Low-Level Code Synthesis Using Constraint Programming

Beatriz Aedo Diaz
Complutense University of Madrid
Madrid, Spain
beaedo@ucm.es

Elvira Albert
Complutense University of Madrid
Madrid, Spain
elvira@pip.ucm.es

Maria García de la Banda
Monash University
Melbourne, Australia
maria.garcialabanda@monash.edu

Alejandro Hernández-Cerezo
Complutense University of Madrid
Madrid, Spain
aleher06@ucm.es

Alexey Ignatiev
Monash University
Melbourne, Australia
alexey.ignatiev@monash.edu

Claudia López-Mingo Moreno
Complutense University of Madrid
Madrid, Spain
claudla3@ucm.es

Albert Rubio
Complutense University of Madrid
Madrid, Spain
alberu04@ucm.es

Peter J. Stuckey
Monash University
Melbourne, Australia
peter.stuckey@monash.edu

ABSTRACT
Superoptimization is a technique that aims to find the optimal sequence for a given one by exploring the space of equivalent sequences. While this technique is very effective in uncovering challenging optimizations, it is computationally demanding to implement in practice. In this study, we explore scalable techniques proposed in previous state-of-the-art tools for superoptimizing stack-bytecode, implementing them within the MiniZinc framework. These techniques can handle different objective functions for optimization and can be tailored to different types of stack-bytecode. We adapt our model to two modern stack-based bytecode languages: the Ethereum Virtual Machine (EVM) and WebAssembly (Wasm). For these languages, we examine different objective functions that are relevant in their respective contexts. Additionally, we propose several enhancements in terms of domain-specific constraints to improve the scalability of the approach and evaluate their impact on the initial proposal. Our experimental findings demonstrate that our model effectively handles significant blocks of code and identifies impactful optimizations, even in programs that have already undergone optimization.

CCS CONCEPTS
• Software and its engineering → Automatic programming: Compilers.

KEYWORDS
Superoptimization, Program Synthesis, MiniZinc, EVM, WebAssembly

1 INTRODUCTION
Superoptimization is a technique that aims to find the optimal translation of a given sequence of loop-free instructions by exhaustively searching the space of valid solutions. The generated code satisfies two critical properties: correctness, ensuring semantic equivalence is preserved; and optimality, achieved by selecting the best solution based on the specified objective function. For many years, this technique was considered prohibitively expensive due to its huge computational demands. However, there has been a recent revival of these techniques driven by the development of new tools within the context of Constraint Programming (CP) that are well-suited to handle them. Constraint programming is a software technique that finds its roots in tackling intricate problems using sequences of variables with explicit domains and a set of finite constraints. These problems, referred to as Constraint Satisfaction Problems (CSPs), aim to find a solution that fulfills all imposed constraints. The solution to the problem will be the one that satisfies all the constraints. Moreover, there are extensions to these problems in which we aim to find the optimal solution according to an objective function and prove it is indeed optimal. Constraint Programming has diverse applications across different domains, from Artificial Intelligence to Molecular Biology.

Within the realm of Constraint Programming, Minizinc [6] emerges as a cornerstone tool, offering a declarative language tailored to model Constraint Programming problems. Minizinc allows one to write code in a very similar way as the mathematical formulation of it would be. The use of Minizinc not only streamlines the modeling process but also facilitates efficient exploration of problem spaces, enabling the rapid prototyping and validation of solution strategies.

Stack-based languages present a very interesting choice for studying these superoptimization techniques, given their usually limited instruction set and straightforward stack management mechanisms. But at the same time, managing the stack efficiently is far from trivial. In our approach, we focus on two different modern stack-based languages: the Ethereum Virtual Machine (EVM) [1] and WebAssembly (Wasm) [2]. These languages employ distinct approaches to stack management.

