SuperStack: Superoptimization of Stack-Bytecode via Greedy, Constraint-Based, and SAT Techniques

ELVIRA ALBERT, Complutense University of Madrid, Spain
MARIA GARCIA DE LA BANDA, Monash University, Australia
ALEJANDRO HERNÁNDEZ-CEREZO, Complutense University of Madrid, Spain
ALEXEY IGNATIEV, Monash University, Australia
ALBERT RUBIO, Complutense University of Madrid, Spain
PETER J. STUCKEY, Monash University, Australia

Given a loop-free sequence of instructions, superoptimization techniques use a constraint solver to search for an equivalent sequence that is *optimal* for a desired objective. The complexity of the search grows exponentially with the *length of the solution* being constructed and the problem becomes intractable for large sequences of instructions. This paper presents a new approach to superoptimizing stack-bytecode via three novel components: (1) a greedy algorithm to refine the bound on the length of the optimal solution; (2) a new representation of the optimization problem as a set of weighted soft clauses in MaxSAT; (3) a series of domain-specific dominance and redundant constraints to reduce the search space for optimal solutions. We have developed a tool, named SuperStack, which can be used to find optimal code translations of modern stack-based bytecode, namely WebAssembly or Ethereum bytecode. Experimental evaluation on more than 500,000 sequences shows the proposed greedy, constraint-based and SAT combination is able to greatly increase optimization gains achieved by existing superoptimizers and reduce to at least a fourth the optimization time.

CCS Concepts: • **Software and its engineering → Automatic programming**; *Compilers*.

Additional Key Words and Phrases: Superoptimization, Program Synthesis, SAT, EVM, WebAssembly

ACM Reference Format:
Elvira Albert, Maria Garcia de la Banda, Alejandro Hernández-Cerezo, Alexey Ignatiev, Albert Rubio, and Peter J. Stuckey. 2024. SuperStack: Superoptimization of Stack-Bytecode via Greedy, Constraint-Based, and SAT Techniques. *Proc. ACM Program. Lang.* 8, PLDI, Article 205 (June 2024), 26 pages. https://doi.org/10.1145/3656435

1 INTRODUCTION

Program optimization is one of the pillars of software system development and a key component of green software. Superoptimization [42] is a powerful but expensive form of program optimization that can achieve results unattainable by pre-cooked transformations (such as peephole optimizations [43]). It was originally defined as a compilation technique that searches for an *optimal* sequence of instructions semantically equivalent to a given *loop-free* sequence. It can hence be classified as a *synthesis* technique, where the original sequence acts as a specification of the semantics for the optimal code and the search is automated often using a constraint solver (e.g., an SMT [10]

Authors' addresses: Elvira Albert, Complutense University of Madrid, Madrid, Spain, elvira@sip.ucm.es; Maria Garcia de la Banda, Monash University, Melbourne, Australia, maria.garciadelabanda@monash.edu; Alejandro Hernández-Cerezo, Complutense University of Madrid, Madrid, Spain, aleher06@ucm.es; Alexey Ignatiev, Monash University, Melbourne, Australia, alexey.ignatiev@monash.edu; Albert Rubio, Complutense University of Madrid, Madrid, Spain, alberu04@ucm.es; Peter J. Stuckey, Monash University, Melbourne, Australia, peter.stuckey@monash.edu.

ACM 2475-1421/2024/6-ART205
https://doi.org/10.1145/3656435

or SAT [41] engine). Recent advances in constraint solving have fostered the development of successful superoptimization tools and their application to real programs using state-of-the-art SMT solvers, as it is the case for the superoptimizers developed for LLVM [32, 44, 55], Ethereum VM (EVM) [3, 46], and WebAssembly (Wasm) [15]. While the original and most common *objective function* so far has been program length reduction, EVM superoptimizers use other criteria relevant in the blockchain context, namely gas and size-in-bytes reduction [66].

Superoptimizers yield code that is always *correct* (i.e., semantically equivalent to the original) and not just *effective* (i.e., its performance *wrt.* the objective function is improved if possible) but *optimal*. However, they are not widely adopted in practice, because their search for optimality can make them so *inefficient* as to become impractical. In particular, the complexity of the optimality search grows exponentially with the *length of the solution* being constructed, and becomes intractable for large sequences of instructions. State-of-the-art approaches use the length of the original sequence as a bound on the length of the solution, and rely on SMT solvers to carry out the search.

Contributions: The challenge we face is to devise a scalable approach to the superoptimization of *stack-bytecode* that is correct, effective and efficient. More precisely, the problem we tackle can be described as follows: *given a sequence b of operation codes (or opcodes) that produces output state S, the superoptimizer searches for a sequence o that also produces S and is optimal regarding stack and register manipulation opcodes.* The next three insights have led to the development of three novel components to solve this problem. To the best of our knowledge, these components have not been used in superoptimization before and constitute our main contributions:

(1) Insight: tight bounds on the maximal length of the solution can critically improve the search. Contribution: we propose a *greedy* algorithm which, given output state S, generates a sequence of opcodes o' whose length is often smaller than that of the original sequence b. If smaller, it is used to bound the length of the optimal code o, and as the initial solution (to be improved upon). This is particularly relevant when optimizing large code blocks.
(2) Insight: *domain-specific constraints* can help solvers speed up the search significantly. Contribution: we present *new* powerful constraints specific to the stack-bytecode domain that allow solvers to eliminate areas of the search that (almost) always lead to failure (referred to as *redundant constraints*), or to remove solutions whose objective value is known to be dominated – i.e., equal or worse – than those not eliminated (*dominance constraints*).
(3) Insight: the target synthesis problem can be reduced to *MaxSAT*. Contribution: we propose an *efficient propositional encoding* for the specification, the dominance and redundant constraints above, and any of the objective functions used. In addition to the ability to directly handle the objective functions by reducing the problem to Weighted Partial Maximum Satisfiability [7], we also propose a *bespoke method to effectively cope with program length minimization* via a series of pure SAT oracle calls.

