO is not free of side-effects: if users inspect a program point, data shall be output from the program.

Property 2.(Time Complexity) When retrieving or updating information, stack and global variables are located in $O(1)$; heap-allocated blocks are retrieved in $O(\log_n)$, where n denotes the number of blocks stored in the heap.

Proof (Sketch): Stack and global variables are accessed via their addresses. WHIRO uses a hash table to record heap blocks. Typically, the average case to access the Heap Table will be $O(1)$, but collisions are handled via a balanced tree, and so the worst case $O(\log_n)$ complexity is possible.

Property 3.(Space Complexity) The sizes of the components G , S and T of the memory monitor are determined statically. G and S exists only at compilation time; T exists at compilation time and also at running time. The size of G is proportional to the number of global variables in the program; the size of S is proportional to the number of local variables declared in the program; the size of T is proportional to the number of types declared in the program.

Proof (Sketch): These facts are consequence of WHIRO's implementation: the sizes of G , S and T are determined statically, based on the number of global and automatic variables, and in the number of types declared in the program.

Property 4.(Ordering) When inspecting visible state, first stack-allocated names and heap-blocks are read, then global names. Global and stack data are reported in lexicographical order on the names of variables in source code. Heap blocks in the root of the reachable graph are reported in the order of updates. Within a connected component, the heap graph is reported in the order of updates.

Proof (Sketch): Global and stack variables are kept in G and S maps respectively. The contents of such maps are sorted by the key values, which in this case are the program symbols. Newly allocated heap blocks are pushed to the end of H , which means that it can be traversed as a list

Property 5.(Covering) At maximum precision granularity, every block that has been allocated in the heap and has not been freed is traversed at least once if this memory is reachable.

Proof (Sketch): Every allocated block is kept in the Heap Table. When inspecting a variable that points to heap-allocated data, WHIRO checks if such data points to another valid memory location. In positive case, WHIRO will also

report the data in said location and perform the reachability checking again. By doing that, WHIRO is able to traverse the entire valid heap. By keeping the *Free* flag in the entries of *H*, we are able to distinguish which addresses are valid in the program

Property 6.(Data Reporting) If data in the program is an alias to the first address of a memory block, then the block is traversed. Otherwise, the data is printed as an array with a single cell with the declared type of the data.

Proof (Sketch): When inspecting programs, WHIRO is able to gather information regarding the size of memory allocation, if such information is available in the program. This information can be either by directly accessing constants or values of variables or by creating instructions to compute such size at runtime. In this case, WHIRO will traverse the entire block of memory using the size information and report all the data within that block. In case there is no size information of memory allocation, WHIRO reports the data by just as a single cell array

4.5 Customizations

WHIRO can be customized along three dimensions: memory allocation, tracking graph, and SIP granularity. These dimensions trade precision for performance. In this context, "precision" is ranked by the amount of information stored in the memory monitor, and "performance" is ranked by the running-time overhead imposed by instrumentation.

Memory Allocation. This customization determines which memory region of the program will be tracked by inspection points. WHIRO recognizes any combination of three regions:

STATIC: Tracks memory allocated statically.

STACK: Tracks memory allocated on the stack.

HEAP: Tracks memory allocated in the heap.

Tracking Graph. When showing the visible program state, WHIRO can either treat program symbols as isolated entities, or can relate them via the pointers present in the instrumented code. These two possibilities give us the following customization modes:

FAST: Only local and static variables are tracked as isolated locations. Hence, this mode does not follow pointers when presenting the program state.

PRECISE: WHIRO shows the graph formed by relations between pointers. Contrary to the previous mode, this customization requires building the heap table.

WHIRO, in the PRECISE mode, works as a dynamic version of shape-analysis [Sagiv et al., 1998]—a fact that we state in Property 7.

Property 7.[Shape] Let π_s be a static inspection point, and let π_d be any related dynamic inspection point. WHIRO will produce for π_d an image of the heap with no more edges than a “may” version of shape-analysis would summarize for π_s . Similarly, its image should contain for any π_d no less edges than a “must” version of shape analysis summarizes.

Proof (Sketch): WHIRO draws edges between heap blocks only if such blocks are reachable. Although the heap table maintains freed addresses, edges that target such addresses are never reported. WHIRO essentially implements a dynamic analysis on programs: every fact that it reports is a fact that happened during some execution of a program. Therefore, if the fact is “Pointer p must refer to memory m ”, then WHIRO will not report false positives, provided that p has been declared with a pointer-qualified type. Similarly, if a correct implementation of a may-shape analysis says that “ p might never refer to m ”, then WHIRO will never report this edge in the heap graph.

```

1 int main(){
2   int* a = (int*)malloc(16);
3   int *b = a;
4   int *c = a + 1;
5   return 0;
6 }
```

Figure 4.10. A program with one heap allocation.

Example 14. Consider the program shown in Figure 4.10. When inspecting this program in PRECISE mode with heap tracking, WHIRO will print:

```
*a main 1: 2347902291 // Hash of a 16-byte array
*b main 1: 2347902291 // Hash of a 16-byte array
*c main 1: 0 // Value at a[1]
```

When inspecting this program in PRECISE mode without heap tracking, WHIRO will print:

```
*a main 1: 0 // Value at a[0]
*b main 1: 0 // Value at a[0]
*c main 1: 0 // Value at a[1]
```

In short, blocks of memory allocated in the heap are reported as a hash code of the contents of their bytes. Pointers to the first address of these blocks are reported as said hash code. Pointers to addresses past that point are reported as single-cell arrays with the type declared for the pointer.

SIP Granularity. WHIRO allows the customization of static inspection points. In principle, any region in the program where a new instruction could be inserted is an inspection point. However, for the sake of pragmatism, we currently support only two granularities of inspection points:

MAIN: the return point of the `main` routine is inspected.

ANY: The return point of any routine is inspected.

Chapter 5

Applications of Inspection Points

This chapter presents three applications of WHIRO, each using a different customization of the framework. We go over the case studies in sections 5.1–5.3.

5.1 Debugging Aid

The construction of tools to debug compiler optimizations has been a common source of research in programming languages [Pnueli et al., 1998; Necula, 2000]. The ability to compare states in an inspection point that is common across different versions of a transformed program lets us contribute along this direction.

Purpose: Given a program P plus a set of inputs, and an optimized version P' of it, compare the internal state of P and P' when executing with the given inputs. Use the result of this comparison to pinpoint bugs in the optimization.

Challenge: We are comparing two potentially very different versions of the same program. There is not a perfect match between inspection points: optimizations like inlining remove some inspection points in the ANY mode. There is not a perfect match between program symbols neither, as optimizations like constant propagation remove symbols.

Instrumentation mode: To maximize the likelihood that differences in program state are observed, we instrument every memory region (STATIC+ STACK+ HEAP), using the PRECISE mode, at maximum granularity (ANY).

Results: We deliver an approximate solution for the problem of bug finding: our debugger only matches program points that are equivalent in both versions of the program. We are able to highlight program symbols that have been erased in the optimized version of the program, to reduce false positives. Auxiliary variables created by the compiler are not considered in this comparison, for they are not associated with

high-level debugging information, nor they have exact correspondence between P and P' . In section 6.5 we show that this usage of WHIRO has been able to detect bugs injected in LLVM's implementation of constant propagation.

