Submitted by
Jonathan Lainer

Submitted at
Department of Computer Science

Supervisor and First Examiner
Prof. Dr.-Ing. **Christoph M. Kirsch**,
Department of Computer Science

Salzburg, June 10, 2024

SYMBOLIC BENCHMARKING FOR ALLOCATION EFFICIENCY

Master Thesis

to obtain the academic degree of

Diplom-Ingenieur

in the Master’s Program

Computer Science

PARIS LODRON
UNIVERSITY SALZBURG
Kapitelgasse 4-6
5020 Salzburg, Austria
www.plus.ac.at

Sworn Declaration

I hereby declare under oath that the submitted Diploma Thesis has been written solely by me without any third-party assistance, information other than provided sources or aids have not been used and those used have been fully documented. Sources for literal, paraphrased and cited quotes have been accurately credited.

The submitted document here present is identical to the electronically submitted text document.

J Lainer

Salzburg, Salzburg, June 10, 2024

Abstract

While there are many tools to optimize programs, they are either source agnostic, like optimizing compilers, or require the user to come up with representative inputs, like profile guided optimization. Symbolic execution could produce insights about a specific program, but all possible inputs, to close this gap.

This thesis evaluates the feasability of extending *unicorn*, an SMT based symbolic execution engine, to count memory and register accesses, thereby enabling the analysis of memory and register access patterns. As an additional constraint, the size of the generated SMT formula, and by extention the runtime of the program, should remain linear in the size of the size of the input program.

We design and implement an extension to unicorn that facilitates counting register accesses at first, then extending it to count memory accesses. Furthermore, we discuss the limitations inherent to the implementation of the chosen model. We show that the size of the generated SMT formula indeed remains linear in the size of the input program by providing a formal proof.

Our implementation enables checking for several patterns in binary RISC-V programs, like finding hot registers and memory locations.

Contents

List of Figures

List of Tables

Listings

Glossary

Boolector A state-of-the-art SMT solver.

BTOR2 A format for bit-precise modelling of word-level problems for model checking.

BtorMC A bounded model checker based on Boolector.

Executable and Linking Format A common file format for executable files.

jalr The jump and link register instruction.

one-hot A form of encoding a number using a set of bits, where only one of the bits can be active at a time.

RISC-U A minimal subset of the RISC-V ISA developed for teaching.

RISC-V A modern reduced instruction set computer (RISC) instruction set.

SAT The boolean satisfiablity problem.

theory of arrays A theory used in hardware and software verification, allowing to model `read` and `write` operations to arrays.

theory of bit-vectors A theory used in hardware and software verification, allowing to model fixed-width integers.

unicorn A platform for symbolic execution. It reads RISC-V binaries.

Acronyms

ABI	Application Binary Interface
BMC	bounded model checking
DAG	directed acyclic graph
ELF	Executable and Linking Format
ISA	instruction set architecture
ite	if-then-else
LTL	linear-time temporal logic
PGO	profile-guided optimization
RISC	reduced instruction set computer
SMT	Satisfiability Modulo Theories

1 Introduction

A lot of effort has been put into optimizing register allocation and devising good memory allocation strategies. Since these strategies have little knowledge about the specific program a future programmer might come up with, they have to be program agnostic.

The cost of bad allocation is high both in terms of time and energy, as it might require fetching a datum from memory instead from a cache. For programs that are deployed and run at scale, for example in data centers, the total cost can become a concern.

Current approaches to this problem, like profile-guided optimization (PGO), require the program to be run either in production environments or with inputs that is expected to approximate real inputs.

Methods like symbolic execution can reason about all code paths of a program, allowing to make absolute statements about the code considered.

We propose a way of quantifying memory access patterns independent of specific inputs, but rather for all possible inputs, providing a complementary approach to PGO. Through this lens we propose a simple measure for gauging allocation efficiency that tries to reduce traffic through the Von Neumann bottleneck by detecting low read utilization of registers.

Problem Statement

We are interested in determining register utilization for a fixed program but all possible inputs. To answer this question, our tool reduces reachability of machine states to satisfiability of Satisfiability Modulo Theories (SMT) formulas.

This premise leads to the following formulation of the research question, that guides this thesis:

> Can the symbolic reasoning engine unicorn be extended in such a way that it facilitates counting read accesses since the last write access for all registers and memory locations, reporting if certain properties about these counts are violated?

Properties this thesis explores involve checking that all registers are accessed most $k \in \mathbb{N}_0$ times, as well as its negation.

Aiming at a high ratio of read to write accesses for memory, for example, could translate to better cache usage, possibly improving program performance.

Furthermore, finding memory locations that are written twice in a row, without being read in between, might expose the contents to side-channel attacks[14].

Structure

As a core component of this thesis is the implementation, we outline the structure of this thesis below.

The chapter *Background* gives an overview of the theoretical topics relevant to this thesis as well as introduces the programs and formats it builds upon. In this chapter, later sections build on previous sections, so we advise reading it sequentially unless the reader has prior knowledge.

In the chapter *Implementation*, we will walk through the theoretical and practical considerations that go into implementing support for the aforementioned changes. We first generally introduce the properties chosen for this thesis and describe extension of the model of unicorn and implementation of the checks, and finally discuss limitations to the implementation. There are code snippets in the programming language Rust[15], but we believe that the code with its accompanying explanation is readable to anyone with basic knowledge of a programming language, and using the actual implementation language instead of pseudocode allows the reader to more easily find the corresponding code in the source code.

Chapter *Discussion* evaluates the results and suggests possible future work whose implementation and exploration were either out of scope or arose as a result of challenges faced during implementation.

2 Background

2.1 RISC-V

RISC-V is a freely licensed, open ISA [10], initially developed at the University of California, Berkeley [11]. In 2015, the RISC-V foundation was created to steward the development and contributions of the project. The latest version of the specification document [20] was released under a Creative Commons license to facilitate contributions to the project.

The ISA is split into modules. Everything builds on the *rv32i* or *rv64i* module, depending on the register size, which provides instructions loading, storing and manipulating integers[20]. A selection of available extensions is listed in Table 2.1. The extensions M and C are relevant to this thesis. Extension M introduces signed and unsigned integer multiplication. The C extension does not add any new instructions. Instead, it provides a more compact encoding for commonly used instructions, which reduces the size of programs by up to 25% [19].

Table 2.1: Selection of available extensions for the base RISC-V instruction set[20]

.

Module	Description
M	integer multiplication and division
A	atomic instructions
F	single-precision floating-point numbers
D	double-precision floating-point numbers
Q	quad-precision floating-point numbers
C	compressed instructions
B	bit manipulation
V	vector operations

Table 2.2: ABI names of RISC-V registers[20]

Register	ABI Name	Description
`x0`	`zero`	Hard-wired zero
`x1`	`ra`	Return address
`x2`	`sp`	Stack pointer
`x3`	`gp`	Global pointer
`x4`	`tp`	Thread pointer
`x5`	`t0`	Temporary/alternate link register
`x6-7`	`t1-2`	Temporaries
`x8`	`s0/fp`	Saved register/frame pointer
`x9`	`s1`	Saved register
`x10-11`	`a0-1`	Function arguments/return values
`x12-17`	`a2-7`	Function arguments
`x18-27`	`s2-11`	Saved registers
`x28-31`	`t3-6`	Temporaries

2.1.1 ABI

The RISC-V ISA has 32 general-purpose registers, named `x0`...`x31`. The size of the registers depends on the base variant used, either *rv32i* or *ri64i*. The base variant *rv128i* is still in draft status as of this writing. Although all registers except for `x0`, which is hard-wired to the literal value zero, can be used freely, when crafting a program by hand and in isolation, a convention is necessary if programs are to load shared libraries. Since this thesis will inspect compiler-generated binaries, and compilers follow the ABI, a basic understanding of the this interface is necessary. Table 2.2 lists the mnemonic register names and their uses in the first RISC-V calling convention[20] for reference.

2.2 Model Checking

Model checking is defined by [5] as a technique for "verifying finite state concurrent programs". In it's most abstract sense, it tries to answer the question of whether a given property φ is satisfied by a model M, usually written as $M \models \varphi$.

The model is usually represented as a Kripke structure, an extended transition system. Formally, a Kripke structure M is defined in [5] as a five-tuple (S, S_0, R, AP, L) where

- S is the set of states,
- $S_0 \subseteq S$ is the set of initial states,

- $R \subseteq S \times S$ is the transition relation,
- AP is the set of atomic propositions and
- $L : S \to \mathcal{P}(AP)$ is a function assigning to each state those atomic propositions that are true in that state.

Property φ is a temporal logic formula. Possible properties and temporal logic will be treated in the following subsections.

For this thesis, the model will always be a program in the form of machine instructions, which is interpreted relative to the instructions semantics as defined by the RISC-V ISA specification [20].

Example 2.2.1. As explained in Section 2.1, the state of a RISC-V machine consists of

- 32 64-bit registers, one of which has the constant value of zero and hence conveys zero bit of information,
- one 64-bit program counter, and
- 2^{64} 64-bit memory locations.

Hence, there are 2^{64} possible assignments for the program counter and every register except for the zero register, amounting to $32 \cdot 2^{64} = 2^{69}$ possible assignments to the set of 31 registers and the program counter[1].

Analogously, there are 2^{64} possible assignments for every one of the 2^{64} memory locations, amounting to $2^{64} \cdot 2^{64} = 2^{128}$ different assignments for memory.

Consequently, there are $2^{69} \cdot 2^{128} = 2^{197}$ possible assignments to registers and memory. Those assignments are the different states, meaning that $|S| = 31 \cdot 2^{197}$.

The transition relation is given by the semantics of the program.

The atomic propositions are the valuations of every register and memory location, which can each hold one of 2^{64} values. All atomic propositions represent the union of the possible assignments for each register and memory location. Therefore, the number of all atomic propositions in this example is $|AP| = (2^{64} + 32) \cdot 2^{64}$.