Impact: As stack-based bytecode is used by modern compilers (e.g., WebAssembly [65], Ethereum bytecode [66], and JVM [36] rely on stack machines), our proposal has wide applicability. To assess its efficiency and impact in a real setting we have developed a new tool, named SuperStack, that can be used for both EVM and Wasm superoptimization. For Wasm code, reducing its length has a significant impact on the time to respond to requests. Our experiments show SuperStack achieves 9% code length reduction over existing Wasm superoptimization [15]. For Ethereum bytecode, we consider not only code size reduction but also *gas* consumption, which measures the computational and storage cost of executing the EVM instructions. Reducing gas consumption has a direct impact on the transaction costs and allows processing more transactions in the overall Ethereum ecosystem. We experimentally compared SuperStack with `GASOL` [4], an existing MaxSMT-based tool for superoptimizing EVM bytecode [66]. The experiments use more than 1,000 deployed smart contracts

Table 1. A General Stack Bytecode Language with Registers and Heap

(1)	POP	removes the topmost element;
	SWAPx	swaps the topmost element with the one in position x+1;
	DUPx	duplicates the element in position x and adds it as the new topmost element;
(2)	SETx	stores the topmost element in register with index x and removes it from the stack;
	TEEx	stores the topmost element in register with index x without modifying the stack;
	GETx	adds the value stored in register with index x as the new topmost;
(3)	STORE	modifies the memory location given by the topmost element with the value given by the next stack element, and removes both elements from the stack;
	LOAD	copies the value at the memory location given by the topmost element onto the old topmost element;
(4)	UF	any other instruction in the language (e.g., arithmetic, bit-wise, etc) will be considered as an uninterpreted opcode

(including the 100 contracts with more transactions received so far). Our results show our approach multiplies the optimization gains by at least two (depending on the objective function), while reducing optimization time to at least a fourth.

Organization: The rest of the paper is organized as follows. Section 2 describes the stack-bytecode language we use to formalize our approach and outlines a general algorithm for superoptimization of stack-bytecode. Section 3 introduces our greedy algorithm which returns a sequence of opcodes that is shorter than the original one while producing the same (symbolic) state. Section 4 describes our SAT-encoding for searching for an optimal solution and introduces new dominance and redundant constraints for stack-bytecode optimization that can improve solver performance. Section 5 reports the results of our thorough experimental evaluation which prove the practical impact of our approach. Lastly, Section 6 overviews related work and concludes.

2 SUPEROPTIMIZATION OF STACK BYTECODE

This section introduces the bytecode language considered in this paper, as well as an algorithmic description of the superoptimization technique for such language. The algorithm is then used to give an overview of the basic components used by the state-of-the-art superoptimization tools.

2.1 The General Stack Bytecode Language

We consider a general stack bytecode language that stores data in three different regions: *the stack*, for which we assume a standard stack-based architecture with words of fixed size and an associated stack-based bytecode language with the opcodes shown at row (1) of Table 1; *the registers*, which are virtual and used to store temporary values accessed using the opcodes shown at row (2) (opcode TEEx is unique to Wasm; we include it as part of the language due to its pivotal role in optimization); and *the memory*, for which we assume a standard volatile heap accessed using the opcodes at row (3). Any other opcode in the language whose actual semantics is not encoded for superoptimization falls into the UF (Uninterpreted Function) category, and we are only interested in the arity of its opcode (denoted $ar(\mathtt{UF})$), its commutative/noncommutative behavior and its cost. In addition, opcodes that may have side-effects must be taken into account when producing the sequences to which superoptimization will be applied. In particular, to ensure we maintain the same semantics, we superoptimize separately the sequence up to the instruction with side-effects and the sequence from that instruction to the end.

This general stack bytecode language allows us to represent languages such as EVM, which hold all intermediate values in the operand stack, and manage it directly by swapping and duplicating its

elements through specific opcodes. Their instruction set encompasses rows (1), (3) and (4). It also allows us to represent languages such as Wasm, which declare virtual registers to store intermediate values that can be accessed afterwards, and use them to perform swaps and duplication of stack elements. Their instruction set encompasses (2), (3), (4) and POP from (1). Other stack bytecode languages represented lie in-between, e.g., JVM combines stack manipulation opcodes and registers.

