SEPO: Optimizing RISC-V using Symbolic Execution

Sora Kanosue*
University of California, Berkeley
Berkeley, CA, USA
sorakanosue@berkeley.edu

Jacob Yim*
University of California, Berkeley
Berkeley, CA, USA
jacobyim@berkeley.edu

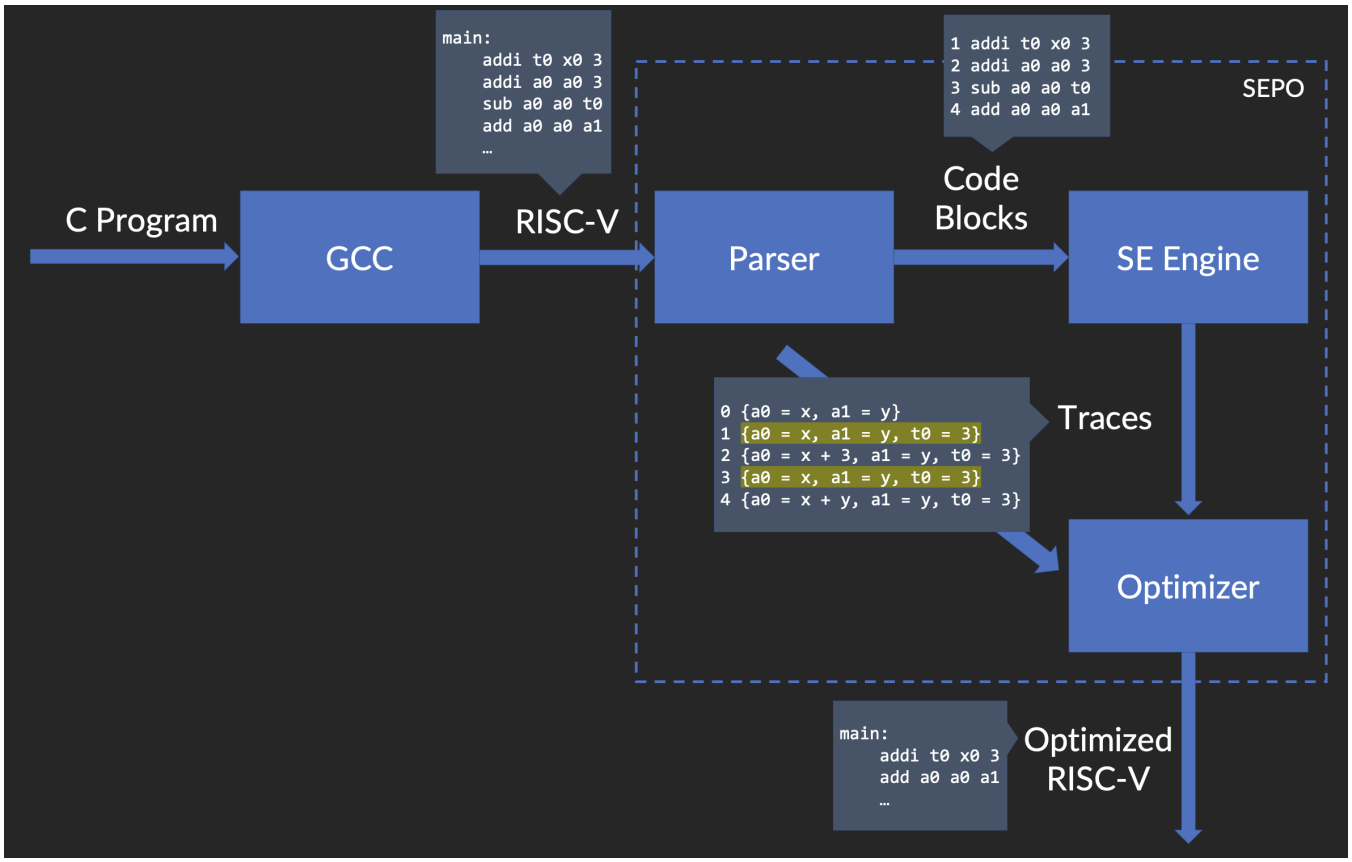

Figure 1. Diagram of the SEPO system, with a toy example

1 Introduction

Since the invention of compilers, computer scientists have continually searched for ways in which their outputs can be improved and made more performant. This problem has been solved in a multitude of ways over the decades, through techniques such as constant propagation, function inlining, and peephole optimization. Each optimization seeks to leverage the semantics of the source and target languages of the compilers to produce smaller assembly executables which are able to run faster.

Previous approaches to compiler optimization which focus on improving the assembly outputs themselves have traditionally focused on peephole optimizations, in which rewrite rules known to be correct were applied to the compiler outputs to produce shorter sequences of instructions. More recent work in the area has focused on superoptimization, which instead focuses on finding the shortest sequence of instructions which matches the semantics of a target program.

In this paper, we present SEPO, a system for finding optimizations in assembly code by processing the states that a RISC-V CPU would pass through, rather than operating directly on the assembly instructions themselves as previous work does. At a high level, our system takes in a RISC-V program, and outputs an optimized version of that program while maintaining its semantics. By symbolically executing the program in order to find optimizations to perform, we are

*Both authors contributed equally to this research.

able to guarantee correctness by only performing changes where the final symbolic state of the improved program is equivalent to that of the original.

Our work takes an approach to assembly optimization which, to the best of our knowledge, is previously unexplored, and provides a promising, fertile ground for future work. In Section 2, we summarize previous work in peephole optimization and superoptimization, and how our approach fundamentally differs. Section 3 provides a high-level overview of our system, and a description of each part. Section 4 provides an in-depth exploration of the implementation of our symbolic execution engine, while Section 5 details the optimizations we have implemented to this point. In Section 6, we detail the (many) ideas we have for future work related to this project.

2 Related Work

2.1 Traditional Compiler Optimization