Finally, the labelling function assigns each state the following set of labels: Each register and memory location gets assigned a value between 0 and 2^{64}.

[1]As this calculation serves to illustrate the magnitude rather than precisely calculate the number of states in the system, we ignore the fact that the program counter actually has less than 2^{64} possible assignments because of alignment conditions.

Uninitialized memory manifests itself in the cardinality of the initial states. If all memory is initialized, then S_0 is a singleton set. Otherwise, the set contains all possible states whose labels satisfy the already-initialized registers and memory locations. ■

2.2.1 Temporal Logic

In our work, we restrict our treatment of temporal logic to linear-time temporal logic (LTL), the most widely used temporal logic in computer science [9]. The logic, which was first introduced by Pnueli in [18], follows the instance-based approach as opposed to an interval-based one.

LTL consists of the operators **X** ("next"), **G** ("globally"), **F** ("eventually"), **R** ("releases"), **U** ("until") and the path quantifier **A**. The path quantifier **E** is not allowed. However, since we will require it to properly negate our LTL formulas in our treatment of counterexamples, we will include and define it here.

An LTL formula can be either

- an atomic proposition $p \in AP$,
- negation of a formula,
- a unary operator applied to a formula ($\mathbf{X}f$, $\mathbf{G}f$ and $\mathbf{F}f$),
- a binary operator applied to two formulas ($f\mathbf{U}g$ and $f\mathbf{R}g$),
- or a conjunction or disjunction of formulas [5].

We restrict our treatment to **X**, **G**, and **F**, as they suffice to describe all properties relevant to this thesis.

The formal semantics of operators and quantifiers, as defined in [5], are reproduced in Equations 2.6 to 2.8 for operators and in Equations 2.9 and 2.10 for quantifiers. The necessary semantics of the operator $\models$ ("models") are explained below.

The notation $M, s \models \varphi$ denotes that state s in model M satisfies the formula φ. A state $s \in S$ satisfies an atomic proposition $p \in AP$ if the proposition is in its labels, i.e. $s \in L(p)$. A formal definition of the semantics for all connectives is presented in Equations 2.1 to 2.4.

$$M, s \models p \iff s \in L(p) \tag{2.1}$$
$$M, s \models \neg\varphi \iff M, s \not\models \varphi \tag{2.2}$$
$$M, s \models \varphi_1 \land \varphi_2 \iff M, s \models \varphi_1 \land M, s \models \varphi_2 \tag{2.3}$$
$$M, s \models \varphi_1 \lor \varphi_2 \iff M, s \models \varphi_1 \lor M, s \models \varphi_2 \tag{2.4}$$

The semantics of the operators is always relative to a (potentially infinite) path $\pi = s_0, s_1, s_2, \ldots$ which is a sequence of states $s_i \in S$. $s_0 in S_0$ and $\forall i \forall j\ R(s_i, s_j)$ must hold for any path. The notation π^k denotes the sequence $s_k, s_{k+1}, s_{k+2}, \ldots$.

$$M, \pi \models \varphi \iff s_0 \models \varphi \tag{2.5}$$
$$M, \pi \models \mathbf{X}\varphi \iff \pi^1 \models \varphi \tag{2.6}$$
$$M, \pi \models \mathbf{F}\varphi \iff \exists k \in \mathbb{N}\ M, \pi^k \models \varphi \tag{2.7}$$
$$M, \pi \models \mathbf{G}\varphi \iff \forall k \in \mathbb{N}\ M, \pi^k \models \varphi \tag{2.8}$$
$$M, s \models \mathbf{E}\varphi \iff \text{there exists a path } \pi \text{ starting from } s \text{ such that } M, \pi \models \varphi \tag{2.9}$$
$$M, s \models \mathbf{A}\varphi \iff \text{every path } \pi \text{ starting from } s \text{ satisfies } M, \pi \models \varphi \tag{2.10}$$

Finally, if every initial state s_0 satisfies a formula φ, then the model M satisfies the formula φ, and we write $M \models \varphi$. LTL restricts the formulas in the following way:

- every formula has to start with the quantifier **A** and
- formulas cannot contain any other quantifiers.

Since the first **A** quantifier is mandatory, it is often omitted. We will follow this convention, except when talking about counterexamples.

2.2.2 Properties

Two prominent properties in model checking are *safety* and *liveness* [5]. Such properties can be expressed as temporal logic formulas. Safety properties state that a bad condition is never met. They often take the form of $\mathbf{AG}\neg\varphi$. Liveness properties state that a desirable condition is met at some point, often taking the form of $\mathbf{AF}\varphi$. However, this formula is satisfied even if the property φ is only satisfied once. To ensure that the property is satisfied continually, the form $\mathbf{AGF}\varphi$ can be used. Here $\mathbf{GF}\varphi$, which literally means that the formula is satisfied “globally eventually” can be interpreted as “infinitely often” as at any instance, there has to be a future instance for which the formula is satisfied.

Far example, consider a concurrent program with two threads that simultaneously access a data structure protected by a lock, assuming interleaving semantics. A possible liveness property would be to require that at any point, if thread A has the lock, thread B will eventually get the lock, and vice versa. More formally, let L_A denote that thread A has the lock and L_B denote that thread B has the lock. Then the above property can be written as

$$\mathbf{AG}((L_A \implies \mathbf{F}L_B) \land (L_B \implies \mathbf{F}L_A)) \tag{2.11}$$

If we assume that each thread can terminate, we can incorporate this by letting T_A and T_B denote that thread A and thread B have stopped, respectively. The modified property, that each thread will get a lock in the future unless it is stopped, can be written as

$$\mathbf{AG}((T_B \lor (L_A \implies \mathbf{F}L_B)) \land (T_A \lor (L_B \implies \mathbf{F}L_A))) \tag{2.12}$$

A common safety property for such a setting is mutual exclusion, first described in [8]. Adapted for our example, it requires that the two threads cannot have a lock and thus access to the data structure at the same time. The undesirable behaviour of this property applied to our setting can be written as

$$\mathbf{AG}(\neg(L_A \land L_B)) \tag{2.13}$$

2.2.3 Bounded Model Checking

Bounded model checking (BMC) capitalizes on advances in SAT-solver performance [3]. It does not use the full model and property but rather an unrolled version of them, represented symbolically. An unrolled model is said to have depth k if all possible states that are reachable within k transitions are encoded in the formula. For those states, the property and the model are encoded into the same formula.

Furthermore, it exploits the fact that every LTL formula begins with the $\mathbf{A}$ quantifier. Hence, the negation of every LTL formula $\mathbf{A}\varphi$ is of the form $\mathbf{E}\neg\varphi$, which can be proven with an example. Which in turn means that $\mathbf{A}\varphi$ can be refuted by the model checker with a counter example if such a counter example exists and is finite. This indicates that the simple form of BMC is not complete. For ways to achieve completeness with BMC, see [3]. For this thesis, however, completeness is not required.

Combining those facts, the symbolic representation of the model and the negation of the property can then be fed to an industrial SAT-solver. If the verdict of the solver is *sat*, i.e. the formula is satisfiable, then there is a counterexample that refutes the property. Such a counterexample can be extracted from the solver. If the verdict is *unsat*, then either the bound was not large enough or the property holds for the model, but no conclusion can be drawn from this result. There is a third possible outcome: *timeout*. This outcome does not allow any conclusions.

But why use a less versatile model checker, if more powerful and complete model checkers exist? Because bounded model checkers are often able to handle larger formulas than classical model checkers [3].

On the flip side, they are limited in the classes of properties they can check. As alluded to earlier, bounded model checkers search for *finite* counterexamples. For properties like $\mathbf{AF}\varphi$, the negation has the form $\mathbf{EG}\neg\varphi$. A witness for $\mathbf{G}\neg\varphi$, however, would be an infinite path, and thus $\mathbf{AF}\varphi$ lies outside of what is refutable by BMC.

2.3 SMT

Improvements in SAT solver performance are the main reason for the success of bounded model checking. However, SAT solvers only accept formulas in propositional logic, whereas the semantics of models are often more naturally expressed using richer languages. [6] SMT extends the expressiveness of propositional formulas by introducing theories that translate between higher-level symbols and functions and interfacing with a SAT solver for solving these formulas.

2.3.1 Theories

In this thesis, we are modelling programs. The fundamental concepts underlying digital computers are memory and modular arithmetic. There are two theories that lend themselves to modelling those two concepts.

- The theory of arrays provides the two function symbols *read* and *write*, modelling read and write access to an array.
- The theory of bit-vectors provides operations on fixed-width numbers.

Here, *read*(a, j) denotes the value of the element in the array a at index i. The expression *write*(a, i, v) denotes an array that is like a, only with value v at index i [2]. Equation 2.14 and Equation 2.15 list the axioms of the theory for arrays a, indices i and j, and values v[2].

$$\forall a \forall i \forall v \ \mathrm{read}(\mathrm{write}(a, i, v), i) = v \tag{2.14}$$

$$\forall a \forall i \forall j \forall v \ (i \neq v \implies \mathrm{read}(\mathrm{write}(a, i, v), j) = \mathrm{read(a,j)}) \tag{2.15}$$

Besides modelling memory more naturally, the theory of arrays also impacts the size of the model. Rather than the model being proportional to the size of memory that is being modelled, the model is proportional to the number of accesses, which are usually much less than the size of memory.

We require the theory of bit-vectors primarily for its capability to directly model the semantics of many machine instructions involving fixed-width numbers.

While combining theories can be a difficult problem, since the two theories we require do not share symbols except for equality, they can be combined using the Nelson-Oppen framework. The resulting theory is referred to as *QF_ABV*, indicating that it is the theory of *arrays and bit-vectors without quantifiers*.