2.2 An Overview of the Superoptimization Algorithm

Algorithm 1 shows the superoptimization algorithm implemented in SuperStack. The instructions displayed in black instantiate it into existing algorithms implemented in state-of-the-art superoptimization tools. Our new components are displayed in gray. This section focuses on the black components, leaving the new gray ones for Secs. 3 and 4. The algorithm receives as input (line 2) the bytecode program b to be superoptimized, an objective function f to be minimized and optionally a timeout time to end up the search. We assume the cost f(b') of any bytecode program b' is defined as the sum of the cost f(i) of each instruction i in b', and that the cost of any instruction is known statically. As mentioned before, the original f used in superoptimization computes the bytecode program's *length* [9, 32, 55] measured by its number of operations. However, other objectives have been introduced by new programming environments. In particular, ebso [46], SYRUP [4] and GASOL [2] minimize the program's *gas*, while GASOL can also minimize its *size-in-bytes*. The former aims to measure the price to pay for executing its instructions, thus assigning larger gas cost to instructions that require more computation or storage. The latter differs from program length only in the size of the opcode PUSH(v) which varies according to the size in bytes of the pushed value v (the PUSH opcode is handled as an uninterpreted opcode UF within our framework and is hence not included in row (1) of Table 1). A precise definition of both criteria appears in [66]. Our SAT-based approach in Sec. 4 and the experimental evaluation in Sec. 5 use these three objective functions.

One can also express other objective functions based on metrics like data locality or number of clock cycles in an instruction pipeline. In the first case, we will need to reduce the distance between

1: **procedure** SuperStack(b,f,[time,]c)
2: **Input:** bytecode program b, objective function f, optional timeout time, dominance/redundant constraints c
3: **Output:** optimized bytecode o
4: **Ensures: if** time not used **then** f(o) = min{f(o') | o' = b} **else** f(o) ≤ f(b)
5: seqs ← *sequences*(b)
6: result ←[ ]
7: **for** (seq,ini,r_f) ∈ seqs **do**
8: (fin,n^{max},s^{max},r^{max}) ← *symbolic*(seq,ini,r_f)
9: (g^{max},seqGrd) ← Greedy(ini,fin)
10: (isOptimal,seqSAT) ← SearchSAT (c,f,ini,fin,s^{max},r^{max},min(g^{max},n^{max}),seqGrd[,time])
11: result ← result.*append*((*min*(f,seq,seqGrd,seqSAT),isOptimal))
12: **return** o ← *rebuild*(result)

Algorithm 1. A novel greedy and SAT-based superoptimization algorithm. Initially, jump-free sequences are identified in the *sequences* procedure. For each sequence seq, ini represents the initial symbolic state, and r_f the number of registers that live-out. This information is used for the symbolic execution of seq, resulting in the symbolic state fin, which is employed in both Greedy and SearchSAT. Greedy generates an equivalent sequence seqGrd to refine the bounds, while SearchSAT produces the equivalent sequence seqSAT and the variable isOptimal to indicate the optimality status. s^{max}, r^{max}, n^{max} and g^{max} denote parameters utilized in SearchSAT: the maximum stack depth allowed, the maximum number of registers, and an upper bound on the number of opcodes determined by the initial sequence and Greedy, resp.

memory instructions that work on close positions. This can be expressed using constant weights on the decisions since we consider a bounded known number of instructions and hence the cost of having two instructions at distance d can be expressed with a constant value that depends on d (for all possible distances in the given number of instructions). For the instruction pipeline, the approach would be, roughly, to assign to each instruction the clock cycle it starts and additionally encode all restrictions on which operations can be handled in parallel in a given cycle. Then, we just need to minimize the needed number of cycles instead of the number of operations.

The algorithm returns as output (line 3) an optimized bytecode o. If a given timeout (optional parameter time in line 10) is not reached by the solver, o is ensured to be optimal (line 4), i.e., it has minimal cost *wrt.* f and is semantically equivalent (denoted =) to b. The remainder of this section is devoted to describe the procedures *sequences* and *symbolic* invoked in lines 5 and 8, resp. Procedure Search in line 10 is assumed to use constraint-solving techniques to perform a complete search for an optimal solution within the bounds given. Thus, optimality is always up to the considered bounds (see Sec. 4). As constraint solvers are often able to produce solutions while searching for an optimal one, Search also yields a boolean isOptimal to indicate whether the solution found (if any) is known to be optimal. This is used in line 11 to ensure the result returned is the best (according to objective function f) between the original sequence and those found by the Greedy and Search procedures (if any). Note that if the timeout is reached, we can only ensure the sequence o returned by the superoptimizer is equal or better than the original b (else condition in line 4), rather than optimal. The final step uses *rebuild* (line 12), a standard procedure to reconstruct executable code from optimized sequences (e.g., jump addresses might need to be recomputed).