Chakraborty provides an overview of research on peephole optimization starting from its invention by William McKeeman in 1965 [3]. Prior approaches involve pattern matching (the traditional method), replacement of adjacent sets of instructions with fixed length, and replacement of instructions modifying the same data unit.

2.2 Symbolic Execution for Compiler Optimization

To our knowledge, there is little existing literature on symbolic execution for compiler optimization. Kloibhofer et al.'s work on SymJEx stands out as one recent effort to automatically infer compiler optimizations using symbolic execution [4]. However, this work focuses largely on the implementation of a symbolic execution engine for the Java Virtual Machine GraalVM, treating compiler optimization as a potential use case. The approach used to accomplish these optimizations differs significantly from our proposed method: the GraalVM optimizer still performs a pre-defined set of optimizations on the Graal IR, using symbolic execution to generate constraints and SMT solving to identify optimizations that satisfy the constraints. Our method, by contrast, performs more general replacement rules based on stored program state. Furthermore, Kloibhofer et al. target the Java Virtual Machine GraalVM's IR, while we hope to optimize RISC-V assembly. During their evaluation, Kloibhofer et al. find that some benchmarks achieve performance improvements of up to 15%, but many benchmarks show no significant improvement. We hope that by targeting a more simple language with potentially fewer existing compiler optimizations and by leveraging a different strategy, our optimizer will be able to achieve larger improvements.

2.3 Superoptimization

Much ongoing work in the compiler optimization domain instead focuses on superoptimization, the problem of finding the optimal sequence of instructions matching an input and output state [8] [1]. This is largely a search problem, which produces highly optimal results using very involved computations. Traditional approaches to building superoptimizers functioned by performing enumerative searches over increasingly larger programs, yielding superoptimizations for programs of length up to 13 instructions [6]. More recent work in the space has explored the idea of using stochastic search methods such as Monte Carlo Markov Chain sampling to increase the scale of superoptimization albeit at the cost of sacrificing completeness [8]. Both these approaches operate on programs, or sequences of instructions directly. Our approach is orthgonal to this, in that we generate a trace of states for a given input sequence of instructions, and use those states to output an equivalent, more performant sequence of instructions.