2.3.2 Solvers

There are multiple SMT solvers that implement decision procedures for this theory.

Z3[7] is an efficient, feature-rich SMT solver supporting many additional theories aside from arrays and bit-vectors.

Boolector is an efficient solver that supports significantly fewer theories than Z3, but has won multiple competitions for its performance in those theories. [**citation-needed**].

Since BtorMC, the model checker that we use for this thesis, uses Boolector as its backend, this is the solver that we use in our thesis.

2.4 BTOR2

The BTOR2 format was introduced in [17] as an attempt to standardize a format for bit-precise hardware and software model checking. It is an extension and generalization of the BTOR format, introduced in [4]. In contrast to formats like SMV, which is frequently used in hardware model checking, BTOR2 aims to specify clear and precise semantics. The format is line-based with every line being numbered. The numbers are used to refer to the nodes described by the line. For ease of parsing, BTOR2 does not allow forward declaration. All nodes have to be declared before they can be used.

2.4.1 Model

BTOR2 allows for the description of sequential and combinational circuits with registers and memories [17]. As a result, a BTOR2 model consists of states and transition functions for those states.

States, which model registers and memories, can be of different *sorts*. Sorts describe the type of information a state holds: a node accepts as input or emits as output. BTOR2 supports bit-vectors of arbitrary width, as well as arrays with bit-vectors as elements and indices. Bit-vectors and arrays are modelled using the theory of bit-vectors and theory of arrays, respectively. Each state can be explicitly initialized using the `init` keyword or left uninitialized. Uninitialized states are treated like inputs.

The transition function is a computational directed acyclic graph (DAG) that computes the new value of a state from the previous states and inputs. A selection of supported operators relevant to this thesis is reproduced in Table 2.3. The third column lists domain and co-domain of each operator.

BTOR2 has syntax to specify safety, liveness, and fairness properties in the model. For reasons explained in Subsection 2.2.3, safety and liveness properties, are not directly encoded but rather their negation. We will label those negations as *violations*. A safety violation can be encoded using the keyword `bad`, which introduces a node into the model that takes a single-bit bit-vector as input. If the input of the state is set, the safety property is violated. Fairness properties, modelled using the `fair` keyword, and liveness properties modelled using the `justice` keywords, are not relevant to this thesis. For their semantics, see [17].

To encode a safety property in the model, the semantics of its negation have to be modelled as a computational DAG of BTOR2 nodes such that the result is true if and only

Table 2.3: Excerpt of supported BTOR2 operands adapted from [17]. $\mathcal{S}$ represents any sort, $\mathcal{B}^n$ represents a bit-vector of size n and $\mathcal{A}^{\mathcal{I}\to\mathcal{E}}$ represents an array sort with index sort $\mathcal{I}$ and element sort $\mathcal{E}$.

indexed		
`uext` w	unsigned extension	$\mathcal{B}^n \to \mathcal{B}^{n+w}$
unary		
`not`	bit-wise negation	$\mathcal{B}^n \to \mathcal{B}^n$
binary		
`eq`	equality	$\mathcal{S} \times \mathcal{S} \to \mathcal{B}^1$
`ult`	unsigned less-than	$\mathcal{B}^n \times \mathcal{B}^n \to \mathcal{B}^1$
`and`, `or`	bit-wise logic operations	$\mathcal{B}^n \times \mathcal{B}^n \to \mathcal{B}^n$
`sll`, `srl`	logical shifts	$\mathcal{B}^n \times \mathcal{B}^n \to \mathcal{B}^n$
`add`, `mul`, `[su]div`, `[u]rem`, `sub`	arithmetic	$\mathcal{B}^n \times \mathcal{B}^n \to \mathcal{B}^n$
`read`	array read	$\mathcal{A}^{\mathcal{I}\to\mathcal{E}} \times \mathcal{I} \to \mathcal{E}$
ternary		
`ite`	conditional	$\mathcal{B}^1 \times \mathcal{B}^n \times \mathcal{B}^n \to \mathcal{B}^n$
`write`	array write	$\mathcal{A}^{\mathcal{I}\to\mathcal{E}} \times \mathcal{I} \times \mathcal{E} \to \mathcal{A}^{\mathcal{I}\to\mathcal{E}}$

if the model satisfies the property. This output is then connected to the bad-state node. In this thesis, we will focus on bad states as a means to assert certain properties about a model, and by extension, a program.

Listing 2.1 shows the model of a simple adder with an added safety property $\mathbf{AG}(\text{result} \neq 2)$. Observe that the numbers attached to each line must be strictly monotonically increasing, and every *nid* or *sid* that is used in that line must be at least as large as the number of that line. Furthermore, since line numbers must be increasing, the set of node ids and sort ids must be disjunctive. It first declares the sorts used in the model in lines 2 and 3. The following lines define a 64-bit bit-vector input and a 64-bit bit-vector state that gets initialized with zero. The block starting on line 11 models the transition function for state *result*. In this example, the result register acts as an accumulator that accumulates the values of the input. The last block models the negation of the safety property, namely $\mathbf{EF}(\text{result} = 2)$.

Listing 2.1: BTOR2 model of addition with safety property $\mathbf{AG}(\text{result} \neq 2)$

```
1  ; File example.btor2
2  1 sort bitvec 1 ; Boolean
3  2 sort bitvec 8 ; Byte
4
5  10 constd 2 0
6  11 input 2 value
```

```
12 state 2 result
13 init 2 12 10

; The transfer function for result
14 add 2 12 11
15 next 2 12 14

; The property AG(result != 2)
16 constd 2 2
17 eq 1 12 16
18 bad 17
```

2.4.2 Witness Format

The verdict that a property has been violated is not always enough. Information about how the model checker arrived at the verdict can be obtained from witnesses, a counterexample to the property. BTOR2 specifies a format for a witness that allows tools like the witness checker to verify and further inspect the model and property. In Subsection 3.2.3 we will use a witness to extract more detailed information from a witness.

A witness always starts with the string `sat`, followed by the properties that are satisfied and a sequence of frames, one for each time step and one for the initial state. The properties are the *bad* nodes that were introduced into the model. Each frame lists all assignments to uninitialized states, followed by all assignments to inputs. The state part begins with `#t` and is delimited from the input part by `@t`, where `t` is the step number.

Listing 2.2 shows the result of checking the model from Listing 2.1 with the bounded model checker BtorMC, which will be described in the next section.

Listing 2.2: Example witness

```
sat
b0
@0
0 00000010 value@0
@1
0 00000000 value@1
.
```

The command to obtain the witness for the model is `btormc example.btor2`. The witness indicates that the first bad state in the model was satisfied. Since all states were initialized, the first frame only shows the assignment for the input *value*. Remembering that the model accumulates the values of input into the register *result*, which was zero initialized, the assignment of two makes the value of the result register two. However, this change is not visible to the bad state in this time frame, as the calculation yields the value of the register in the next time step, when the property will eventually be violated. And finally, the witness is terminated with a dot in line 7.

2.4.3 Tools

We use the reference implementations of a BTOR2 bounded model checker and witness checker provided by [17] in this work.

The bounded model checker BtorMC uses the SMT solver Boolector[16] as a back end. It supports the quantifier-free theories of bit-vectors and arrays.

There are some useful command-line flags for BtorMC. By default, it will stop when it finds a witness for any bad state. This behaviour can be changed using the command-line flag `-checkall`. Furthermore, the default bound is set to 20 steps, which can be changed using `-bound-max=<n>`.

The simulator btorsim takes both the BTOR2 model and the witness as arguments and checks if the witness actually satisfies the stated property. Additionally, when using the command-line flag `--states`, it prints the values of all states, even those that were initialized initially. This will be essential for further analysis in Subsection 3.2.3.

2.5 Selfie

The Selfie Project is an educational framework created by professor Christoph Kirsch, head of the Computational Systems Group at Paris Lodron University Salzburg. [13]

Originally developed to target the MIPS ISA, the project was later ported to use the RISC-V ISA. For teaching purposes, the instruction set was reduced to RISC-U, a minimal working subset of 14 RISC-V instructions [1]. Currently, the compiler emits ELF binaries that are compatible with the official specification [20].

The project consists of four core parts, which are all implemented in a single C* file with less than 10 000 lines of code:

- *starc*, a single pass compiler for C*,
- *mipster*, an emulator for the RISC-U instruction set,
- *hypster*, a hypervisor and
- *libstarc*, a small "standard" library shared between all components.

The language C* is a simple subset of the C language [12], whose grammar is LL(1).

In addition, the project also contains *babysat*, a minimal SAT solver; *monster*, a symbolic execution engine utilizing an SMT solver as a back end; and *beator*, a front end for bounded model checking that emits BTOR2 formulas of the program, which include checking for non-zero exit codes, division by zero, and memory access outside of allocated blocks, among others. Furthermore, we use the *starc* compiler to generate binaries for use in this thesis.

2.6 Unicorn

Unicorn is a platform for symbolic execution that takes inspiration from both *beator* and *monster* of the selfie project. It takes Executable and Linking Format (ELF) binaries as input, models them, optionally unrolls the sequential model and produces a BTOR2 model as output. Figure 2.1 graphically depicts an overview of unicorn's stages in context with the additional tools used to achieve this thesis' goal.

Unicorn is written in the Rust programming language[15].

Parsing

Unicorn parses binaries that conform to the *rv64imc* RISC-V instruction set. All instructions contained in this instruction set are listed in Table 2.4. Support for instructions was added as needed. Hence, there are still unimplemented instructions. It fully supports binaries compiled by *selfie*, but has also been tested with binaries from gcc.

Table 2.4: Instructions of the *rv64imc* ISA[20]. Instructions marked with a star (*) are not yet supported by unicorn.