2.3 Generation of Jump-free Sequences

As superoptimization is applied to loop-free sequences, the first step for all superoptimizers is to build a control flow graph (CFG) from the bytecode program that allows detection of its loops and branching. A common approach is to generate one sequence per block of the CFG; hence, the sequence is not only loop-free but also *jump-free*. This is done, for example, by the superoptimizers `ebso` [46], `SYRUP` [4], and `Souper` [55]. In this first step one can also statically compute, for each block, the number k of elements needed in the operand stack before the block, and the number r of registers that are relevant before or after the block, i.e., those registers whose first access in the block or in a trace after the block, respectively, is a read. The k is used to create an initial stack with k variables `istack`=$[s_0, \ldots, s_{k-1}]$. Stacks are represented from top to bottom, with `istack`[0] representing the topmost element s_0. The r is used to renumber the registers from 0 to $r-1$ (a mapping to the original numbers is kept to reconstruct the code) with the r_f registers that *live-out*, i.e., they are relevant *after* the block, taking the initial (i.e., 0 to $r_f - 1$) numbers. It is also used to create a list with r variables `iregs`=$[s_k, \ldots, s_{k+r-1}]$ for the registers. Regarding memory, we use "[]" to denote that no information is known about the memory contents when starting the analysis of the sequence. The *initial state* is made up of the initial stack, registers and an unknown memory.

Definition 2.1 (sequences). Given bytecode program b, procedure *sequences*(b) returns a set of tuples (seq, ini, r_f) with all the jump-free sequences of b, where seq is the sequence, ini=(`istack`,`iregs`, []) contains the initial state, and r_f is the number of relevant registers after the sequence.

The longer the sequence, the more potential for optimization but the longer the search in general. This is why some approaches, e.g., unbounded optimization [32], generate the longest possible loop-free sequences while others, e.g., `GASOL` [2], even split the jump-free sequences when their number of instructions is larger than a given threshold to avoid search timeouts. Our framework admits any such implementation of the *sequences* procedure. Our current implementation works with jump-free sequences and allows splitting them into smaller subsequences if requested. Working

(stack, regs, mem)	$\rightarrow_{\text{pp:POP}}$	(stack[1 : n], regs, mem)
(stack, regs, mem)	$\rightarrow_{\text{pp:SWAPx}}$	(stack′, regs, mem), stack′[0] = stack[x], stack′[x] = stack[0], stack′[i] = stack[i] $\forall i > 0, i \neq$ x
(stack, regs, mem)	$\rightarrow_{\text{pp:DUPx}}$	([stack[x − 1] | stack], regs, mem)
(stack, regs, mem)	$\rightarrow_{\text{pp:SETx}}$	(stack[1 : n], regs[$x \mapsto$ stack[0]], mem)
(stack, regs, mem)	$\rightarrow_{\text{pp:TEEx}}$	(stack, regs[$x \mapsto$ stack[0]], mem)
(stack, regs, mem)	$\rightarrow_{\text{pp:GETx}}$	([regs[x] | stack], regs, mem)
(stack, regs, mem)	$\rightarrow_{\text{pp:STORE}}$	(stack[2 : n], regs, mem ++ [STORE_{pp}(stack[0], stack[1])]))
(stack, regs, mem)	$\rightarrow_{\text{pp:LOAD}}$	([LOAD_{pp}(stack[0]) | stack[1 : n]], regs, mem ++ [LOAD_{pp}(stack[0])]))
(stack, regs, mem)	$\rightarrow_{\text{pp:UF}}$	([UF(stack[0], ..., stack[δ − 1]) | stack[δ : n]], regs, mem), $\delta = ar$(UF)

Fig. 1. Symbolic Execution of Opcodes

Table 2. EVM ($_1$) and Wasm ($_2$) sequences, greedy (seqGrd$_*$) and SAT (seqSAT$_*$) optimized and their lengths

seq_1	PUSH(64) LOAD PUSH(7) DUP3 STORE PUSH(0) PUSH(128) LOAD SWAP4 POP SWAP4 SWAP1 DUP3 STORE SWAP1 POP	16
seqGrd_1	PUSH(0) SWAP3 SWAP1 PUSH(64) LOAD PUSH(7) DUP3 STORE PUSH(128) LOAD SWAP4 POP SWAP1 STORE	14
seqSAT_1	DUP1 PUSH(7) PUSH(64) LOAD SWAP3 STORE PUSH(128) LOAD SWAP3 POP STORE PUSH(0) SWAP2	13
seq_2	LOAD* SET2 PUSH(0) PUSH(1) STORE GET2 GET1 I32.REM PUSH(2) I32.SHL SET0 PUSH(1) SET2	13
seqGrd_2	LOAD* GET1 I32.REM PUSH(2) I32.SHL SET0 PUSH(0) PUSH(1) TEE2 STORE	10
seqSAT_2	PUSH(0) PUSH(1) TEE2 LOAD* GET1 I32.REM PUSH(2) I32.SHL SET0 STORE	10

with loop-free rather than jump-free sequences simply requires adding jump-instructions to the language and merging blocks within the CFG to propagate known information from the caller block to the callee (e.g., propagating the conditions in jump instructions) to build the sequences.