5.2 Adding Verification Outputs to Benchmarks

The synthesis of benchmarks is an important program in the construction of predictive compilers [Cummins et al., 2017, 2018]. A common technique to synthesize benchmarks is to mine code from open-source repositories [da Silva et al., 2021]. However, for verification purposes, a benchmark must have some output. By comparing said output with a reference, compiler engineers have confidence on the validity of any test produced with that benchmark.

Purpose: Given a program P , add output to it, by printing the state of static and local variables after execution.

Challenge: If instrumentation is too intrusive, the results obtained through benchmarking P might be subject to “probe effects”. These effects emerge when inspecting the program provokes unintended alterations of its behavior. Therefore, the performance of P must be preserved as much as possible.

Instrumentation mode: To reduce probe effects, we configure WHIRO to instrument MAIN. It reads STATIC and STACK memories in the FAST mode.

Results: We have applied the above customization of WHIRO onto 137 executable programs downloaded from ANGHABENCH. Each benchmark was tested with 10 different inputs. Outputs were produced for all the 137 programs, in 1,359 executions (out of a total of 1,370). Property 4 was verified when repeating this experiment. Each benchmark consists of a driver plus a function (which is the benchmark proper). Not counting the driver, which always gives us the same outputs, on average WHIRO inspects 7.8 variables per function, with a minimum of 2 variables and a maximum of 32 variables. The instrumented benchmarks have been returned to the maintainers of ANGHABENCH, and are now publicly available.

5.3 Data Visualization

Tools able to provide graphic representation of the heap are useful for program understanding and debugging [Aftandilian et al., 2010]. Viewing data structures and other program elements in a graphical format makes it easier to analyze the state of memory, recognize patterns, and observe the relation between different allocated blocks.

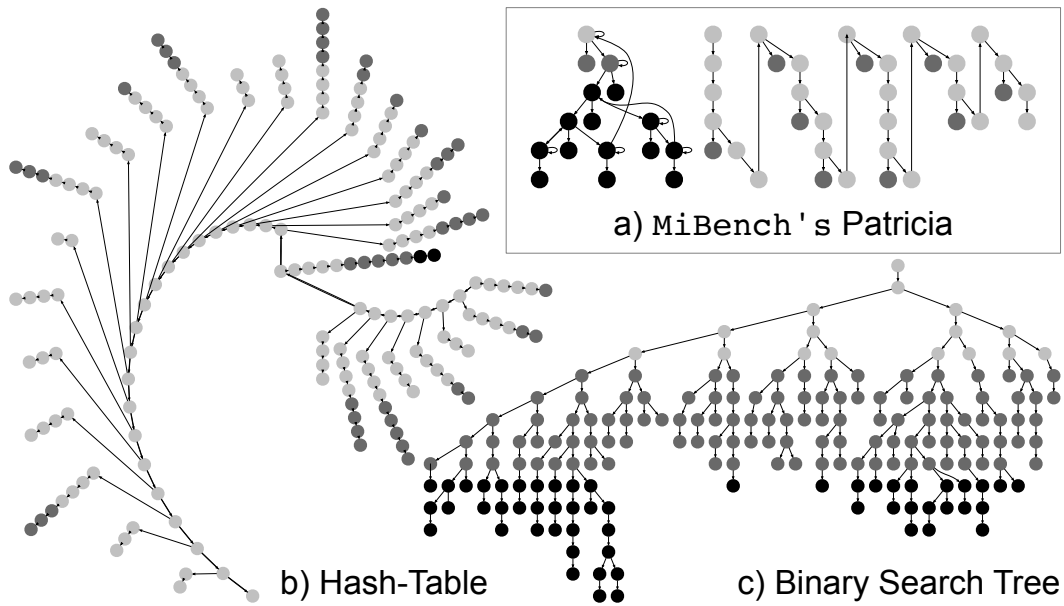


Figure 5.1. (a) Snapshot showing the two disjoint data structures in MiBENCH’s Patricia, after the first invocation of `pat_search` returns with the test input. (b-c) The two disjoint graphs in the heap of a program that copies a binary tree into a hash table. Collisions are stored in a linked list.

Purpose: Adapted WHIRO to render a visual representation the graph determined by relations between pointers in the heap in a program.

Challenge: To the best of our knowledge, all the techniques discussed in the literature to visualize program state deal with type-safe languages with managed memory. Java is the usual target [Aftandilian et al., 2010; Grech et al., 2017]. Tracking relations between pointers in C or C++ is non-trivial, due to the difficulty to distinguish memory addresses from scalar types.

Instrumentation mode: We customize WHIRO with the following configurations: the tracking graph is PRECISE; the SIP granularity is ANY; and considering the HEAP memory allocations.

Results: Figure 5.1 shows heap snapshots produced with our adaptation of WHIRO. To ease visualization, we are showing only nodes stored in the heap. Currently, we can visualize the heap of all the programs in the MiBENCH collection, for instance. Graphs are produced in DOT format. Users can render them using different graph visualization algorithms. We are not able to distinguish pointer relations created by non-pointer types, like it happens in the infamous doubly-linked XOR list. As an example, when given the program in Figure 1 of Banerjee et al., WHIRO prints a series of unconnected blocks with the `xorlist` type.

Chapter 6

Experimental Results

In this chapter we evaluate the ideas presented in this paper. We implemented our instrumentation on LLVM version 10.0.0. Data structures that store the auxiliary state are implemented in C. They are linked statically with the bytecode that LLVM produces. All the experiments were performed in an 8-core Intel i7-8565U processor with a clock of 1.80 GHz, and 8GB of DDR4 (RAM) operating at 2,400MHz and running Linux Ubuntu 64-bit version 18.04.5 LTS. Running time and memory usage are collected via Linux' built-in `time` command. Number of LLVM instructions are collected using LLVM's `-instcount`. The other statistics reported in Section 6.4 are gathered directly by the instrumentation pass.

We chose MIBENCH [Guthaus et al., 2001] to evaluate our techniques. We use the version of MIBENCH available in the LLVM test repository, which contains 16 benchmarks. We have constructed inputs for these programs using a synthesizer available at <https://github.com/ekut-es/mibench>, to ensure that each benchmark runs for more than 1.0 second when compiled with `clang -O0 (v10.0)`. We failed to meet this criterion for MIBENCH's *Patricia*, which runs for about half-a-second with the largest input that we found.

To guide our evaluation process, we investigate the following research questions:

Compilation Overhead: What is the overhead that our instrumentation adds to the compilation time?

Running Time Overhead: What is the runtime overhead imposed by our transformation on the instrumented programs?

Memory Overhead: What is the memory consumption imposed by our technique?

Code-Size Overhead: How does the size of the instrumented program grows in relation to its original size?

Effectiveness: Can our technique be used to detect actual bugs in compiler optimizations?

Experiments in sections 6.1, 6.2, and 6.3 report averages of 12 executions, while discarding the 2 slowest ones. The programs were compiled with `mem2reg`, to promote memory references to virtual registers, and `mergereturn`, to ensure that every function will have a single exit point. Results are considered statistically significant within a confidence level of 99% via a Student T-Test.