Extension	R-type	I-type	S-type	B-type	U-type	J-type
rv32i	ADD	JALR	SB	BEQ	LUI	JAL
	SUB	LB	SH	BNE	AUIPC	
	SLL	LH	SW	BLT		
	SLL	LW		BGE		
	SLT*	LBU		BLTU		
	SLTU	LHU		BGEU		
	XOR*	ADDI				
	SRL	SLTI*				
	SRA	SLTIU				
	OR	XORI				
	AND	ANDI				
	FENCE*	SLLI				
	ECALL	SRLI				
	EBREAK*	SRAI				
rv64i	ADDW	LWU	SD			
	SUBW	LD				
	SLLW	SLLI				
	SRLW*	SRLI				
	SRAW*	SRAI				
		ADDIW				
		SLLIW				
		SRLIW				
		SRAIW				
rv32m	MUL					
	MULH*					
	MULHSU*					
	MULHU*					
	DIV					
	DIVU					
	REM*					
	REMU					
rv64m	MULW					
	DIVW					
	DIVUW*					
	REMW*					
	REMUW*					

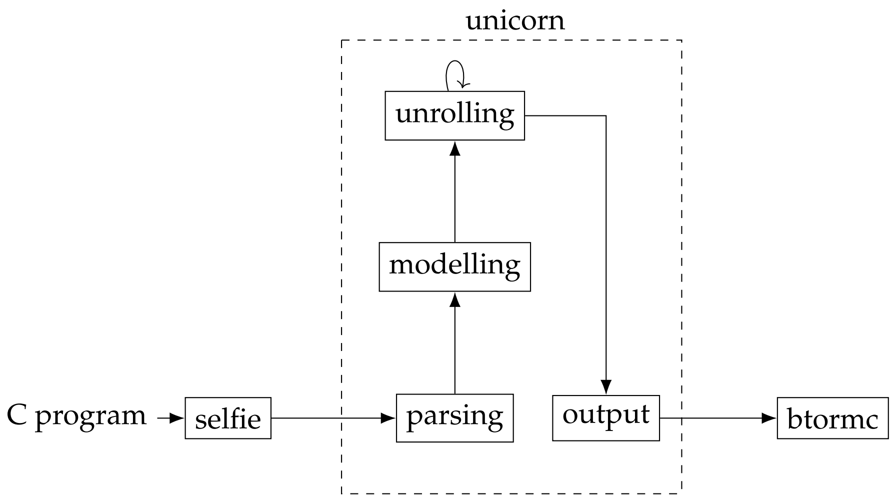

Figure 2.1: Overview of the components of unicorn in the context of this thesis.

Model

After parsing, unicorn builds a sequential model of the program. This model is an internal representation of the model of BTOR2, introduced in Section 2.4. After modelling, unicorn has a full model of the bit-precise semantics of the program for an arbitrary number of steps. This model is enriched with additional computations to allow checking properties like out-of-bounds access checking.

Unrolling

As mentioned in Subsection 2.2.3, BMC is performed on an unrolled model. Unicorn can unroll the model by itself. For this thesis, we are using the bounded model checker as a downstream checker, which performs the unrolling of the model anyway. However, downstream tasks like the evaluation of the resulting model on a quantum computer do not have the capability.

Output

The resulting, potentially unrolled model can be output in different formats, tailored to the intended further processing. For this thesis, we perform further analysis using BTOR2, hence we let unicorn output in the BTOR2 format.

3 Implementation

This thesis's implementation is an extension of the unicorn symbolic execution engine, enabling the tracking of memory accesses in the model. We extend the internal model of unicorn and the translation of that model to a BTOR2 model to contain information about the number of read and write accesses and add queries that check if the property is satisfied.

For a general formulation, we use the term *register* to refer to either one of the 32 registers of the RISC-V ISA or a memory location, since they differ only in their implementation but are similar in their ability to hold a datum of the size of a machine word.

We implement checking the following property:

Property 1. *A register is read less than or equal to $k \in \mathbb{N}_0$ since the last write access.*

When writing the property as an LTL formula, we must be more specific. Firstly, we want the property to hold for all inputs and, hence, all possible program executions. Secondly, we want the property to hold at any point in the program. Equation 3.1 describes the property with the added clarifications, assuming that the variable *registerReadCounter* holds the number of read accesses to the register since the last write access.

Property 2 (Property 1 expressed as an LTL formula.)**.**

$$\boldsymbol{AG}(\text{registerReadCounter} \leq k) \qquad k \in \mathbb{N}_0 \tag{3.1}$$

The property has two dual interpretations, depending on the bounded model checker's verdict. If the model checker cannot find a counter example, then the property holds for the program for an execution depth of k instructions, independent of inputs to *read* operations.

Dually, if the model checker can find a counter example, that is, a witness to the negation of the property, then the witness provides the inputs required to steer the program into the undesired state. The model checker also provides information about the register or memory location that violated the property.

As explained in Subsection 2.2.3, the verdict *timeout* does not allow any conclusions about the property, only indicating a lack of resources.

The dual property of Property 2 can be obtained by flipping the comparator from less than or equal ($\leq$) to greater than or equal ($\geq$).

We will begin with the implementation of modelling registers, as this requires fewer SMT theories than the modelling of memory.

Subsequently, we will extend our approach to memory and discuss the differences and hurdles that arise from the change.

3.1 Register Accesses

The RISC-V architecture has 32 registers, 31 of which can change their contents during program execution. Those changes might be explicitly due to an instruction in the program, such as an immediate instruction, like `ld`, or implicitly, like the program counter increasing by four after an instruction has been executed.

The register `r0`, also called `zero`, always contains the literal value zero. Hence, we omit modelling for the zero register, since writing to the zero register does not change the state of the system.

Each register is modelled in two parts in the unicorn model: its state and its update function. The state of a register represents the value that it contains at a given point and is modelled using a bit-vector. The update function is represented as a cascade of conditionals. For each instruction that might change the register's contents, an *ite* node enables the instruction's effect only if the program counter bit corresponding to the instruction is active.

For example, consider the simple instruction `addi t0, t0, 1` at location `0x104` in isolation. Figure 3.1 depicts the model's nodes that are relevant to modelling said instruction. The rectangular boxes represent the stateful components of the system, whereas the ellipses represent the update function. In this particular case, the *t0* box represents what is called the *register node*, whereas the *ite* node, being the last node with incident edges, is called the *register flow*. Solid edges represent data flow during the computation of the *register flow* node's value. The dashed line indicates that the calculated value will be the future value of the state node *t0*.

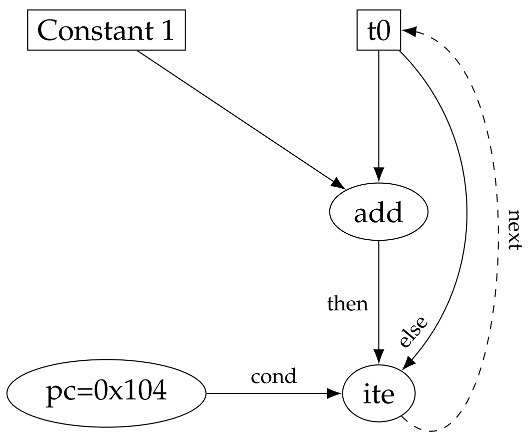

Figure 3.1: Model of the simple program `0x104:  addi t0, t0, 1`

Since the effects of each instruction do not have to appear in order and the effects do not interfere with the effects of other instructions, modelling more instructions is simply composing their update functions. This is also a result of the fact that only one instruction can be active at a given time. All other effects act as the identity if their program counter bit is not active. Furthermore, since every function commutes with the identity function and every function except the one that is active acts as the identity function, the order of composition is not important.

3.1.1 Modelling Counters

Looking at the previous example (cf. Figure 3.1), we can observe that the update graph only selects the effect of the instruction if the corresponding program counter bit is active. Since the program counter bit corresponds to the program counter's numeric value, only the effect of one single instruction can be active at a time. Furthermore, an instruction in the RISC-V ISA can only modify one machine word, that is, 64 bits, of the processor's state per clock cycle explicitly. Hence, to count the number of register accesses, it suffices to determine them for individual instructions and mirror the structure of the registers in the model.

We extend the model by adding a state that counts the number of read accesses for each register. We do not implement a counter for write accesses, as they are not required to check the desired properties outlined at the beginning of this section. To counter the number of read accesses, we have to determine which instructions perform read accesses and increment the register's counter when the instruction's program counter bit is active.

Of the six types of instructions outlined in Table 2.4, only four can read values from registers, namely R-type, I-type, S-type, and B-type instructions. This means that we need to consider 35 instructions from the *rv32i* module, 13 instructions from the *rv64i* module, four instructions from the *rv32m* module, and two instructions from the *rv64m* module. This amounts to a total of 53 instructions.

In the case of both source registers being the same, we count both occurrences as read accesses, even though this might only cause a single read in an actual processor. We will discuss the special case where a source and the destination register coincide in Subsection 3.1.4.

In unicorn, the model is built during a forward pass over instructions in the code segment of the binary. For every instruction, there is a `model_[instruction]` method, which adds the effects of the instruction to the model. For illustration purposes, consider the implementation of the `model_add` method, shown in Listing 3.1.

Listing 3.1: Implementation of `model_add`

```rust
impl ModelBuilder {
    // [...]
    fn model_add(&mut self, rtype: RType) {
        let add_node = self.new_add(
            self.reg_node(rtype.rs1()),
            self.reg_node(rtype.rs2())
        );
        self.reg_flow_ite(rtype.rd(), add_node);
    }
    // [...]
    fn reg_flow_ite(&mut self, reg: Register, node: NodeRef) {
        let ite_node = self.new_ite(
            self.pc_flag(),      // Condtion
            node,                // Consequence (if branch)
            self.reg_flow(reg),  // Alternative (else branch)
            NodeType::Word
        );
        self.reg_flow_update(reg, ite_node);
    }
    // [...]
}
```

Up until this point, the struct `ModelBuilder` maintains an internal representation of the model. The method creates a new node according to the instruction's semantics and introduces it into the destination register's update function. This effect is only applied if the program counter bit corresponding to the instruction is active. Otherwise, the previous value is forwarded through the *ite* node's *else* branch.