2.4 Symbolic Execution

Before the search, most superoptimization approaches *symbolically execute* the jump-free sequence being optimized to obtain and simplify the resulting *symbolic state*. The search is then done using this state as input, rather than the sequence. We use (stack, regs, mem) $\rightarrow_{\text{pp:OP}}$ (stack', regs', mem') to represent the symbolic execution of opcode OP at program point pp over a symbolic state with operand stack stack, registers regs and memory accesses mem, which results in a symbolic state with operand stack stack', registers regs' and memory accesses mem'. We use $\rightarrow^*_{\text{seq}}$ rather than $\rightarrow_{\text{pp:OP}}$ to abbreviate the symbolic execution of the opcodes in jump-free sequence seq. Figure 1 defines the symbolic execution of the opcodes in our language. Memory opcodes are labeled with their program point pp for distinguishing them later. The subindex pp is omitted when there is no need to use it. We follow standard notation for lists and its operations: a list L of n elements is denoted as $L[0 : n-1]$, $L[k]$ denotes the element in position k, $[a \mid L]$ denotes the list resulting from adding a to the beginning of L, L[x $\mapsto$ v] indicates the value for position x in L is now v (i.e., L[x $\mapsto$ v][x] = v) while the rest remains unchanged, and operator ++ concatenates two lists. For every (seq,ini, r_f) in *sequences*(b) of bytecode program b, we symbolically execute ini=(istack, iregs, []) $\rightarrow^*_{\text{seq}}$ fin=(stack, regs, mem) where stack and regs are the resulting *final* stack and registers and mem the list of memory accesses within seq.

Example 2.2. We illustrate our approach using an EVM sequence (seq_1 in Table 2) and a Wasm sequence (seq_2) found in real code from our benchmarks in Sec. 5. Opcodes in the sequences that are handled as UF in our framework are underlined. Also, LOAD* in seq_2 denotes a variant of opcode LOAD that consumes no element from the stack instead of the topmost; Sec. 2.5 discusses why LOAD (and STORE) instructions with different arity are introduced in some cases when applying our framework to Wasm. The following shows some steps of the symbolic execution of seq_1 for istack = $[s_0, s_1, s_2]$ and iregs = [], as well as the final state fin_1:

$$\begin{aligned}\texttt{ini}_1 \equiv\ & ([s_0, s_1, s_2], [], []) \rightarrow_{\texttt{0:PUSH(64)}} ([\texttt{PUSH(64)}, s_0, s_1, s_2], [], []) \rightarrow_{\texttt{1:LOAD}} \\ & ([\texttt{LOAD}_1(\texttt{PUSH(64)}), s_0, s_1, s_2], [], [\texttt{LOAD}_1(\texttt{PUSH(64)})]) \rightarrow^* \ldots \rightarrow_{\texttt{15:POP}} \\ & ([s_2, \texttt{LOAD}_7(\texttt{PUSH(128)}), \texttt{PUSH(0)}], [], [\texttt{LOAD}_1(\texttt{PUSH(64)}), \texttt{STORE}_4(s_0, \texttt{PUSH(7)}), \\ & \texttt{LOAD}_7(\texttt{PUSH(128)}), \texttt{STORE}_{13}(s_0, \texttt{LOAD}_1(\texttt{PUSH(64)}))]) \equiv \texttt{fin}_1\end{aligned}$$

For $\texttt{seq}_2$ with `istack` = [] and `iregs` = $[s_0, s_1, s_2]$, some steps of its symbolic execution are as follows:

$$\begin{aligned}\texttt{ini}_2 \equiv\ & ([], [s_0, s_1, s_2], []) \rightarrow_{\texttt{0:LOAD}^*} ([\texttt{LOAD}_0^*], [s_0, s_1, s_2], [\texttt{LOAD}_0^*]) \rightarrow_{\texttt{1:SET2}} ([], [s_0, s_1, \texttt{LOAD}_0^*], [\texttt{LOAD}_0^*]) \\ & \rightarrow^* \ldots \rightarrow_{\texttt{12:SET2}} ([], [\texttt{I32.SHL}(\texttt{PUSH(2)}, \texttt{I32.REM}(s_1, \texttt{LOAD}_0^*)), s_1, \texttt{PUSH(1)}], \\ & [\texttt{LOAD}_0^*, \texttt{STORE}_4(\texttt{PUSH(1)}, \texttt{PUSH(0)})]) \equiv \texttt{fin}_2\end{aligned}$$

During symbolic execution, one can obtain upper bounds on the number of opcodes (denoted $\texttt{n}^{max}$), on the size of the stack (denoted $\texttt{s}^{max}$), and on the number of the registers (denoted $\texttt{r}^{max}$) allowed in a solution (see, e.g., [4, 32]). These bounds will be used in the next steps.

2.4.1 Memory Dependencies. A dependency analysis can be used to infer (in)dependencies among the memory accesses. A dependency of the form $x < y$ indicates that memory access x must happen before y. In the absence of a dependency analysis, one must keep the order in which memory accesses appear in seq. However, more advanced approaches (see e.g. [2, 9]) may be able to detect independence among accesses. For instance, *read* access $\texttt{LOAD}_i$ does not have to depend on former *write* access $\texttt{STORE}_j$ if they access different memory locations. Other types of independencies are described in [2, 9]. Our framework works for any correct analysis that provides such a dependency list. For the sake of generality, we assume `mem` contains the list of memory accesses performed during the symbolic execution and, optionally, a dependency list referred to as deplist(`mem`), whose elements provide partial orders indicating the order in which the accesses must happen.