6.1 Compilation Overhead

One of the design principles of WHIRO is to move to compilation time as much instrumentation overhead as possible. On the one hand, this strategy reduces the overhead that our inspection points impose onto executable programs, as we shall demonstrate in Section 6.2. On the other, it extends compilation time. This section analyzes this impact.

Discussion: Figure 6.1 shows the time required to instrument the different programs in MIBENCH. For reference purposes, we also show the time taken to compile each program with `clang -O0 -g`. Instrumentation time is shorter than standard compilation time in every case. The FAST and PRECISE inspection modes lead to longer instrumentation time in almost all the benchmarks, since all the variables in every function must be inspected. MAIN tends to be always the fastest instrumentation mode, because it inspects only one function. STATIC is also fast, because MIBENCH programs usually have less static variables than local and heap variables. The slowest absolute instrumentation time was 7.26 seconds using the PRECISE mode in MIBENCH’s `typeset`. Yet, just to compile `typeset` without any optimization already takes 33.55 seconds. Table 6.4 provides complete ratios.

6.2 Running Time Overhead

WHIRO imposes an overhead on executable programs due to: (i) keeping the auxiliary state; and (ii) shadowing stack-allocated variables. This section analyzes this impact.

Discussion: Figure 6.2 shows how the performance of executable programs varies depending on the instrumentation mode. Points in Figure 6.2 show the ratio between

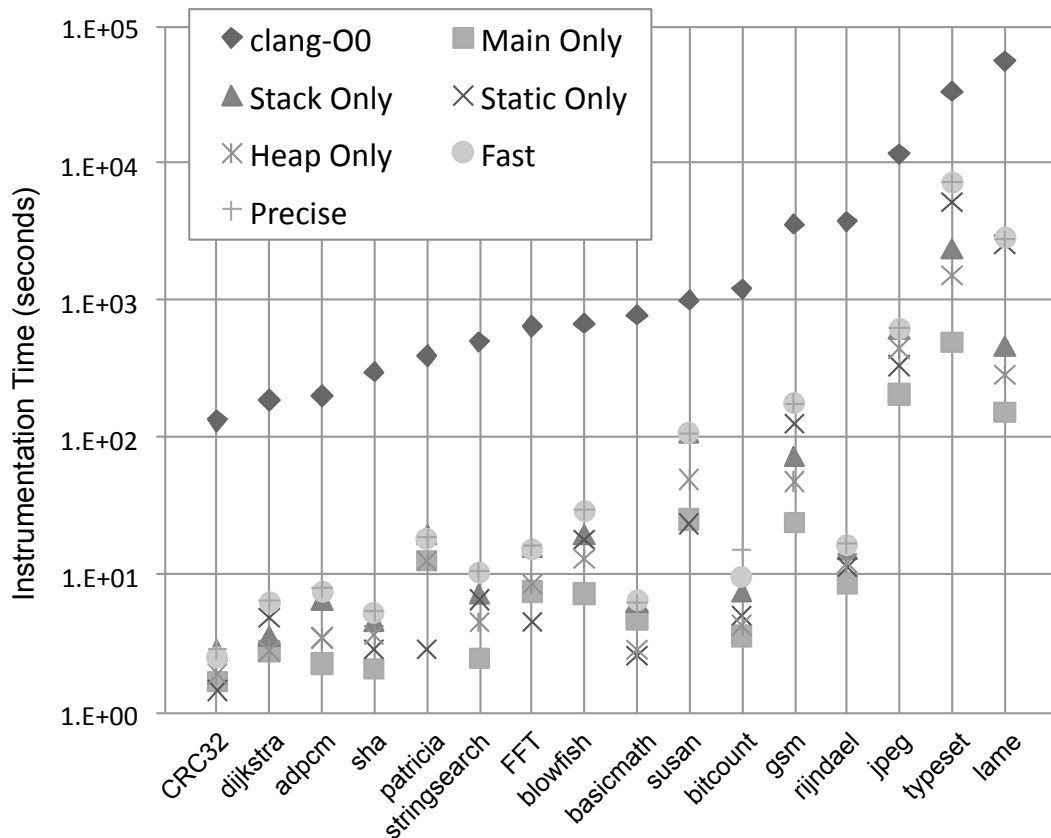


Figure 6.1. Time to instrument programs (in seconds). For reference, we provide the time of compiling each program with `clang -O0 -g`.

the execution time of the instrumented program and its original version. Overhead increases with the granularity of the inspection; hence, PRECISE yields the slowest executables. This outcome is to be expected, because PRECISE inspects all the variables in a program and updates the heap table during execution. STATIC is also slow in general, because this mode inspects static variables at every function call (unless MAIN only is used). MIBENCH’s `dijkstra` accounted for the largest overheads. The slowdown, in this case, was caused by excessive heap usage: the program manipulates a graph stored dynamically. There were cases in which the instrumented program was faster than its original version. However, for all these cases we found p-values greater than 0.01; hence, we cannot consider them statistically significant.

Printing Data Overhead. Depending on the instrumentation mode, we have observed runtime overheads of almost 40x in `dijkstra`. However, WHIRO manipulates a large quantity of data; hence, this overhead is to be expected. To give the reader some perspective on this observation, we analyze the overhead that WHIRO incurs if this data is to be printed. Table 6.1 compares the running time of three benchmarks:

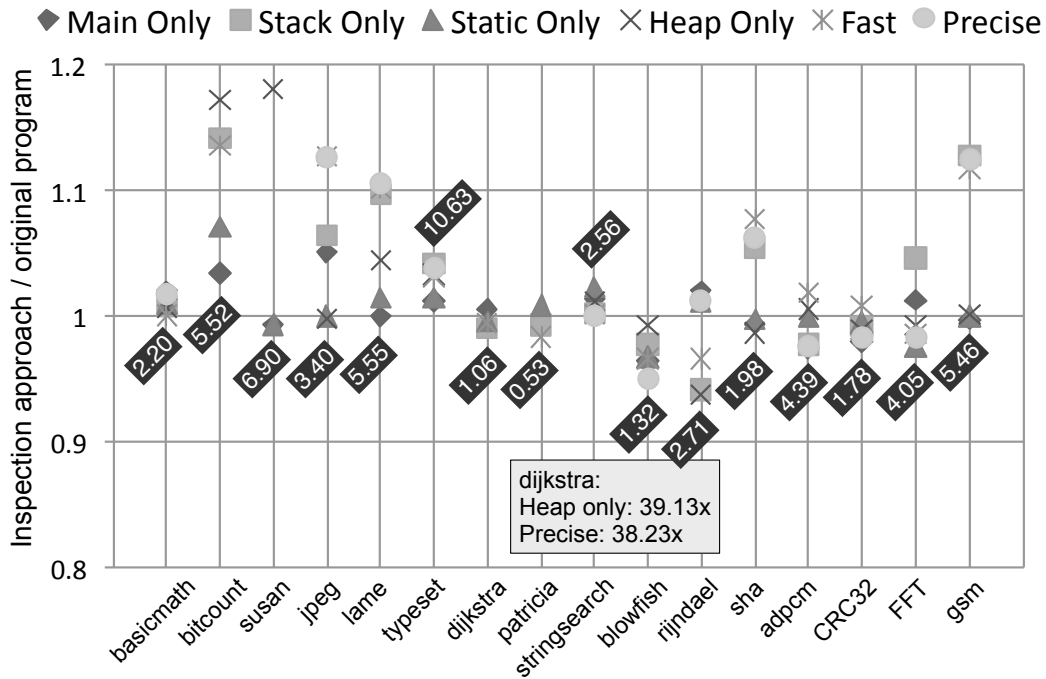


Figure 6.2. Increase of execution time (with relation to the original program). Numbers in boxes denote the running time of the original program.