3 SEPO

Our system, SEPO, consists of a parser, a symbolic execution engine, and an optimizer. SEPO processes arbitrary RISC-V assembly and outputs optimized RISC-V. We implemented SEPO in Java. SEPO is freely available online at https://github.com/skberkeley/sepo.

The parser is responsible for identifying sections of straight-line code and parsing RISC-V into an intermediate representation (IR). The parser processes unoptimized RISC-V code, identifying all branching or jumping instructions. The instructions in between these branches or jumps are extracted to produce a list of straight-line "segments." Each segment is a list of instructions encoded in our IR, which represents each RISC-V instruction as a Java object.

The symbolic execution engine takes in a list of segments in our IR. For each segment, it produces a trace, a list of symbolic CPU states between every instruction. Each state encodes the values in registers as well as memory. Details about the implementation of the symbolic execution engine are discussed in greater detail in section 4.

Finally, the optimizer takes in a list of instructions in our IR and a list of CPU states to produce a list of instructions that is optimized based on the states. Optimization is applied to each segment individually to produce optimized segments, which are then "stitched" back into the original program with

branching and jumping instructions. In our implementation, we included one optimization, dead code elimination. Details about this implementation are discussed in detail in section 5.

4 Symbolic Execution Engine

We implemented a symbolic execution engine capable of executing straight-line blocks of RISC-V (RV64I) code, generated by our parser. Since we chose to support only straight-line pieces of code for this initial implementation, our engine does not support branch or jump instructions.

In order to model the semantics of the RISC-V ISA, we maintain two maps as state. One map represents the register file in a RISC-V CPU, while the other models the memory of the CPU. The symbolic nature of our engine comes from the expressions which are stored as values in registers and in memory, as well as the expressions used as memory addresses. The symbols present in these symbolic expressions are generated when an uninitialized value is referenced. In a fully fledged RISC-V CPU, these values would have been initialized in an preceding block of straight-line code.

We designed SEPO so that the symbolic execution engine outputs a trace of states that it passes through as the instructions are executed. In order to support this, we make a copy of the register file and memory maps before executing each instruction, and after the very last instruction being executed. Once execution of a block of straight line code completes, these states are outputted.

4.1 Value Representation

In order to accurately model RISC-V while accessing the power of SMT solvers, we chose to represent values in SEPO in a manner analogous to the theory of bitvectors. Supporting bitvectors required some engineering of our type system to capture all the semantics of RISC-V instructions. The most significant of these were `Slices` and `Concatenations`, which we used extensively to accurately process memory and 64-bit specific instructions. For example, stores often involved slicing expressions into their byte components, while loads entailed concatenating byte-sized values together.

4.2 Memory Model

We model the memory of the RISC-V system being executed by mapping symbolic expressions denoting memory addresses to other symbolic expressions representing the values being stored at those memory addresses. To support RISC-V's semantics of loading and storing in denominations of bytes, half-words, words, and double words, each byte of memory is stored as a separate entry in the map. Then, each load and store on an operand larger than a byte is executed via the appropriate number of loads and stores into the map. For example, a store word instruction to address M is executed by splitting the value being stored into 4 byte-sized values and executing stores into the memory map at addresses M, $M+1$, $M+2$, and $M+3$. For the sake of the system's simplicity, we do not support non-byte aligned memory accesses. Since our load and store algorithms entail checking the satisfiability or equivalence of certain expressions, we use the Princess SMT solver [9] [7] to handle this.