Example 2.3. In the absence of a dependency analysis, or if the dependency analysis cannot prove independence between any of the accesses in $\texttt{seq}_1$, the dependency list would be deplist(mem)=$[\texttt{LOAD}_1 < \texttt{STORE}_4 < \texttt{LOAD}_7 < \texttt{STORE}_{13}]$, which provides a total order among the accesses (and is assumed for the next examples in the paper). If the analysis infers that $\texttt{LOAD}_1$ is independent from the remaining accesses, the dependency list would instead be $[\texttt{STORE}_4 < \texttt{LOAD}_7 < \texttt{STORE}_{13}]$, thus providing more freedom to place $\texttt{LOAD}_1$. If the analysis infers that $\texttt{LOAD}_1$ must happen before $\texttt{STORE}_4$, $\texttt{LOAD}_7$ must happen before $\texttt{STORE}_{13}$, and the order among the two pairs is irrelevant, the dependency list would be $[\texttt{LOAD}_1 < \texttt{STORE}_4, \texttt{LOAD}_7 < \texttt{STORE}_{13}]$.

2.4.2 Peephole Optimizations. The final symbolic state can be optionally (rule-based) optimized before starting the search. This enables optimizing the arithmetic and bit-wise opcodes (abstracted by means of `UF`) that are left out of the SMT- or SAT-encodings (see Sec. 5). For example, the following are peephole optimizations involving opcode `MUL`:

$$\begin{aligned}&\texttt{MUL(PUSH(}v_1\texttt{), PUSH(}v_2\texttt{))} \mapsto v_1 \cdot v_2, \quad \texttt{MUL(X, PUSH(0))} \mapsto \texttt{PUSH(0)}, \\ &\texttt{MUL(X, PUSH(1))} \mapsto \texttt{X}, \quad \texttt{MUL(SHL(PUSH(1), X), Y)} \mapsto \texttt{SHL(Y, X)},\end{aligned}$$

where, for example, the first rule replaces the multiplication of constants by its result. A peephole optimization phase achieves most optimizations on these opcodes without threatening the applicability of superoptimization (see comparison with `ebso` superoptimizer in Sec. 6).

2.4.3 Flattening and Notation. To formalize the method, it is convenient for the final symbolic state (`stack`, `regs`, `mem`) to be *flattened* by introducing new fresh variables $s_{k+r}, s_{k+r+1}, \ldots, s_m$ after the last element s_{k+r-1} in the initial registers iregs, and creating a mapping `map` for each fresh element to a *shallow term*, i.e., to an expression built by a function symbol and fresh elements as arguments. We assume the *minimal* number of fresh elements needed to describe `stack`, `regs` and `mem` via shallow expressions is introduced. This is achieved by using the same element if some subterm occurs more

than once, and by taking into account the commutativity properties of UF opcodes to avoid creating unnecessary elements. Let $\texttt{StackElems} = \{s_0, \ldots, s_{k+r-1}, s_{k+r}, \ldots, s_m\}$ denote the full set of elements either initial or fresh. Then, after flattening, stack and regs contain only elements in StackElems, mem contains only shallow terms whose arguments are in StackElems, and map denotes the mapping of elements in $\{s_{k+r}, \ldots, s_m\}$ to shallow terms formed by either an uninterpreted function or $\texttt{LOAD}_{\texttt{pp}}$ applied to elements in StackElems. Note that opcodes POP, SWAPx, DUPx, SETx, TEEx, GETx and STORE are never added to the stack. The following simplified grammar characterizes the flattened symbolic state as assumed in the remaining sections, where superindex $*$ denotes any number of instances and subindex $_i$ identifies the number of elements created:

$$
\begin{array}{lcl}
\textit{stack-elem} & = & s_i, \text{ where } s_i \in \texttt{istack} \cup \texttt{iregs} \cup \textit{dom}(\texttt{map}) \\
\textit{map-elem} & = & s_i \mapsto \texttt{UF}(\textit{stack-elem}_0, \ldots, \textit{stack-elem}_{ar(\texttt{UF})-1}) \mid s_i \mapsto \texttt{LOAD}_{\texttt{pp}}(\textit{stack-elem}_0) \\
 & & \text{where } s_i \notin \texttt{istack} \cup \texttt{iregs} \\
\texttt{map} & = & \{\textit{map-elem}^*\} \\
\texttt{stack} & = & [\textit{stack-elem}^*] \\
\texttt{regs} & = & [\textit{stack-elem}_0, \ldots, \textit{stack-elem}_{r-1}] \\
\textit{mem-elem} & = & \texttt{LOAD}_{\texttt{pp}}(\textit{stack-elem}_0) \mid \texttt{STORE}_{\texttt{pp}}(\textit{stack-elem}_0, \textit{stack-elem}_1) \\
\texttt{mem} & = & [\textit{mem-elem}^*]
\end{array}
$$

In what follows, we use Terms to denote the set of terms mapped by stack, regs and mem (i.e., the image of map), as well as all terms of mem that correspond to store elements. From any flattened symbolic state st, we can access st.stack, st.regs, st.mem, st.map, st.StackElems and st.Terms.