`stringsearch`, `sha`, and `FFT` with and without counting the time necessary to print the inspection traces. Printing obviously increases the running time in all inspection modes. This increase is more noticeable in the `FAST` and `PRECISE` modes, which inspect values stored in all memory regions. Time grows, in the worst case, by factors of 2.7x, 35.7x and 24.8x in `stringsearch`, `sha`, and `FFT`, respectively.

Table 6.1. Time to run the benchmarks (in seconds) with and without counting the time to print inspection traces.

	stringsearch		sha		FFT	
Inspection Mode	No Print	Print	No Print	Print	No Print	Print
MAIN	2.59	2.85	1.97	2.89	4.09	4.24
STACK	2.56	4.35	2.09	57.44	4.23	3.62
STATIC	2.62	6.75	1.97	2.31	3.95	92.03
HEAP	2.59	2.67	1.95	1.94	4.02	4.32
FAST	2.56	2.67	2.13	57.80	3.99	85.90
PRECISE	2.56	6.88	2.11	75.26	3.98	93.83

6.3 Memory Overhead

The dynamic components of the Memory Monitor exist during all the execution of a program. Therefore, inspection points are expected to increase memory consumption. To read peak memory usage, we use Linux' `time -v` and report memory consumption as the "maximum resident set size".

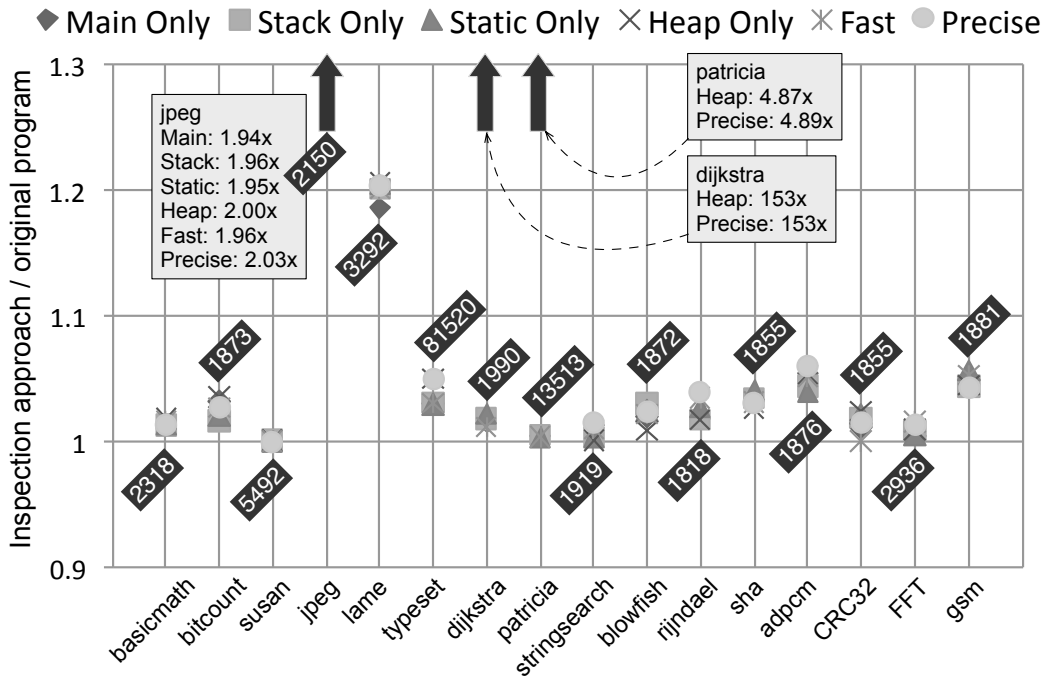


Figure 6.3. Increase in memory consumption (with relation to the original program). Numbers in boxes denote memory consumed by the original program, in Kilobytes.

Discussion: Figure 6.3 plots the memory consumption ratio between instrumented and non-instrumented programs. In 81% of the benchmarks, memory consumption increased by no more than 1.3x, and in 75% of them, this overhead was less than 1.1x. The inspection modes which use more memory are PRECISE and HEAP. In addition to the type table T , these instrumentation modes keep the heap table H in memory. Large memory usage was observed in `dijkstra` when using either of these instrumentation modes. The largest increase of memory consumption due to non-heap memory was observed in `jpeg`. With 2,560 entries, this program gave us the largest type table among all the benchmarks.

6.4 Code-Size Overhead

Instrumentation increases code. This boost is due to the new routines that access the T and H tables, due to the variables created to shadow stack-allocated data, and due to code to print the values of variables at inspection points. In this section, we analyze this growth. The size of code is measured in number of LLVM instructions.

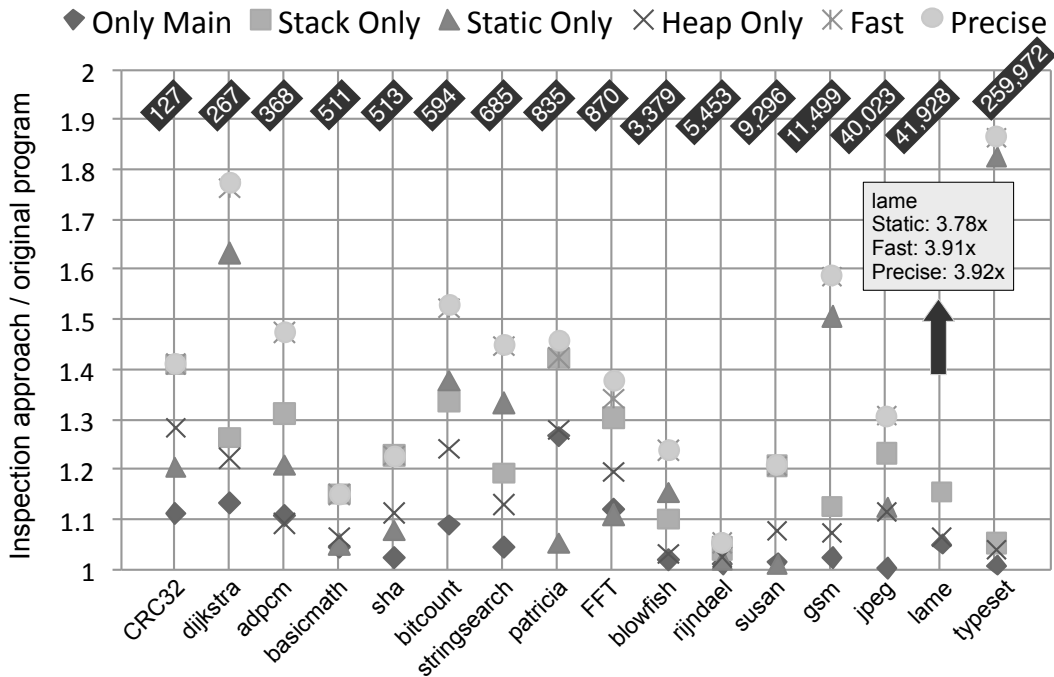


Figure 6.4. Code growth (with relation to the original program). Numbers in boxes denote the number of LLVM instructions in the original program, compiled with `mem2reg`. Benchmarks are sorted by the size of the original program.