The Ethereum Virtual Machine serves as the execution environment for the Ethereum blockchain to process the transactions in the network. It employs a set of stack operations to manage the stack: SWAPx, which swaps the topmost element with the element in position x+1, DUPx, that duplicates the element at position x and POP, that removes the topmost element. Additionally, it introduces the concept of gas, a metric quantifying the computational effort required to execute a transaction. Transactions are paid according to the amount of gas spent, thus making optimizations highly valuable for both users and developers. The gas consumption is determined
in terms of the opcodes executed following the cost model in the Yellow Paper [7]. Another relevant optimization criteria in the EVM is the size in bytes of the program (referred to as smart contracts), as deployment costs are based on this metric.

WebAssembly (or Wasm), on the other hand, is an instruction set architecture designed for fast and safe execution of applications on the Web. It specifies a set of instructions, its semantics and the textual representation of programs, with no assumptions about the concrete execution environment. Wasm uses registers to manage the stack efficiently, storing elements and retrieving them when needed. Optimization in Wasm is also crucial, as optimized code has a notable impact on the overall performance.

2 MODELLING OF THE PROBLEM

To model the problem, we have used the symbolic execution engine within SuperStack [3], a superoptimization tool that targets various types of stack-based bytecode. This phase of symbolic execution analyzes the initial bytecode sequence to determine the minimum number of elements required for its execution. Employing symbolic variables to represent these elements, it generates a specification of the block execution in terms of the operations performed, which is then passed to our MiniZinc model. This representation encompasses:

1. The initial and final states of the stack during the symbolic execution.
2. The set of operations applied in the sequence.
3. Upper bounds on the maximum number of operations and stack elements considered in a solution.
4. The initial and final states of the registers used and an upper bound on the maximum number of registers allowed in case of Wasm.

In our MiniZinc model, we use these upper bounds to encode a well-defined problem, introducing variables and constraints to express the stack (and register) state after assigning each operation. Additionally, we encode the overall state before and after assigning all operations, ensuring only equivalent solutions are considered. Our model incorporates constraints that represent the application of each instruction in the set at every possible step within the sequence, thereby exploring the space of valid solutions. Consequently, the optimal solution can be reconstructed from the assigned instructions at each step.

Another key aspect in our model pertains to the objective functions we are considering. As previously indicated, we have studied two criteria for EVM (gas and size-in-bytes) and one criterion for Wasm (number of instructions). Our model assigns corresponding costs to instructions so that upon assignment, they add to the final value we aim to minimize.

3 EXTENSIONS

SuperStack already tackles both EVM and Wasm superoptimization, employing an encoding based in propositional logic. This approach proves highly efficient but presents challenges in terms of extensions and debugging, due to the huge number of variables and constraints involved. Our MiniZinc model aims to replicate a similar encoding to the one in SuperStack, streamlining the prototyping of future extensions. In this work, we focus on modelling and studying the impact of different domain-specific constraints that aim to speed up the superoptimization process, some of which were already included in SuperStack. These constraints avoid exploring areas of the search space that do not lead to better solutions.

Moreover, we propose an alternative encoding that allows for elements in the final stack to appear in an arbitrary order rather than a fixed one. This order is specified by imposing a distance restriction among the final stack elements. While this extension does not adhere strictly to superoptimization (since equivalence is not preserved), it is interesting to study how the arrangement of these elements influences the overall optimization.

4 EXPERIMENTAL EVALUATION

We have assessed our approach using benchmark sets for both EVM and Wasm, which correspond to a subset of the benchmarks used in the SuperStack paper [3]:

1. A collection of 10 programs written in Circom from the Circom library [5], a DSL to create arithmetic circuits for zero-knowledge proofs.
2. A compilation of 30 optimized smart contracts also employed in the evaluation of the GASOL superoptimization tool [4], a precursor to SuperStack.

Our experimental results demonstrate that our MiniZinc model finds equivalent solutions for challenging sequences within a reasonable time frame. These sequences had been previously optimized by their respective compilers, which further proves the impact of the superoptimization techniques.

5 CONCLUSIONS

In summary, our study embarks on a comprehensive exploration of optimization strategies for both EVM and Wasm operations, showcasing the viability of the approach and measuring its impact over relevant set of benchmarks.

REFERENCES