To augment the model with the register access counting capability, we introduce the method `reg_count_read_if_not_zero`. This method, shown in Listing 3.2, takes a register as an input and extends the update function of the register access counter in such a way that it increments the counter by one if the program counter bit of the current instruction is active. The special case with the zero register, as mentioned earlier, is due to the implementation of the modelling of registers in unicorn prior to this work.

Listing 3.2: Implementation of `reg_read_counter_flow_ite`

```
impl ModelBuilder {
    // [...]
    fn reg_read_counter_flow_ite(&mut self, reg: Register) {
        let increment_counter_node = self.new_add(
            self.reg_read_counter_node(reg),
            self.one_word.clone()
        );
        let ite_increment_counter_node = self.new_ite(
            self.pc_flag(),
            increment_counter_node,
            self.reg_read_counter_flow(reg),
            NodeType::Word,
        );

        self.reg_read_counter_flow_update(reg, ite_increment_counter_node);
    }

    fn reg_count_read_if_not_zero(&mut self, reg: Register) {
        if reg != Register::Zero {
            self.reg_read_counter_flow_ite(reg);
        }
    }
    // [...]
}
```

Every method that models an instruction takes the concrete instruction type as a parameter. Hence, we know which registers are involved and can add counting the register access to the update function of the corresponding register counter. This amounts to simply adding the lines emphasized in Listing 3.3 to every instruction that models an instruction of either R-type, I-type, S-type, or B-type. For a list of all instructions, refer to Table 2.4 in Chapter 2.

Listing 3.3: Added Counting Of Registers to `model_add`

```
impl ModelBuilder {
    // [...]
    fn model_add(&mut self, rtype: RType) {
        let add_node = self.new_add(
            self.reg_node(rtype.rs1()),
            self.reg_node(rtype.rs2())
        );
        self.reg_flow_ite(rtype.rd(), add_node);

+       self.reg_count_read_if_not_zero(rtype.rs1());
+       self.reg_count_read_if_not_zero(rtype.rs2());
    }
    // [...]
}
```

So far, we extended unicorn to count the number of read accesses to a register. In the next step, we will reset the count when a register write happens.

3.1.2 Modelling Reset

Just as the counter is increased when a read access happens, it must be reset to zero when a write access happens. Conceptually, this would require that we add a new state to the model and find every instruction that can issue write access to a register.

However, because unicorn models all register write accesses, which correspond to changing the update function of a register through a uniform interface, we only have to alter the function `reg_flow_ite`, which is shown in Listing 3.4.

Listing 3.4: Implementation of `reg_flow_ite`

```
impl ModelBuilder {
    // [...]
    fn reg_write_counter_flow_ite(&mut self, reg: Register) {
        let increment_counter_node = self.new_add(
            self.reg_write_counter_node(reg),
            self.one_word.clone()
        );
        let ite_increment_counter_node = self.new_ite(
            self.pc_flag(),
            increment_counter_node,
            self.reg_write_counter_flow(reg),
            NodeType::Word,
        );

        self.reg_write_counter_flow_update(reg, ite_increment_counter_node);
    }
    // [...]
    fn reg_flow_ite(&mut self, reg: Register, node: NodeRef) {
        let ite_node = self.new_ite(
            self.pc_flag(),
            node,
            self.reg_flow(reg),
            NodeType::Word
        );

        self.reg_flow_update(reg, ite_node);
        % self.reg_write_counter_flow_ite(reg);
    }
    // [...]
}
```

Because of this generic implementation of `reg_flow_ite`, all write accesses can be intercepted at the same spot. Simply adding the lines in Listing 3.5 to every instruction suffices to reset the counter. Here, the function `reg_write_counter_flow_ite` closely mirrors its read sibling in Listing 3.2. The code change closely mirrors the one presented in Listing 3.3 as the only differences lie in the affected instructions and the number of registers that can be accessed in a single instruction.

Listing 3.5: Added Counting Of Write Accesses to `reg_flow_ite`

```
impl ModelBuilder {
    // [...]
    fn reg_flow_ite(&mut self, reg: Register, node: NodeRef) {
        let ite_node = self.new_ite(
            self.pc_flag(),
            node,
            self.reg_flow(reg),
            NodeType::Word
        );

        self.reg_flow_update(reg, ite_node);
+       self.reg_write_counter_flow_ite(reg);
    }
    // [...]
}
```

We have to introduce a new state and cannot directly update the update function of the read counter since the order in which we increment the read counter and reset it has to reflect the semantics of the processor's implementation. When looking at the conceptual implementation of a modern processor, read accesses to the register file happen before the register write access corresponding to the same instruction. Therefore, we have to augment the update functions of the read and write counters after generating the model. The implementation of this augmentation is shown in Listing 3.6.

The listing depicts the final step in augmenting the unicorn model with the counters. First, the node `read_counter_update_node` is constructed. It is an ite node that either selects the zero node if the `reg_write_counter_flow` bit is set or the value computed by the corresponding update function. Secondly, the counter gets connected to the output of this ite node's value after a cycle. Lastly, the `reg_write_counter_node` gets connected with the static `zero` node, resetting it every cycle. It only indicates that a write access occurred in the current cycle.

Listing 3.6: Excerpt from `generate_model` (Updating Counters)

```
impl ModelBuilder {
    //[....]
    fn generate_model(&mut self, program: &Program, argv: &[String]) -> Result<()> {
        //[....]
        for r in 1..NUMBER_OF_REGISTERS {
            self.current_nid = 65000000 + (r as u64);
            let reg = Register::from(r as u32);

            let read_counter_update_node = self.new_ite(
                self.reg_write_counter_flow(reg),
                self.zero_word.clone(),
                self.reg_read_counter_flow(reg),
                NodeType::Word,
            );

            self.new_next(
                self.reg_read_counter_node(reg),
                read_counter_update_node,
                NodeType::Word,
            );

            self.new_next(
                self.reg_write_counter_node(reg),
                self.zero_bit.clone(),
                NodeType::Bit,
            );
        }
        //[....]
    }
    //[....]
}
```

3.1.3 Checked Properties

As alluded to in the introduction to this section, we implement checking two complementary properties for registers: Checking if the number of read accesses between two write accesses to a register exceeds or falls short of a previously chosen value. The condition is computed on the graph and fed into a bad state. In Listing 3.7, we show the corresponding code for modelling the property for registers. It creates a node for each register, which compares the number of read accesses to the limit set by the constant `READ_UNTIL_WRITE_LIMIT`. The listing outlines the implementation for the property

$$\mathbf{AG}(\text{readCount} \leq \text{readCountLimit}) \tag{3.2}$$

expressed in LTL. Recall from Subsection 2.2.3 that we formulate the property as a valid LTL formula, but encode the negation of that property into the model to search for a counter example. The formula we encode in the model is therefore

$$\mathbf{EF}(\text{readCount} > \text{readCountLimit}) \tag{3.3}$$

To formulate Equation 3.3 in the context of our problem: The formula is true for a given register if there exists an execution of the program for which the register is read more than readCountLimit times before the counter is reset.

The property's other variant, namely

$$\mathbf{AG}(\text{readCount} \geq \text{readCountLimit}) \tag{3.4}$$

can be obtained by modifying line 10 in Listing 3.7 from `self.new_ugt` to `self.new_ult`. However, without an extension, it is not useful. The property requires that the read count, which is initialized to zero at the beginning of the symbolic execution of the program, is always greater than zero, which is violated instantly. It can be altered to provide more useful insight, requiring the property to hold after some time has elapsed.

$$\mathbf{AG}((\text{pc} < 10100_{16}) \lor (\text{readCount} \geq \text{readCountLimit})) \tag{3.5}$$

This would ensure that there are always more read accesses than write accesses after a certain warm-up period has passed [1], but its implementation is outside of the scope of this thesis and would require significant changes to this architecture.

If the output of the comparison is true, our original safety property is violated, its negation satisfied, and the engine evaluating the model reacts appropriately.

Listing 3.7: Excerpt from `generate_model` (Checking Property for Registers)

```
1 impl ModelBuilder {
2     //[....]
3     fn generate_model(&mut self, program: &Program, argv: &[String]) -> Result<()> {
4         //[....]
5         const READ_UNTIL_WRITE_LIMIT: u64 = /* a number */;
6 
7         let limit_node = self.new_const(READ_UNTIL_WRITE_LIMIT);
8         for r in 1..NUMBER_OF_REGISTERS {
9             let reg = Register::from(r as u32);
```

[1]This simplistic condition does not account for loops and jumps.

```
            let check = self.new_ugt(
                self.reg_read_counter_node(reg),
                limit_node.clone()
            );

            self.new_bad(
                check,
                &format!(
                    "register-{:?}-reads-until-write-more-than-{}",
                    reg, READ_UNTIL_WRITE_LIMIT
                ),
            );
        }
        //[....]
    }
    //[....]
}
```

Checking the property requires two steps. First, unicorn generates a model from the executable. If the option *-s boolector* is selected, it will run boolector during the unroll steps and report if a bad state is either satisfiable or always true. Subsequently, BtorMC checks the model and reports which bad state, if any, is reachable. If a bad state can be reached, it provides a witness. Using Boolector in the first step is not required, but has the potential to significantly reduce the model size.

3.1.4 Limitations

Performing formal reasoning of the semantics of programs at the instruction level has the drawback that some of the semantics are only specified up to a certain point to allow for optimization by vendors that implement a processor using the architecture.

In our case, this means that, by virtue of abstracting the semantics between cycles as a computation graph, we can only reason about the state of the processor after a full cycle.