Discussion: Figure 6.4 shows the ratio between the sizes of instrumented and original programs in MIBENCH. The FAST and PRECISE instrumentation modes are, as expected, the most prodigal, increasing code size by factors of almost 4x in `lame`. Growth in the other benchmarks is more moderate, but typically above 2x for these modes. STATIC also tends to increase code-size by substantial margins, because static variables are inspected in every function call. The MAIN mode accounts for the smallest code increase, because an inspection point is created in only one function.

Factors of Code Growth. We conducted an experiment aiming to investigate whether there exist features of programs that correlate well with code growth. Table 6.2 summarizes these results. MIBENCH contains 1,228 functions spread across 16 benchmarks. Most variables, 7,130, are stack allocated; 3,582 variables are non-static

pointers and 788 are static (of any type). Table 6.2 shows that the number of functions and variables is strongly correlated with code growth when inspecting only the stack or the heap. In all these cases, Pearson R^2 is above 0.95. When inspecting only static memory, the number of variables and the number of original instructions are the determining factors. In this case, R^2 is always above 0.85. When inspecting the entire program—with either the FAST or the PRECISE modes—instructions are the factor that determines growth, with an R^2 value of 0.86.

Table 6.2. Relations between program features and code growth. We let `fst` = FAST and `prc` = PRECISE.

Inspection Mode	Variables			Functions			Instructions		
	Total	Mean	R^2	Total	Mean	R^2	Total	Mean	R^2
MAIN	891	52.4	0.80	1	1	X			0.29
STACK	7130	419.4	0.97			0.97	388,003	22,823.7	0.73
STATIC	788	46.4	0.87	1,228	76.75	0.57			0.85
HEAP	3582	210.7	0.95			0.97			0.91
<code>fst/prc</code>	7918	465.8	0.63			0.60			0.86

6.5 Effectiveness

As mentioned in Section 5.1, WHIRO can be used to debug program transformation techniques like compiler optimizations. A state mismatch between two versions of a program processing the same input is a strong indication of a bug. Since inspection points are inserted after the program has been optimized, they do not prevent optimizations from happening altogether and can therefore be used as an alternative to detect bugs. Nevertheless, it is important to emphasize that our instrumentation is not capable of building the same execution trace in face of all compiler optimizations. Some transformation that change the shape of the program may invalid inspection points, and WHIRO would miss some data in those cases.

To analyze the effectiveness of WHIRO as a debugging aid, we manually inserted a bug into the LLVM’s sparse conditional constant propagation pass. We altered the optimization lattice by changing the semantics of its meet operator. Whenever we have two constants $c_1 \wedge c_2$, with $c_1 = c_2$, we propagate an undefined value instead of the constant. Programs were instrumented using PRECISE mode.

Discussion: The unsound implementation of constant propagation caused bugs in five, out of 16 MIBENCH programs. Opportunities for injecting bugs did not occur in the other benchmarks. Figure 6.3 summarizes results. The first row reports the

Table 6.3. Bugs reported by WHIRO.

	basicmath	susan	jpeg	FFT	gsm
Differences in IR	8	4	1	2	1
Bugs injected	21	1	1	2	1
WHIRO warnings	2	0	10	2	0

number of different instructions found in a `diff` between programs compiled with the correct and the buggy implementation of constant propagation. The second row reports how often a wrong value was computed in the incorrect implementation of constant propagation. The last row shows the number of program variables identified by WHIRO with incorrect values due to the bug. WHIRO has correctly pinpointed 14 of the 26 bugs. The correspondence is not perfect because: (i) WHIRO only prints out results for variables that have a name in the program—auxiliary locations created by the compiler are not tracked; and (ii) WHIRO only compares state at the return point of functions. Notice that this choice of inspection point is configurable. Furthermore, in `jpeg`, one wrong value propagation led to 10 incorrect values found at execution time.

6.6 Summary of Results

Table 6.4 summarizes the key results discussed in this section. For reference, the 16 programs in MIBENCH, together, gives us 376,338 LLVM instructions when compiled with `clang -O0 -g` with the `mem2reg` pass. It takes 115 seconds to compile these programs (with the above flags) in our setup.

Table 6.4. Summary of results. **Cmp**: compilation time (Sec. 6.1). This column is the geometric mean of 16 ratios comparing the instrumentation time with the compilation time (`clang -O0 -g`). **Exe**: runtime overhead (Sec. 6.2); **Mem**: Memory consumption (Sec. 6.3); **CG**: code growth (Sec. 6.4). These three columns report geometric means of 16 ratios between instrumented and original program. **Size**: number of LLVM instructions in the 16 benchmarks.

	Cmp	Exe	Mem	CG	Size
MAIN	0.01	1	1.07	1.06	381,129
STACK	0.02	1.04	1.07	1.21	409,706
STATIC	0.01	1	1.07	1.32	720,339
HEAP	0.01	1.31	1.63	1.12	395,882
FAST	0.03	1.04	1.07	1.48	747,094
PRECISE	0.03	1.34	1.63	1.48	747,439

Chapter 7

Conclusion

In this dissertation, we presented a technique to inspect the internal state of programs. We are able to observe values stored in the heap, in the stack of functions, and in static memory. In terms of implementation, this technique borrows most of its ideas from previous research concerning the design and implementation of tracing-based garbage collectors for C and C++. However, we have redirected these ideas: instead of doing memory management, we give users a human-observable “peephole” into the program state. This peephole can be used in various ways, as Chapter 5 demonstrates; furthermore, it is adjustable, going from slow/precise to fast/cursory modes. For pragmatic reasons, we choose to inspect the return point of functions, but our solution can report the program state at any program point where code can be injected. In addition to it, we can handle programs optimized in several ways.

Dealing with a type-unsafe language has indeed proved to be a challenge. For example, not being able to know whether an address is valid within the program’s memory segment made us take decisions that limit our extent, but that make our implementation safe. We also had to overcome the fact that some information is not available at compilation time, such as the final values stored in union variables. Nevertheless, our technique is effective and efficient for different purposes and, as Chapter 6 shows, we imposed minimum overhead to the execution of a program.

7.1 Limitations & Future Work

This work, which exists today as the WHIRO framework, has limitations. In particular, WHIRO, like most on-the-fly garbage collectors [Zakowski et al., 2019], does not synchronize access to the heap table H by default. Thus, it cannot guarantee Property 4 for multi-threaded programs. As many of our decisions, this one is also pragmatic:

between performance and ordering, we opted for the former. Reversing this decision is a small change in the framework's implementation.