The algorithms in Figures 2 and 3 describe how loads and stores into the memory map are executed. In these algorithms, `M` denotes the memory map, `M[e]` the mapping of `e` in `M`, `M[e] := v` updates or adds to `M` so that `e` is mapped to `v`, and `M -= e` removes `e`'s mapping from `M`.

```
store(e, v):
  for e' in M:
    if is_sat(e = e'):
      M -= e'
  M[e] = v
```

Figure 2. Store algorithm for symbolic execution engine's memory model

When storing a value at a symbolic address e, then every value in memory whose address may be equal to e may be overwritten by the store being executed. To model this, we remove every entry (e', v) from the memory map where the expression $e = e'$ is satisfiable.

```
load(e):
  if e is concrete:
    if !e in M:
      M[e] := new_symbol()
    return M[e]
  else:
    for e' in M:
      if is_equiv(e, e'):
        return M[e']
    M[e] := new_symbol()
    return M[e]
```

Figure 3. Load algorithm for symbolic execution engine's memory model

When loading a value from a symbolic address e, then two situations present themselves. The first is that e can be simplified to a concrete value v, in which case we check whether

the memory map contains a mapping for v, returning it if it does. If the memory map does not contain a mapping for v, then we instantiate a new symbol, and map v to it. Then, this new symbol is returned. This case models load operations from memory locations which were instantiated by a code block previous to the one currently being executed by the symbolic execution engine. If e is symbolic, we first check if the memory map contains any mappings for symbolic expressions equivalent to e. If no equivalent expressions exist, then e could be concretized to a memory address not initialized within the code block currently being executed, so a new symbol is instantiated and e mapped to it.

5 Optimizer

5.1 State Equivalence

We use the algorithm presented in Figure 4 to determine whether two states in a trace outputted by our symbolic execution engine are equivalent. Since parts of it require checking whether two symbolic expressions are equivalent, we enlist the help of an SMT solver [9] [7].

```
states_equiv(s, s'):
  for reg in s.regs:
    if !is_equiv(s.regs[reg], s'.regs[reg]):
      return false
    if s.mem.size != s'.mem.size:
      return false
    for addr in s.mem:
      for addr' in s'.mem:
        if is_equiv(addr, addr'):
        if !is_equiv(s.mem[addr], s'.mem[addr']):
            return false
          else:
            break
      return false
  return true
```

Figure 4. Algorithm to check whether two states are equivalent

5.2 Dead Code Elimination

We define dead code to be code which can be removed without affecting the functionality of the program from which it is removed. In Figure 5, we present the algorithm used by our optimizer to identify and eliminate sections of dead RISC-V code. By iterating over the traces from the end in the inner for loop, we guarantee that the dead code being eliminated is as large a sequence as possible.

```
dead_code_elimination(trace, instrs):
  for i in 0..trace.size:
    for j in (trace.size - 1)..(i + 1):
      if states_equiv(trace[i], trace[j]):
        return instrs[:i] + instrs[j:]
  return instrs
```

Figure 5. Algorithm for dead code elimination

We were successfully able to run our system with dead code elimination on a toy example where we injected redundant instructions into an assembly file outputted by GCC. We were unable able to run the system on larger examples due to compatibility issues with Z3 and versioning issues with the prebuilt GCC tool-chain we used.

6 Future Work

6.1 Additional Optimizations

In this work, we only implemented one SEPO optimization, dead code elimination. In practice, however, dead code may not be common in compiled code, especially if traditional compiler optimizations have already been applied. Thus, extending the optimizer beyond just dead code elimination is a priority for future work.

The natural successor to dead code elimination, which can be thought of as finding different states which have as the shortest path between them the empty sequence of instructions, is to find non-consecutive states which are one instruction apart. This would firstly involve finding states which are separated by more than one instruction in the original code block, but have the property that an instruction can be constructed to transform the earlier appearing state to one that is equivalent to the later one. We would then have to actually construct this instruction to arrive at a shorter, equivalent sequence of instructions. Identifying states which are exactly an instruction apart may not be as difficult a task as it immediately sounds, given that RISC-V instructions only modify one value of state at a time.