Consider the instruction `add t0, t1, t0` at location `0x104` in isolation. During a single processor cycle, registers `t0` and `t1` are read, and register `t0` is written. The model of this instruction is shown in Figure 3.2.

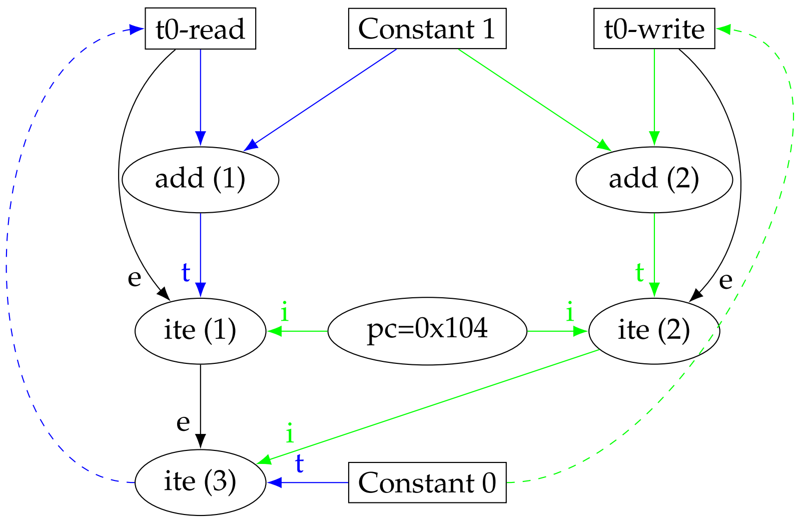

Figure 3.2: Model of the counters for register `t0` for the simple program `0x104:  add t0, t1, t0`. The ite node edges are labelled by the initials of their respective function: *i* for if, *t* for then, *e* for else.

Note that, whereas in Figure 3.1 we depicted the update graph for the register itself, in Figure 3.2 we show the update graph of the counter for register `t0`. For simplicity, we label the state nodes as `t0-read` for the read counter and `t0-write` for the write flag. We color edges that are active based on the information they carry: Bit information is shown in green, whereas word information is shown in blue.

Since the register `t0` is being written in this instruction, we would expect that its read counter would be incremented by one. But inspecting the graph in Figure 3.2, we see that the edge between the next value of the counter is coming from the node *ite (3)*, which is the *read flow* node in the model. This if-then-else node receives an incoming signal from the write flow, which selects the constant value *zero* instead of the incremented count value. Thought the counter value is incremented in the first part of the update function, the reset triggered by the *write flow* node being active takes precedence over the increment.

This observation limits the precision with which our predicates can be checked in this special case. It prevents us from checking the predicates' special cases.

$$\mathbf{AG}(readCount \leq 1) \tag{3.6}$$

and

$$\mathbf{AG}(readCount \geq 2) \tag{3.7}$$

in the presence of such instructions.

However, the presence of such instructions can be easily checked by a simple program that scans the binary beforehand, or by embedding such a check into the parser of unicorn. Hence, this limits only the programs that can be checked in those special cases, not the correctness of the approach in general.

3.2 Memory Accesses

In principle, registers and memory locations are very similar. Both typically hold a single datum of the same size and can be read and written. However, modelling access counters for memory locations explicitly in the same way as registers inflates the size of the model needlessly. Instead, we can take a look at the implementation of main memory itself in the model and, as with the registers, model the counters after it.

Unicorn abstracts main memory as an array of words, and it uses the theory of arrays to model all memory locations in a single object. Using this theory allows us to use calculated values as an index into the array and read or write to this location.

3.2.1 Modelling Counters

Since RISC-V is a load-store architecture, only one memory location can be accessed in a cycle. This access can be either a write or a read operation. Hence, counting main memory accesses does not suffer from the limitations mentioned in Subsection 3.1.4.

Furthermore, there are only two types of instructions that can trigger a main memory read or write: I-type and S-type instructions. More precisely, only variants of the load and store instructions can trigger memory access. Conveniently, the theory of arrays only allows for read and write accesses to items, which are word-sized. Unicorn mirrors this in its implementation with the `new_read` and `new_write` methods. This further reduces the implementation effort to extending those functions.

Therefore, modelling the counters for main memory locations is done by creating an array the same size as main memory and extending the functions `new_read` and `new_write` and store instructions to augment the update functions of the counters. Again, we only have to count read accesses and reset the counter on write accesses. The implementation of modelling the counters is shown in Listing 3.8 and Listing 3.9.

For the implementation of `new_read`, we first read the current value of the counter. Because we mirror the implementation of modelling memory, we use the same address to model indexing into the counter array in line 6. Subsequently, we model the next value of the counter in line 12, which is the previous value plus one. In line 17 models saving the counter value to the array of counters. As the counter is only incremented if the correct program counter flag is active, we create an ite node that bypasses the modification if the flag is not active (line 24) and saves this node to the update graph of the memory counter.

Listing 3.8: Implementation of `new_read`

```
1  impl ModelBuilder {
2      // [...]
3
4      fn new_read(&mut self, address: NodeRef, should_count: bool) -> NodeRef {
5          if should_count {
6              let current_value_node = self.add_node(Node::Read {
7                  nid: self.current_nid,
8                  memory: self.memory_counter_node.clone(),
9                  address: address.clone(),
10             });
11
12             let next_value_node = self.new_add(
13                 current_value_node,
14                 self.one_word.clone()
15             );
16
17             let update_counter_node = self.add_node(Node::Write {
18                 nid: self.current_nid,
19                 memory: self.memory_counter_node.clone(),
20                 address: address.clone(),
21                 value: next_value_node.clone(),
22             });
23
24             self.memory_counter_flow = self.new_ite(
25                 self.pc_flag(),
26                 update_counter_node,
27                 self.memory_counter_flow.clone(),
28                 NodeType::Memory,
29             );
30         }
31 
32         self.add_node(Node::Read {
33             nid: self.current_nid,
34             memory: self.memory_node.clone(),
35             address,
36         })
37     }
38     // [...]
39 }
```

The implementation of `new_write` is similar to `new_read`, with the difference that no value has to be read or incremented. To reset the counter, it suffices to create a node that writes the zero word to the counter (line 6) and gate it using an ite node that bypasses the modification if the flag is not active (line 13).

Listing 3.9: Implementation of `new_write`

```
 1 impl ModelBuilder {
 2     // [...]
 3     fn new_write(&mut self, address: NodeRef, value: NodeRef, should_count: bool)
 4     -> NodeRef {
 5         if should_count {
 6             let reset_counter_node = self.add_node(Node::Write {
 7                 nid: self.current_nid,
 8                 memory: self.memory_counter_node.clone(),
 9                 address: address.clone(),
10                 value: self.zero_word.clone(),
11             });
12 
13             self.memory_counter_flow = self.new_ite(
14                 self.pc_flag(),
15                 reset_counter_node,
16                 self.memory_counter_flow.clone(),
17                 NodeType::Memory,
18             );
19         }
20 
21         self.add_node(Node::Write {
22             nid: self.current_nid,
                memory: self.memory_node.clone(),
                address,
                value,
            })
        }
        // [...]
}
```

3.2.2 Limitations

As mentioned in Subsection 2.3.2, Boolector only supports the quantifier-free theory of arrays and bit-vectors. Since BtorMC uses Boolector as a back end and this thesis relies on BtorMC for its model checking, we cannot use quantifiers to express properties about the array of counters. An example of a property we would like to check is, for some $k \in \mathbb{N}$,

$$\forall i \ \mathbf{AG}(\text{memoryCounter}[i] \leq k) \tag{3.8}$$

A witness to a violation of this property would tell us which memory location was written more than k times in an execution.

To avoid using quantifiers in the property, we introduce a new state tracking the maximum of the array of counters. When updating the counter, if the new value is greater than the maximum, we store it as the next maximum.

Listing 3.10 shows the changes to `new_read`, as presented in Listing 3.8, that enable eliminating the need for the quantifier to model the property in Equation 3.8.

Listing 3.10: Implementation of `new_read` with maximum tracking

```
impl ModelBuilder {
    // [...]

+   fn memory_counter_max_update(&mut self, node: NodeRef) {
+       let should_update_node = self.new_ugt(
+           self.memory_counter_max_node,
+           node.clone()
+       );
+
+       let update_node = self.new_ite(
+           should_update_node,
+           node,
+           self.memory_counter_max_node.clone(),
+           NodeType::Word
+       );
+
+       self.memory_counter_max_flow = self.new_ite(
+           self.pc_flag(),
+           update_node,
+           self.memory_counter_max_flow.clone(),
+           NodeType::Word
+       );
+   }

    fn new_read(&mut self, address: NodeRef, should_count: bool) -> NodeRef {
        if should_count {
            let current_value_node = self.add_node(Node::Read {
                nid: self.current_nid,
                memory: self.memory_counter_node.clone(),
                address: address.clone(),
            });

            let next_value_node = self.new_add(
                current_value_node,
                self.one_word.clone()
            );

+           self.memory_counter_max_update(next_value_node.clone());

            let update_counter_node = self.add_node(Node::Write {
                nid: self.current_nid,
                memory: self.memory_counter_node.clone(),
                address: address.clone(),
                value: next_value_node.clone(),
            });

            self.memory_counter_flow = self.new_ite(
                self.pc_flag(),
                update_counter_node,
                self.memory_counter_flow.clone(),
                NodeType::Memory,
            );
        }

        self.add_node(Node::Read {
            nid: self.current_nid,
            memory: self.memory_node.clone(),
            address,
        })
    }
    // [...]
}
```

Using this bespoke additional construct of tracking the maximum, we resolve the quantifier to a maximum. This does not generalize well, however, as it requires coming up with a bespoke way to circumvent the quantifier. This limitation prevents useful properties like a modified complement of the previous one, namely, for some $k \in \mathbb{N}$,

$$\forall i \ \mathbf{AG}(\text{memoryCounter}[i] \geq k \lor \text{memoryCounter}[i] = 0) \tag{3.9}$$

Allowing for memory not to be accessed is a reasonable assumption since most programs will not access the processor's entire address space.