Another limitation of our solution concerns types. As mentioned in section 5.3, we are not able to correctly inspect the doubly-linked XOR list because the pointer relations are created by non-pointer types. Without type information, it becomes very challenging to recover the intention of a program and WHIRO cannot retrieve tricky handling of program elements.

```
1 int main(){
2   int* p1 = (int*) malloc (sizeof(int));
3   char a[80];
4   scanf("%s", a);
5   int* p2 = (int*)atoi(a);
6   int i = (int)p2;
7   return 0;
8 }
```

Figure 7.1. A program with type casts between a pointer and integer.

Example 15. In the program in 7.1, the programmer might want to use $p2$ as an integer, and i as a pointer. It might be possible to design static analysis that recover part of this original intent, e.g., if an integer eventually is cast into a pointer, then it was meant to be used as a pointer in the first place. This said, we have not worked on this line of research.

Concerning the choice of tools to implement our ideas, LLVM proved to be a good choice for some aspects and not a good fit for others. We used it to instrument programs, by means of a pass that works in the intermediate representation produced by compiling the program with the clang compiler. LLVM provides a great interface to create instructions and inject them into programs, which facilitated the creation of shadow variables, code to update the heap table, and inspection points. Furthermore, the debug information emitted by the compiler is accurate enough to support the building of traces of source code variables.

On the other hand, LLVM makes some of the aspects of our approach difficult. The LLVM intermediate representation has its own type system, which is different from the type system of high-level programming languages. Although we instrument the LLVM bytecodes of programs, we rely on the source type system to inspect variables because it allows us to report information that may be not available in the intermediate representation, such as names of fields in structure types. Relating the SSA definitions with source code types is not a straightforward task. Another drawback ensued from

working at the intermediate representation level is handling the variables which are dead at inspection points. As we explained in Chapter 4, we deal with this by extending live ranges and shadowing variables in the stack of functions. This would not be necessary in case the instrumentation is inserted before other compiler optimizations run. We believe that moving our instrumentation to earlier stages in the compilation pipeline would solve this problem and the generated code would probably be more efficient. We leave this as possible future direction for this research.

Bibliography

- Aftandilian, E. E., Kelley, S., Gramazio, C., Ricci, N., Su, S. L., and Guyer, S. Z. (2010). Heapviz: Interactive heap visualization for program understanding and debugging. In *SOFTVIS*, page 53–62, New York, NY, USA. Association for Computing Machinery.
- Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA. ISBN 0321486811.
- Alipour, M. A., Groce, A., Gopinath, R., and Christi, A. (2016). Generating focused random tests using directed swarm testing. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 70–81.
- Amodio, M., Chaudhuri, S., and Reps, T. (2017). Neural attribute machines for program generation. *arXiv preprint arXiv:1705.09231*.
- Banerjee, S., Devescery, D., Chen, P. M., and Narayanasamy, S. (2020). Sound garbage collection for c using pointer provenance. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28.
- Barany, G. (2017). Liveness-driven random program generation. In *LOPSTR*, pages 112–127, Heidelberg, Germany. Springer.
- Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. (2006). The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, NY, USA. ACM.
- Boehm, H.-J. (1993). Space efficient conservative garbage collection. *ACM SIGPLAN Notices*, 28(6):197–206.

- Boehm, H.-J. and Weiser, M. (1988). Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807--820.
- Brusilovsky, P. (1993). Program visualization as a debugging tool for novices. In *INTERACT*, page 29--30, New York, NY, USA. Association for Computing Machinery.
- Chen, T., Sura, Z., and Sung, H. (2016). Automatic copying of pointer-based data structures. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 265--281. Springer.
- Cummins, C., Petoumenos, P., Murray, A., and Leather, H. (2018). Compiler fuzzing through deep learning. In *ISSTA*, pages 95--105, New York, NY, USA. ACM.
- Cummins, C., Petoumenos, P., Wang, Z., and Leather, H. (2017). Synthesizing benchmarks for predictive modeling. In *CGO*, pages 86--99, Piscataway, NJ, USA. IEEE.
- Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1989). An efficient method of computing static single assignment form. In *POPL*, pages 25--35, New York, NY, USA. ACM.
- da Silva, A. F., Kind, B. C., de Souza Magalhães, J. W., Rocha, J. N., Guimarães, B. C. F., and Pereira, F. M. Q. (2021). AnghaBench: A suite with one million compilable C benchmarks for code-size reduction. In *CGO*, pages 378--390, USA. IEEE Computer Society.
- Dor, N., Rodeh, M., and Sagiv, M. (2000). Checking cleanness in linked lists. In *International Static Analysis Symposium*, pages 115--134. Springer.
- Eide, E. and Regehr, J. (2008). Volatiles are miscompiled, and what to do about it. In *Proceedings of the 8th ACM international conference on Embedded software*, pages 255--264.
- Grech, N., Fourtounis, G., Francalanza, A., and Smaragdakis, Y. (2017). Heaps don't lie: Countering unsoundness with heap snapshots. *Proc. ACM Program. Lang.*, 1(OOPSLA).
- Guthaus, M. R., Ringenber, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. (2001). MiBench: A free, commercially representative embedded benchmark suite. In *WWC*, pages 3--14, Washington, DC, USA. IEEE.
- Hathhorn, C., Ellison, C., and Roşu, G. (2015). Defining the undefinedness of c. In *PLDI*, page 336--345, New York, NY, USA. Association for Computing Machinery.

- Henderson, F. (2003). Accurate garbage collection in an uncooperative environment. *ACM SIGPLAN notices*, 38(2):256--262.
- Henning, J. L. (2006). SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1--17. ISSN 0163-5964.
- Kumar, R., Myreen, M. O., Norrish, M., and Owens, S. (2014). Cakeml: a verified implementation of ml. *ACM SIGPLAN Notices*, 49(1):179--191.
- Kundu, S., Lerner, S., and Gupta, R. K. (2010). Translation validation of high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(4):566--579.
- Le, V., Afshari, M., and Su, Z. (2014). Compiler validation via equivalence modulo inputs. In *PLDI*, page 216--226, New York, NY, USA. Association for Computing Machinery.
- Lee, D., Won, Y., Park, Y., and Lee, S. (2020). Two-tier garbage collection for persistent object. In *SAC*, page 1246--1255, New York, NY, USA. Association for Computing Machinery.
- Leroy, X. (2009). Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107--115. ISSN 0001-0782.
- Leung, A., Bounov, D., and Lerner, S. (2015). C-to-verilog translation validation. In *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 42--47.
- Li, T., Guo, Y., Liu, W., and Tang, M. (2013). Translation validation of scheduling in high level synthesis. *GLSVLSI '13*, page 101--106, New York, NY, USA. Association for Computing Machinery.
- Lindig, C. (2005). Random testing of c calling conventions. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 3--12.
- Lopes, N. P., Lee, J., Hur, C.-K., Liu, Z., and Regehr, J. (2021). *Alive2: Bounded Translation Validation for LLVM*, page 65--79. Association for Computing Machinery, New York, NY, USA.
- Lopes, N. P., Menendez, D., Nagarakatte, S., and Regehr, J. (2015). Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 22--32.