Example 2.4. States $\texttt{fin}_1$ and $\texttt{fin}_2$ are flattened into $([s_2, s_3, s_4], [\,], [\texttt{LOAD}_1(s_6), \texttt{STORE}_4(s_0, s_7), \texttt{LOAD}_7(s_5), \texttt{STORE}_{13}(s_0, s_8)])$ and $([\,], [s_8, s_1, s_5], [\texttt{LOAD}_0^*, \texttt{STORE}_4(s_5, s_4)])$ resp., with

$$
\begin{array}{rl}
\texttt{fin}_1\texttt{.map} = \{ & s_3 \mapsto \texttt{LOAD}_7(s_5), s_4 \mapsto \texttt{PUSH(0)}, s_5 \mapsto \texttt{PUSH(128)}, \\
 & s_6 \mapsto \texttt{PUSH(64)}, s_7 \mapsto \texttt{PUSH(7)}, s_8 \mapsto \texttt{LOAD}_1(s_6)\} \\
\texttt{fin}_2\texttt{.map} = \{ & s_3 \mapsto \texttt{LOAD}_0^*, s_4 \mapsto \texttt{PUSH(0)}, s_5 \mapsto \texttt{PUSH(1)}, \\
 & s_6 \mapsto \texttt{I32.REM}(s_1, s_3), s_7 \mapsto \texttt{PUSH(2)}, s_8 \mapsto \texttt{I32.SHL}(s_7, s_6)\}
\end{array}
$$

Definition 2.5 (symbolic). Given the jump-free sequence seq, the initial state ini, and the number of relevant final registers r_f, the generic procedure $\textit{symbolic}(\texttt{seq}, \texttt{ini}, r_f)$ returns $(\texttt{fin}, \texttt{n}^{max}, \texttt{s}^{max}, \texttt{r}^{max})$, as the result from flattening and (rule-based) optimizing the symbolic state obtained from applying $\texttt{ini} \rightarrow^*_{\texttt{seq}}$ (possibly using a memory dependency analysis) and taking from fin.regs only the first r_f elements, as well as the bounds obtained during symbolic execution for the length ($\texttt{n}^{max}$), stack size ($\texttt{s}^{max}$) and registers size ($\texttt{r}^{max}$) of the solution.

We can now specify the semantic equivalence relation that forces the stack and the memory accesses obtained by two different sequences given the same initial state to match, but allows extra registers to store intermediate values and allows any order of memory accesses as long as they are coherent with the list of dependencies.

Definition 2.6 (semantic equivalence). Two jump-free sequences seq, seq' are semantically equivalent up to n registers (denoted $=_n$) iff for every symbolic state ini, if $\texttt{ini} \rightarrow^*_{\texttt{seq}} \texttt{fin}$, $\texttt{ini} \rightarrow^*_{\texttt{seq}'} \texttt{fin}'$ and assuming the shallow terms shared in fin.map and fin'.map are assigned to the same element s_i, then fin.stack = fin'.stack, $\texttt{fin.regs}[x] = \texttt{fin}'\texttt{.regs}[x]\ \forall x \in \{0, \ldots, n-1\}$, fin.map = fin'.map and fin'.mem satisfies the dependencies in deplist(fin.mem) and vice versa. If deplist is not provided, then fin.mem = fin'.mem.

In what follows, the relation = refers to $=_{r_f}$, where r_f corresponds to the inferred number of registers that are relevant after the sequence we are trying to optimize.

2.5 Instantiation of the Framework to EVM and Wasm

Adapting the presented superoptimization framework to EVM and Wasm requires additional considerations regarding their respective architectures and data structures. In the case of EVM, the set of SWAPx and DUPx opcodes ranges from x = 1 to 16. Furthermore, EVM employs two distinct data regions in addition to the operand stack: the *storage*, which stores persistent data between external function calls; and a local volatile memory used to allocate dynamic local data. Both data structures are concrete cases of the broader concept of memory, and two corresponding functions manage the dependencies for each type of access. In the case of Wasm, three data regions are used: the *local variables* or *locals*, which are registers used to hold temporary values within the scope of a function; the *global variables* or *globals*, which are global registers; and a linear memory. In our framework, *locals* correspond to the registers, while both *globals* and the linear memory are concrete types of memory. *Globals* are considered as memory because external functions can be invoked in the middle of the block and change their contents, thus enforcing dependencies among accesses to globals. In general, opcodes that have dependencies with other instructions are treated similarly to LOAD or STORE, depending on whether they introduce values in the stack or not. Function deplist provides an abstraction for handling such dependencies.

Example 2.7. Sequence $\mathtt{seq}_2$ corresponds to the Wasm sequence "`call $rand local.set 4 i32.const 0 i32.const 1 i32.store local.get 4 local.get 2 i32.rem_s i32.const 2 i32.shl local.set 0 i32.const 1 local.set 4`", where `$rand` is an external function (i.e., not declared in the Wasm program) that receives no parameters and returns a value as a result (represented by the opcode LOAD* in our framework). Instructions `call $rand` and `i32.store` are dependent (written as $\mathtt{LOAD}_0^* < \mathtt{STORE}_4(s_5, s_4)$ in deplist), as `$rand` can modify the linear memory.