3.2.3 Checked Properties

As with the properties about registers, there are two dual properties, Equation 3.10 and Equation 3.10, which can be checked using the changes introduced by this thesis. Here, Idx denotes the set of all array indices $\text{Idx} := \{i | i \in \mathbb{N}_0 \land i < 2^{64}\}$ and $k \in \mathbb{N}_0$ is arbitrary but fixed.

$$\forall i \in \text{Idx} \ \mathbf{AG} \left(\text{memoryCounter}[i] \leq k\right) \tag{3.10}$$

$$\forall i \in \text{Idx} \ \mathbf{AG} \left(\text{memoryCounter}[i] \geq k\right) \tag{3.11}$$

Of those properties, as with the properties for registers, the second property, Equation 3.11, is only useful if it is augmented similarly to Equation 3.5, as its unaugmented form is true for the initial state of the model.

Additionally, as outlined in Subsection 3.2.2, we rely on solvers without support for theories with quantifiers. Hence, we have to reformulate the properties we want to check without quantifiers.

Equation 3.10 can be expressed without quantifiers as

$$\mathbf{AG}\left(\max_{i\in\mathrm{Idx}}(\mathrm{memoryLocation}[i]) \leq k\right) \tag{3.12}$$

and its negation as

$$\mathbf{EF}\left(\max_{i\in\mathrm{Idx}}(\mathrm{memoryLocation}[i]) > k\right) \tag{3.13}$$

Lemma 1. *Equation 3.10 and Equation 3.12 are equivalent.*

Proof. Let Idx be the set of indices, and k be arbitray but fixed as introduced earlier. Observing that for an array A

$$\forall i \in \mathrm{Idx}\ \left(\min_{j\in\mathrm{Idx}}(A[j]) \leq A[i] \leq \max_{j\in\mathrm{Idx}}(A[j])\right) \tag{3.14}$$

we can rewrite Equation 3.10 as

$$\forall i \in \mathrm{Idx}\ \mathbf{AG}\left(\max_{j\in\mathrm{Idx}}(\mathrm{memoryCounter}[j]) \leq k\right) \tag{3.15}$$

Since the maximum does not depend on the index, the quantifier can be dropped, leaving

$$\mathbf{AG}\left(\max_{j\in\mathrm{Idx}}(\mathrm{memoryCounter}[j]) \leq k\right) \tag{3.16}$$

which is equivalent to Equation 3.12. □

The implementation of the property follows analogously to the implementation of the properties for registers in Subsection 3.1.3 since the maximum, as introduced in Subsection 3.2.2, is a scalar bit-vector state and hence like a register.

Running BtorMC on a model that satisfies the property, the model checker reports that the property is satisfied and provides a witness. However, it only reports that the property holds for the state variable *maximum*, not the address of the memory location, which corresponds to this number of accesses. To extract this information, we need to simulate the model with the inputs of the witness. The authors of [17] provide a set of tools (see Subsection 2.4.3) for dealing with BTOR2 models and witnesses, including a model simulator and witness checker, *btorsim*.

The witness checker simulates applying the transition functions with the inputs provided by the witness iteratively and verifies that the properties claimed by the witness are actually satisfied. Since it simulates the whole model, it has to store the values of the states for this execution, which it can print to standard output when run using the `--states` flag.

The output of the witness checker with the `--states` is still a valid BTOR2 witness as described in Subsection 2.4.2 but explicitly outputs all assignments to states that have been assigned to. For scalar bit-vector states, this follows the format

```
state-number value name@t
```

and for array states,

```
state-number [address]value name@t.
```

As discussed in Subsection 2.4.2, the bad state is satisfied in the frame after the assignment to the *maximum* state, and hence also for the corresponding memory location. Consequently, the address of the memory location that corresponds to the maximum is

1. part of the penultimate witness frame and
2. unique.

It is unique because only one memory location can be accessed per iteration. Hence, if there had been another memory location larger than k, the bad state would have been satisfied earlier.

To extract the address, we only have to find the penultimate frame, extract the value of the *maximum* state, find the memory location that contains this value, and extract the address. Listing 1 reproduces a simple bash script that extracts the address and prints it as a hexadecimal.

For the simple program *while.c* (Listing 2), the model checker finds a memory location that is read more than five times before being written again after about 120 steps. To obtain this result form an end-to-end walk-through, refer to the case study in Section 1 in the appendix.

4 Performance Analysis

Unicorn, which forms the basis of this thesis's implementation, relies on SMT solvers in two ways. First, during the unrolling phase, it directly queries a solver, Second, it outputs a BTOR2 model that has to be run by a model checker. We used BtorMC, which in turn uses SMT solver Boolector. Since Boolector is a well-established solver and out of the scope of this thesis, the main performance metric is the size of the model produced by unicorn.

In this chapter, we analyze the size of models produced by unicorn before our changes and how our changes influence the size of the models produced. Next, we analyze the size of test programs quantitatively.

4.1 Initial Performance

The model generated by unicorn is linear in the size of the program. To show this bound on the size of the model for a given program, we first decompose it into components that can be analyzed separately.

As described in Subsection 2.4.1, the model consists of

- sort, state, input and constant declarations,
- initialization and next statements,
- operations, and
- bad state declarations.

We will analyze those components on their own and then combine them into one final proof.

Lemma 2. *The number of sorts N_{sorts} stays constant, independent of the input program or bound.*

Proof. The sorts used by unicorn are

- 1-bit bit-vectors to model booleans,
- 8, 16, 24, 32, 40, 48, 56 and 64-bit bit-vectors to model 1 to 8-bit numbers, and
- arrays with 64-bit values and a 64-bit address space to model memory.

Hence, the number of sorts is $N_{sorts} = 10$, which is constant. □

Lemma 3. *The number of states N_{states} is linear in the size of the program P, independent of the bound. So are the number of initializations $N_{initializations}$ and next statements N_{nexts}.*

Proof. As explained in Section 2.1, the state of a RISC-V machine consists of 32 64-bit registers, a 64-bit program counter, and 2^{64} 64-bit memory locations. The theory of arrays encapsulates all memory locations in a single state. Since unicorn models the program counter as a one-hot encoded set of boolean states, the number of states introduced to represent the program counter is linear in the size of the program P. The total number of states is therefore $N_{states} = 33 + |P|$, which is linear in the size of the program. Since every state can only be initialized at most once and have at most one next statement, we get $N_{initializations} \leq N_{states}$ and $N_{nexts} \leq N_{states}$. Therefore, the number of initializations and the number of next statements are also linear in the size of the program. □

Lemma 4. *The number of inputs N_{inputs} stays constant, independent of the input program and bound.*

Proof. The current version of unicorn only allows one input in the input program. The input size is dynamically selected to be between 1 and 8 byte wide. Hence, the number of inputs is $N_{inputs} = 8$ (one for each width), which is a constant. □

Lemma 5. *The number of constants in the model $N_{constants}$ is linear in the size of the program P, independent of the bound.*

Proof. The method `ModelBuilder::generate_model` introduces a constant number of constants. When modelling the semantics of a program, modelling a single instruction introduces at most four new constants into the model. Hence, the number of constants in the model is bounded by $N_{constants} \leq 4 \cdot |P| + K$ with $K \in \mathbb{N}$, which is linear in the program size. □

Lemma 6. *The number of operations in the model $N_{operations}$ is linear in the size of the program P, independent of the bound.*

Proof. Each instruction is modelled by a corresponding `model_` method, which is non-recursive and does not use iteration. Hence, the number of nodes introduced to model each instruction can be bounded by a constant. Let $K_{max} \in \mathbb{N}$ denote the number of instructions introduced by the method that models the instruction which requires most nodes to model. We can then bound the number of operation nodes in the model by $N_{operations} \leq K_{max} \cdot |P|$, which is linear in the size of the program. □

Lemma 7. *The number of bad states in the model N_{bad} is linear in the size of the program P, independent of the bound.*

Proof. Unicorn produces one bad state for every property to check. Firstly, it produces a constant number of bad states to check for illegal memory accesses, division by zero and non-zero exit codes. Secondly, It produces a bad state for every occurrence of a jalr instruction that dispatches on the zero register. Hence, the number of bad states in the model can be bounded by $N_{bad} \leq K + |P|$ where $K \in \mathbb{N}$, which is linear in the size of the program. □

Theorem 1. *The size of the generated BTOR2 model for a given program P and bound k is linear in the size of the program.*

Proof. All nodes in the model have been accounted for at least once by Lemmata 2 to 7. By using the union bound, the cardinality of the union of the sets is less or equal to the sum of the cardinality of the individual sets. By the same argument, we approximate the total model size by

$$N_{nodes} = N_{sorts} + N_{states} + N_{inputs} + N_{constants} + N_{initializations} + N_{nexts} + N_{operations} + N_{bad}$$

Since all summands of the sum are linear in the size of the program, so is the sum. Therefore, the size of the program is linear in the size of the program. □

4.2 Cost of the Changes

This thesis modifies the model in two significant ways. First, introduce new states, one for each register and memory, to count the number of accesses. Secondly, by introducing new nodes to compute the next value of each counter. Finally, we will introduce new bad states that correspond to the properties we want to check.

The number of new states that get introduced to count the memory accesses is equal to the number of states due to registers and memory themselves. Hence, $N_{counters} = N_{states}$.

For the counter's transition functions, we mirror the transition function of the corresponding state. Additionally, we introduce a constant number of nodes to keep track of the maximum of all memory access counters when accessing memory, which is at most once per instruction. Since not every instruction accesses a register or memory, we can bound the number of nodes in the transition function by

$$N_{counter\_operations} \leq K \cdot N_{operations}$$

for some $K \in \mathbb{N}_0$.