- Mastrangelo, L., Ponzanelli, L., Mocci, A., Lanza, M., Hauswirth, M., and Nystrom, N. (2015). Use at your own risk: The java unsafe api in the wild. In *OOPSLA*, page 695–710, New York, NY, USA. Association for Computing Machinery.
- Menendez, D., Nagarakatte, S., and Gupta, A. (2016). Alive-fp: Automated verification of floating point based peephole optimizations in llvm. In *International Static Analysis Symposium*, pages 317–337. Springer.
- Morisset, R., Pawan, P., and Zappa Nardelli, F. (2013). Compiler testing via a theory of sound optimisations in the c11/c++ 11 memory model. *ACM SIGPLAN Notices*, 48(6):187–196.
- Mullen, E., Zuniga, D., Tatlock, Z., and Grossman, D. (2016). Verified peephole optimizations for compcert. In *PLDI*, page 448–461, New York, NY, USA. Association for Computing Machinery.
- Nagai, E., Awazu, H., Ishiura, N., and Takeda, N. (2012). Random testing of c compilers targeting arithmetic optimization. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*, pages 48–53.
- Nagai, E., Hashimoto, A., and Ishiura, N. (2014). Reinforcing random testing of arithmetic optimization of c compilers by scaling up size and number of expressions. *IPSJ Transactions on System LSI Design Methodology*, 7:91–100.
- Necula, G. C. (2000). Translation validation for an optimizing compiler. In *PLDI*, page 83–94, New York, NY, USA. Association for Computing Machinery.
- Nethercote, N. and Seward, J. (2007). Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, page 89–100, New York, NY, USA. Association for Computing Machinery.
- Pnueli, A., Siegel, M., and Singerman, E. (1998). Translation validation. In *TACAS*, page 151–166, Berlin, Heidelberg. Springer-Verlag.
- Rafkind, J., Wick, A., Regehr, J., and Flatt, M. (2009). Precise garbage collection for c. In *Proceedings of the 2009 international symposium on Memory management*, pages 39–48.
- Richards, G., Gal, A., Eich, B., and Vitek, J. (2011). Automated construction of javascript benchmarks. *SIGPLAN Not.*, 46(10):677–694.

- Rinetzky, N. and Sagiv, M. (2001). Interprocedural shape analysis for recursive programs. In *International Conference on Compiler Construction*, pages 133–149. Springer.
- Ryabtsev, M. and Strichman, O. (2009). Translation validation: From simulink to c. In *International Conference on Computer Aided Verification*, pages 696–701. Springer.
- Sagiv, M., Reps, T., and Wilhelm, R. (1998). Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1):1–50. ISSN 0164-0925.
- Schildt, H. (1990). *The Annotated ANSI C Standard American National Standard for Programming Language—C: ANSI/ISO 9899-1990*. McGraw-Hill, Inc., USA. ISBN 0078819520.
- Sewell, T. A. L., Myreen, M. O., and Klein, G. (2013). Translation validation for a verified os kernel. In *PLDI*, pages 471–482, New York, NY, USA. ACM.
- Sidiroglou-Douskos, S., Lahtinen, E., Long, F., and Rinard, M. (2015). Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 43–54.
- Stallman, R. M., Pesch, R., and Shebs, S. (2002). *Debugging with GDB: The GNU Source-Level Debugger*. GNU Press, USA. ISBN 1882114884.
- Sun, C., Le, V., and Su, Z. (2016). Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 849–863.
- Tristan, J.-B., Govereau, P., and Morrisett, G. (2011a). Evaluating value-graph translation validation for llvm. In *PLDI*, pages 295–305, New York, NY, USA. ACM.
- Tristan, J.-B., Govereau, P., and Morrisett, G. (2011b). Evaluating value-graph translation validation for llvm. In *PLDI*, pages 295–305, New York, NY, USA. ACM.
- Tun Li, Yang Guo, Wanwei Liu, and Chiyuan Ma (2013). Efficient translation validation of high-level synthesis. In *International Symposium on Quality Electronic Design (ISQED)*, pages 516–523.
- Wilhelm, R., Sagiv, M., and Reps, T. (2000). Shape analysis. In *International Conference on Compiler Construction*, pages 1–17. Springer.

- Wilson, P. R. (1992). Uniprocessor garbage collection techniques. In *IWMM*, page 1–42, Berlin, Heidelberg. Springer-Verlag.
- Yang, L., Gurumain, S., Fahmy, S. A., Chen, D., and Rupnow, K. (2015). Jit trace-based verification for high-level synthesis. In *2015 International Conference on Field Programmable Technology (FPT)*, pages 228–231.
- Yang, X., Chen, Y., Eide, E., and Regehr, J. (2011). Finding and understanding bugs in c compilers. In *PLDI*, pages 283–294, New York, NY, USA. ACM.
- Zakowski, Y., Cachera, D., Demange, D., Petri, G., Pichardie, D., Jagannathan, S., and Vitek, J. (2019). Verifying a concurrent garbage collector with a rely-guarantee methodology. *J. Autom. Reason.*, 63(2):489–515. ISSN 0168-7433.
- Zhang, K., Lin, C., Chen, S., Hwang, Y., Huang, H., and Hsu, F. (2011). Transsql: A translation and validation-based solution for sql-injection attacks. In *2011 First International Conference on Robot, Vision and Signal Processing*, pages 248–251.
- Zhang, Q., Sun, C., and Su, Z. (2017). Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 347–361.
- Zorn, B. (1990). Comparing mark-and sweep and stop-and-copy garbage collection. In *LFP*, page 87–98, New York, NY, USA. Association for Computing Machinery.

Appendix A

Whiro Documentation

In this chapter we present an overview of WHIRO framework and a brief documentation of the main methods implemented.

A.1 Overview

Figure A.1 depicts a summary of our implementation. At compilation time, the instrumentation pass builds the static components of the memory monitor. First, it gathers information concerning the types present in the original program and builds the type table. Then, it analyzes the entire intermediate representation to create the global map G and the stack maps S for each function in the program. Using this information, the monitor inserts all the required instrumentation in the program to track and report its internal state. The instrumented program is statically linked against the bytecodes containing the dynamic components of the memory monitor. We call **Composite Inspector** the set of functions that are used to report non-scalar variables, such as pointers, unions, or arrays. At execution time, the instrumented version of the program reads the type table T and executes normally while updating the heap table H and using the composite inspector as an auxiliary library at the inspection points.

A.2 Methods

We will here focus on the methods that execute together with the program at runtime. These methods are used to report values of variables and to update the heap table. They are all implemented in C.