The extension of this idea, then, would be to identify states which are two instructions apart. This poses a more challenging problem than the above, since it would necessarily involve generating an intermediate state which we do not already have in hand. Once this intermediate state has been concretized, then the connecting instructions can be generated assuming the above problem has been solved. Deducing candidate intermediate states can potentially be aided by the fact that most of their values should be unchanged from the states that we identify to be close neighbors.

6.2 Jumps and Branching Instructions

SEPO currently only performs optimizations on straight-line code, as the parser separates instructions in between jumps or branches into "segments" to be passed to the symbolic execution engine. Likewise, branching and jumping instructions are not encoded in the symbolic execution engine, which only tracks a sequence of states along a single branch. This means that states across branches cannot be compared, and some potential optimizations are likely missed.

In order to allow the symbolic execution engine to process branches, it may be possible to attach path conditions to values in the states and memory. However, the path explosion problem is a major limitation of this approach: the number of paths in a program grows exponentially as branches are created. Some existing works in this area attempt to address the path explosion problem using strategies like state merging [5]. Another challenge is processing loops, especially when the number of iterations may be dependent on a symbolic expression. More work must be done to determine whether handling branches and jumps is feasible for our approach.

6.3 Improving Memory Representation

Currently, when storing values in memory at an address e, the symbolic execution engine flushes entries at all memory map addresses e' that could be equivalent to e. This means that storing values at symbolic addresses potentially leads to many values being wiped from the memory map immediately or after subsequent stores. Thus, a significant amount of information is lost during memory stores, which could be used to discover additional optimizations. One potential solution is to add conditions to addresses in memory, such that our memory model represents all possible destinations for stores at symbolic addresses. While this approach is subject to a similar path explosion problem to branching instructions, prior works like MemSight [2] address these issues using state merging, among other optimizations.

6.4 Constant Propagation

One of the obstacles we faced during implementation was that our symbolic expressions grew to such a size that the SMT solver we were using, Princess, was unable to handle them. Although using a more mature solver such as Z3 [10] might have alleviated this issue, we believe that implementing a version of constant propagation within our symbolic execution engine would greatly improve its performance. One example of how constant propagation might operate could be to preemptively add two values if an add instruction is being processed by the engine, rather than symbolically encoding the addition.

6.5 Support for Assembler Directives

An idea closely tied to that of constant propagation is that of adding support for assembler directives to our system. For example, the GCC tool-chain we used to compile C programs to RISC-V code inserts static strings directly into the RISC-V files and references them using assembler directives. By adding support for assembler directives and using constant propagation, we can reduce the amount of symbolic expressions present in the states outputted by our symbolic execution engine.

6.6 Control Flow Analysis to Enhance States

We believe that control flow analysis (CFA) of programs could enhance the information available in each state, and greatly improve the quality and quantity of the optimizations inferred from them. For example CFA could be used to deduce whether any memory locations or registers contain concrete values in them at the beginning of a given straight line block of code. These concrete values would then be propagated throughout the states as that block of code is executed, reducing the number of symbolic values. Another, potentially more powerful way in which CFA could be useful is in deducing which values are live at the end of a straight line block of code. Say, for example, that CFA reveals that `x5` is referenced at some point after a block of code is executed, but `x7` is not. Then any optimizations our optimizer performs should preserve the final value of `x5`, but no longer needs to care what the final value of `x7` is.

6.7 Extension to Other Assembly Languages

Although we chose to implement our system targeted at the RISC-V RV64I instruction set due to the simplicity of the ISA's semantics, nothing prevents us from implementing equivalent systems for other assembly languages, provided we are able to model their semantics. Extending SEPO to other instruction sets, like x86, LLVM, or other RISC-V extensions, would broaden its applicability.