The number of bad states we introduce is constant since we introduce a bad state for every register and for memory, which is constant, and hence $N_{counter,bad} = 32$, which is constant.

Theorem 2. *The size of the generated BTOR2 model after introducing counters for memory and registers for a given program P and bound k is linear in the size of the program.*

Proof. The total number of nodes after accounting for the changes introduced by this thesis are

$$N_{counter,nodes} = N_{nodes} + N_{counters} + N_{counter\_operations} + N_{counter,bad}$$

Since all summands of the sum are linear in the size of the program, so is the sum. Therefore, the size of the program is linear in the size of the program. $\square$

5 Discussion and Conclusion

In this thesis, we lay the groundwork for exploring patterns in memory and register usage of programs. We extend the symbolic execution engine unicorn to count register and memory read accesses since the last write access, thereby showing that the answer to our research question is yes. Furthermore, we implemented properties checking for usage patterns that either bound the number of read accesses since the last write access to a register or memory location from above or below.

We explained that the bounding from below does not produce useful insight with our approach to modelling counters, as the property is trivially true for any program. However, as we will explain shortly, by slightly altering our modelling approach, these properties can give meaningful insights.

As for the bound from above, the properties can guarantee that there are no outliers in terms of the number of read accesses since the last write access for a certain execution depth. While this might be useful in some cases, violations of the property indicate a register or memory location that has a higher number of read accesses since the last write access than the bound. If the inputs provided by the witness are within the problem domain, this could be a desirable outcome. Conversely, if the inputs are not in the problem domain, this program's allocation strategy could be improved.

With programs running in large data centers and power costs being a concern, program efficiency is an important business problem. Our implementation allows engineers and researchers to examine the results of heuristic algorithms for memory and register allocation for a fixed program. Since heuristic algorithms are either program-agnostic or designed with specific trade-offs in mind, the result might not be optimal. This thesis provides a tool to evaluate heuristics with regard to certain access patterns.

Returning to the bound from above, that is, to ensure that the number of read accesses since the last write access of a register is less than a certain bound, the current model fails to deliver useful results. Since the only reasonable initialization for counters is to set them to zero, the property is satisfied immediately. This is due to the fact that the property checks if the value of the read counter ever is below the bound, even when the counter is not being reset in this step.

We suggest the following change to modelling properties: Instead of modelling the semantics of

$$\mathbf{AG}(\text{registerReadCounter} \leq k) \qquad k \in \mathbb{N}_0 \tag{5.1}$$

we propose modelling the following enhanced property:

$$\mathbf{AG}(\text{registerWrite} \implies \text{registerReadCounter} \leq k) \qquad k \in \mathbb{N}_0 \tag{5.2}$$

assuming that *registerWrite* is true if and only if the register is being written in this step.

This modified property only inspects the counter when it has to be reset. Modelling this property requires a significant change for the implementation of the register access counters, which is out of scope for this work. We therefore propose this as possible future work.

Furthermore, we only implement checking for simple properties in this work since our main goal is to extend unicorn to count register accesses. We recommend the following property for further work: Instead of checking for a violation of the property, allow each register a budget $B \in \mathbb{N}_0$ and check that it does not violate the property more than B times. This could allow to come closer to checking a notion of "average register usage" in a finite context.

Moreover, this thesis did not explore advanced techniques in bounded model checking such as k-Induction, Craig interpolation, or completeness bounds, which could allow to extend checking the properties for a finite number of steps to the checking properties for any number of steps, essentially transcending the constraints of bounded model checking.

Bibliography

[1] The Selfie Project Authors. *RISC-U*. 2023. URL: `https://github.com/cksystemsteaching/selfie/blob/cbbac18e15ade5dc32bd0af1204143e02fb71253/riscu.md` (visited on 03/19/2024).

[2] Clark Barrett et al. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability*. Ed. by Hans van Maaren Armin Biere Marijn Heule and Toby Walsch. IOS Press, 2008.

[3] Armin Biere and Daniel Kröning. "SAT-Based Model Checking". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 277–303. ISBN: 978-3-319-10575-8. DOI: `10.1007/978-3-319-10575-8_10`.

[4] Robert Brummayer, Armin Biere, and Florian Lonsing. "BTOR: bit-precise modelling of word-level problems for model checking". In: *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*. SMT '08/BPR '08: 2008 Joint Workshops on the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning. Princeton New Jersey USA: ACM, July 7, 2008, pp. 33–38. ISBN: 978-1-60558-440-9. DOI: `10.1145/1512464.1512472`.

[5] Edmund M. Clarke et al. *Model Checking*. second edition. The Cyber-Physical Systems Series. Cambridge, Massachusetts: The MIT Press, Dec. 4, 2018. 424 pp. ISBN: 978-0-262-03883-6.

[6] Leonardo De Moura and Nikolaj Bjørner. "Satisfiability modulo theories: introduction and applications". In: *Commun. ACM* 54.9 (Sept. 2011), pp. 69–77. ISSN: 0001-0782. DOI: `10.1145/1995376.1995394`.

[7] Leonardo De Moura and Nikolaj Bjørner. "Z3: an efficient SMT solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340. ISBN: 3540787992.

[8] E. W. Dijkstra. “Solution of a problem in concurrent programming control”. In: *Commun. ACM* 8.9 (Sept. 1965), p. 569. ISSN: 0001-0782. DOI: `10.1145/365559.365617`.

[9] Valentin Goranko and Antje Rumberg. “Temporal Logic”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta and Uri Nodelman. Fall 2023. Metaphysics Research Lab, Stanford University, 2023.

[10] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN: 0128119055.

[11] *History of RISC-V*. 2023. URL: `https://riscv.org/about/history/` (visited on 03/23/2024).

[12] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2nd. Prentice Hall Professional Technical Reference, 1988. ISBN: 0131103709.

[13] Christoph M. Kirsch. “Selfie and the basics”. In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2017. Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 198–213. ISBN: 9781450355308. DOI: `10.1145/3133850.3133857`.

[14] Moritz Lipp et al. “Meltdown: reading kernel memory from user space”. In: *Communications of the ACM* 63.6 (May 21, 2020), pp. 46–56. DOI: `10.1145/3357033`. (Visited on 03/28/2024).

[15] Nicholas D Matsakis and Felix S Klock II. “The rust language”. In: *ACM SIGAda Ada Letters*. Vol. 34. 3. ACM. 2014, pp. 103–104.

[16] Aina Niemetz, Mathias Preiner, and Armin Biere. “Boolector 2.0”. In: *J. Satisf. Boolean Model. Comput.* 9.1 (2014), pp. 53–58. DOI: `10.3233/sat190101`.

[17] Aina Niemetz et al. “Btor2 , BtorMC and Boolector 3.0”. In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10981. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 587–595. ISBN: 978-3-319-96144-6 978-3-319-96145-3. DOI: `10.1007/978-3-319-96145-3_32`.

[18] Amir Pnueli. “The temporal logic of programs”. In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. SFCS ’77. USA: IEEE Computer Society, 1977, pp. 46–57. DOI: `10.1109/SFCS.1977.32`.

[19] Andrew Waterman. “Improving Energy Efficiency and Reducing Code Size with RISC-V Compressed”. Master’s Thesis. Berkley: University of California, 2011. (Visited on 03/23/2024).

[20] Editors Andrew Waterman and Krste Asanovi´c. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. RICS-V Foundation, 2019.

Appendix A: Code

Listing 1: A bash script `extract_address_from_witness.sh` that extracts the address of the memory location that triggered the bad state from a BTOR2 model and witness.

```
#!/bin/bash

# Precondition: The environment variable
# WITNESS_WITH_STATES contains the path to a file
# containing the output of
# ‘btorsim $BTOR2_FILE --states > $WITNESS_WITH_STATES‘

COUNT=$(cat "$WITNESS_WITH_STATES"\
  | grep memory-counter-max\
  | tail -n 1\
  | cut -d " " -f 2)

printf "The memory location that triggered the bad state is "
grep "$COUNT" "$WITNESS_WITH_STATES"\
  | head -n 1\
  | sed "s/.*\[\([01]\+\)\].*/\1/"\
  | python -c "import sys; print(hex(int(sys.stdin.read(), 2)))"
```

1 Case Study: `while.c`

The file `while.c` is part of the test suite of unicorn in the folder `examples`. To generate the BTOR2 model, first compile it to a RISC-V binary using *selfie* by running

```
$ selfie -c while.c -o while.m
```

Then, to generate the BTOR2 model, run

```
$ cargo run beator while.m -o while.btor2
```

To run the BtorMC model checker on the model, run

```
$ btormc while.btor2 -kmax 120
```

The argument `-kmax 120` instructs to search for witnesses of up to 120 steps. The default is 20, which is not enough to trigger the condition for this model. The model checker will report that the property `b1`, which is our "more than five reads before a write" property.

The above command prints the witness to the standard output. To redirect the witness to a file, run

```
$ btormc while.btor2 -kmax 120 > while.witness
```

To extract the memory location, run

```
$ btorsim while.c.witness --states while.c.with-states
$ WITNESS_WITH_STATES=while.c.with-states \
> ./extract_address_from_witness.sh
```

The resulting output is

```
The memory location that triggered the bad state is 0xfffa0
```

which aligns with the fact that the memory location of `x` is on the heap.

Listing 2: Test program `while.c` from the unicorn test suite.

```
uint64_t main() {
  uint64_t  i;
  uint64_t* x;

  i = 0;
  x = malloc(8);

  *x = 0;

  read(0, x, 1);

  while (*x) {
    if (i >= 10)
      return 1;
    else
      i = i + 1;
  }

  return 0;
}
```