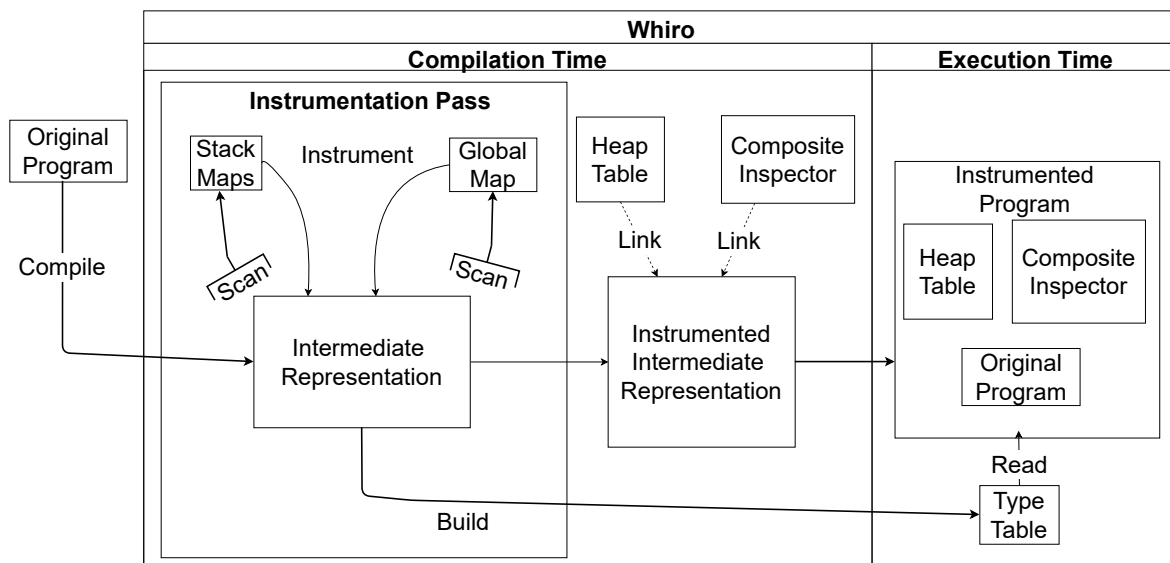


Figure A.1. WHIRO framework overview

openTypeTable

Signature: openTypeTable (const char* ProgramName, int TableSize, int InsHeap, int InsStack, int TrackPtr)

This function is in charge of reading the type table binary file built by the instrumentation pass. It allocates memory for each one of the type descriptors. This function is called at the beginning of the execution of the program and since it is called only once, we also use it to set some flags concerning filtering. More specifically, through this function we set the tracking graph granularity and we filter memory for pointer variables that may point either to stack locations or heap data. Static pointers are handled during instrumentation.

Parameters:

- *ProgramName* is the name of the program. It is used to open the type table file
- *TableSize* is the size of the type table
- *InsHeap* is true if the heap is to be inspected
- *InsStack* is true if the values store in the stack are to be inspected
- *TrackPtr* is true for PRECISE and false for FAST

insertHeapEntry

Signature: void insertHeapEntry(void* Block, int Size, int ArrayStep, int TypeIndex)

This function is responsible to insert a new entry in the heap table H . This entry corresponds to the heap block addressed by $Block$. It first checks whether there exists an entry holding that address. In a positive case, it just updates its size and type index. Otherwise, the function allocates a new entry and sets it up

Parameters:

- $Block$ is the heap address
- Size is the number of elements allocated
- ArrayStep is the increment the pointer to $Block$, so WHIRO can visit all data allocated in that block
- $TypeIndex$ is the type to access the type descriptor of that data

deleteHeapEntry

Signature: void deleteHeapEntry(void* Block)

This function sets the heap entry addressed by $Block$ to unreachable, if such entry exists in the table.

Parameters:

- $Block$ is the heap address

inspectHeapData

Signature: void printHeapData(HEAPENTRY* Entry, char* PtrName, char* FuncName, int CallCounter)

This function takes an entry from the heap table and report the data contained in it at some inspection point. It prints it together with the calling context, formed by the pointer name in the source code, the name of the function in which the inspection point is defined, and the current value of the call counter of that function. If that heap data contains a reference to another heap address, this functions is called recursively. Before it returns, it sets $Entry$ as visited.

Parameters:

- *Entry* is the heap table entry to be inspected
- *PtrName* is the name of the pointer that points to that heap address in the source code
- *FuncName* is the name of the function from the source code that it is being inspected
- *CallCounter* is the current value of *FuncName*

updateHeapEntrySize

Signature: void updateHeapEntrySize(void* Block, int NewSize)

This function updates the size of a *H* entry when there is a heap reallocation in the original program.

Parameters:

- *Block* is the heap address
- *NewSize* is the new size of the entry

setAllHeapUnvisited

Signature: void setAllHeapUnvisited()

This function sets the entire heap table as unvisited. WHIRO uses it to report aliases

inspectData

Signature: void inspectData(void* Data, TYPE* DataType, char* Name, char* FuncName, int CallCounter)

This function prints any type of data manipulated by the program. It receives a type descriptor and print every field within data data. It is usually used to print non-scalar variables, except for unions.

Parameters:

- *Data* is the value to be inspected
- *DataType* is the type descriptor of *Data*
- *Name* is the name of the variable holding *Data* in the program
- *FuncName* is the name of the function currently being inspected
- *CallCounter* is the current value of the call counter of *FuncName*

inspectPointer

Signature: void inspectPointer(void* Ptr, int TypeIndex, char* Name, char* FuncName, int CallCounter)

This function is responsible to report a value pointed by a pointer in the program. If the instrumentation mode is FAST, this function only prints the type of the pointer. If the instrumentation mode is PRECISE, this function will track the pointer to print its contents. It checks if *Ptr* is pointing to an address stored in the heap table. If yes, the function to print heap data is called to inspect the value accordingly. Otherwise it checks if *Ptr* is pointing to a location within ELF segments, as 4.3 explains. In a positive case, this function retrieves the type descriptor accessing *T* using *TypeIndex* and calls `printData`, which will dereference *Ptr* and print its contents.

Parameters:

- *Ptr* is the value to be inspected
- *TypeIndex* is the index to access the type descriptor of *Ptr* in the type table
- *Name* is the name of the pointer in the program
- *FuncName* is the name of the function currently being inspected
- *CallCounter* is the current value of the call counter of *FuncName*

inspectUnion

Signature: void inspectUnion(char* Union, size_t Size, char* Name, char* FuncName, int CallCounter)

This function is in charge of inspecting variables of union type. As described in section 4.3, WHIRO prints unions as bitmaps. It iterates the pointer to *Union* and prints every byte of it.

Parameters:

- *Union* is the pointer to an union
- *Size* is the size of the largest composing part in the union type
- *Name* is the name of the variable holding *Data* in the program
- *FuncName* is the name of the function currently being inspected
- *CallCounter* is the current value of the call counter of *FuncName*

computeHashcode

Signature: int computeHashcode(void* Array, int TotalElements, int Step, int Format)

This method computes the hashcode for any multidimensional array of primitive types. It traverses the array entirely. It receives a pointer to the beginning of the array and increment it using a step value. Using this step, WHIRO can reach all the 1-D arrays inside this one. This function returns the hashcode value.

Parameters:

- *Array* is a pointer to the beginning of the array
- *Step* is the step that the base pointer must take
- *TotalElements* is the number of elements of the array
- *Format* is a value indicating the primitive type of *Array*, so WHIRO can cast each element to compute the hashcode.