6.8 Superoptimization

We believe that encoding assembly programs in terms of states rather than the instructions themselves holds potential as another avenue for approaching the problem of superoptimization. For example, for a given straight-line block of code, the functionality of the block can be encoded in terms of the start and end states produced by running the block through

our symbolic execution engine. Then, the problem of superoptimization can be reformulated as finding the shortest sequence of instructions such that symbolically executing them produces the same start and end states. It can also be re-framed as a shortest path problem from graph theory, in which the nodes are the states produced by symbolic execution and the directed edges between them the instructions which transform one state to another.

7 Conclusion

In this work, we present SEPO, a system for optimization of RISC-V assembly based on symbolic states. We contribute 1) a framework for compiler optimizations based on traces of program state generated using symbolic execution, 2) an implementation of the symbolic execution engine used to generate these traces, and 3) an implementation of one symbolic state based optimization. We believe that symbolic execution and symbolic state based approaches hold unexplored potential for future work in program optimization, and hope that this work enables further research in this area.

References

[1] Sorav Bansal and Alex Aiken. "Automatic generation of peephole superoptimizers". In: *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. ASPLOS XII. New York, NY, USA: Association for Computing Machinery, Oct. 20, 2006, pp. 394–403. ISBN: 978-1-59593-451-2. DOI: 10.1145/1168857.1168906. URL: https://dl.acm.org/doi/10.1145/1168857.1168906 (visited on 10/04/2023).

[2] Luca Borzacchiello et al. "Memory models in symbolic execution: key ideas and new thoughts". In: *Software Testing, Verification and Reliability* 29.8 (2019). e1722 stvr.1722, e1722. DOI: https://doi.org/10.1002/stvr.1722. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1722. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1722.

[3] Pinaki Chakraborty. "Fifty years of peephole optimization". In: *Current Science* 108.12 (2015). Publisher: Current Science Association, pp. 2186–2190. ISSN: 0011-3891. URL: https://www.jstor.org/stable/24905654 (visited on 12/11/2023).

[4] Sebastian Kloibhofer et al. "SymJEx: symbolic execution on the GraalVM". In: *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes*. MPLR '20. New York, NY, USA: Association for Computing Machinery, Nov. 4, 2020, pp. 63–72. ISBN: 978-1-4503-8853-5. DOI: 10.1145/3426182.3426187. URL: https://dl.acm.org/doi/10.1145/3426182.3426187 (visited on 10/04/2023).

[5] Volodymyr Kuznetsov et al. "Efficient state merging in symbolic execution". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (2012). URL: https://api.semanticscholar.org/CorpusID:135107.

[6] Henry Massalin. "Superoptimizer: a look at the smallest program". In: *ACM SIGARCH Computer Architecture News* 15.5 (Oct. 1, 1987), pp. 122–126. ISSN: 0163-5964. DOI: 10.1145/36177.36194. URL: https://dl.acm.org/doi/10.1145/36177.36194 (visited on 12/12/2023).

[7] Philipp Rümmer. "A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic". In: *Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. Vol. 5330. LNCS. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 274–289. ISBN: 978-3-540-89438-4. DOI: 10.1007/978-3-540-89439-1_20. URL: http://link.springer.com/10.1007/978-3-540-89439-1_20 (visited on 12/11/2023).

[8] Eric Schkufza, Rahul Sharma, and Alex Aiken. "Stochastic superoptimization". In: *ACM SIGPLAN Notices* 48.4 (Mar. 16, 2013), pp. 305–316. ISSN: 0362-1340. DOI: 10.1145/2499368.2451150. URL: https://dl.acm.org/doi/10.1145/2499368.2451150 (visited on 10/05/2023).

[9] *sosy-lab/java-smt*. original-date: 2015-11-18T11:36:07Z. Dec. 11, 2023. URL: https://github.com/sosy-lab/java-smt (visited on 12/11/2023).

[10] *Z3Prover/z3*. original-date: 2015-03-26T18:16:07Z. Dec. 12, 2023. URL: https://github.com/Z3Prover/z3 (visited on 12/12/2023).