3 THE GREEDY ALGORITHM

The Greedy procedure invoked by Algorithm 2 aims to, given the initial symbolic state ini and its corresponding final state fin obtained by *symbolic* in line 8, quickly return a sequence of opcodes seqGrd that is shorter than seq such that seqGrd = seq. Starting with ini, the algorithm determines the necessary operations to transform it into fin. The process is quick because seqGrd is obtained by making greedy (i.e., locally optimal) decisions rather than globally optimal ones. The key idea is to minimize the number of stack operations and register accesses by prioritizing operations that place the top of the stack in its final position at each step when possible. This is achieved by removing unnecessary stack elements as soon as they reach the top of the stack and placing the element in its final position (given by fin.stack and fin.regs) when possible. When neither step can be taken, either a new stack element is computed and moved to its final position in the subsequent step; or a memory access is performed in a compatible order with the memory dependencies. This process is repeated until the current state becomes a *permutation* of fin. A permutation of a symbolic state st is any symbolic state st' such that the stack elements in st.stack and st.regs appear either in st'.stack or in st'.regs, and vice versa. Further, st'.mem must be a permutation of st.mem, thus ensuring the same memory accesses have been performed although not necessarily in the same order. The greedy algorithm invokes function $uses(s_i, \mathtt{st}', \mathtt{st})$, which counts the minimal number of times a stack element s_i is needed to go from st' to st. It indicates how many times a stack element must be computed and/or duplicated to produce st and can be obtained by counting the minimal number of occurrences of s_i needed to compute the elements in the (multiset) difference between the union of st.stack, st.regs and the store elements in st.mem, and those in st'.

The algorithm proceeds as follows. Variables current and seqGrd, initialized in lines 5 and 6, represent the current state and the operations computed so far, resp., and satisfy invariant $\mathtt{ini} \rightarrow^*_{\mathtt{seqGrd}} \mathtt{current}$ at each step. The loop started in line 7 modifies current and seqGrd until current is a

```
1:  procedure GREEDY(ini, fin)
2:   Input: initial and final states from the symbolic execution of a sequence seq
3:   Output: greedy sequence seqGrd and its length
4:   Ensures: seqGrd ≡ seq
5:   current ← ini
6:   seqGrd ← []
7:   while current ≠ permutation(fin) do
8:    if length(current.stack) > 0 ∧ uses(current.stack[0],current,fin) = 0 then
9:     current.stack ← pop(current.stack)
10:    seqGrd ← seqGrd.append(POP)
11:   else if length(current.stack) > 0 ∧ misplaced(current.stack[0],fin)
12:            ∧ canBePlaced(current,fin) then
13:    (current, ops) ← moveTopToPosition(current)
14:    seqGrd ← seqGrd.append(ops)
15:   else
16:    v ← chooseNextElement(current,fin)
17:    (ops,current) ← computeElement(v,current,fin)
18:    seqGrd ← seqGrd.append(ops)
19:  seqGrd ← solvePermutation(current,fin,seqGrd)
20:  return (length(seqGrd),seqGrd)
```

Algorithm 2. A greedy algorithm for stack-bytecode optimization

permutation of `fin` (condition in line 7). This condition can be assessed in $O(1)$ by tracking which computations have yet to be completed. Once this happens, `fin` can be obtained from `current` by reordering its elements. This is achieved (line 19) by *solvePermutation*(`current`,`fin`,`seqGrd`), which performs the minimum number of instructions in `current` to yield `fin` and adds those instructions to `seqGrd`. Its time complexity is $O(|\texttt{fin.stack}| + |\texttt{fin.regs}|)$, as it rearranges elements in both structures. If a loop iteration is performed instead, one of the following three cases is followed depending on the topmost element of the current stack (if the stack is empty, case **(iii)** is chosen):

(i) If `current.stack[0]` does not appear in `fin` (line 8), it must be popped and opcode `POP` added to `seqGrd` (lines 9 and 10). To verify this, the algorithm checks if $uses(\texttt{current.stack}[0], \texttt{current}, \texttt{fin}) = 0$, which means it does not appear in `fin` and must be removed.

(ii) If `current.stack[0]` appears in `fin` though is misplaced in `current` (according to `fin`), and can be put in its final position (line 12), then it is placed there (line 13) and the opcodes used are added to `seqGrd` (line 14). An element is already placed in its final position in `current` if it appears in `fin.stack`, there is no register in `fin.regs` in which it must appear and its position *wrt.* the bottom of `current.stack` is the same as it is to the bottom of `fin.stack`. Intuitively, this notion of order works because stacks can only grow and decrease from the top element. For the EVM, a `SWAPx` instruction is used to place the element in its final position in the stack. For Wasm, either a `SETx` or a `TEEx` instruction is used to place the value in a register, depending on whether the element is used in another computation or not. This is discussed in the next subsections in further detail. Once an element is placed in its final position from the bottom, the number of elements deeper cannot vary. This also means that computed terms in `fin.stack` cannot be placed directly in their final position when `current.stack` does not contain enough elements to access the corresponding index.

(iii) Otherwise (line 15), we first call *chooseNextElement*(`current`, `fin`) to select an element `v` from `fin.stack` or `fin.regs` (line 16) that does not occur in `current` the *needed number of times* ($uses(\texttt{v}, \texttt{current}, \texttt{fin}) > 0$), or is an operation of `fin.mem` that satisfies the dependency

constraints in deplist (i.e., must happen according to deplist, has been computed). Then, procedure *computeElement*(v, current, fin) returns the sequence of opcodes that must be applied to current to compute v and updates current. If v corresponds to a stack element that already appears in current.stack or current.regs, then it is duplicated or loaded from a register. Otherwise, *computeElement* computes the corresponding term by placing all its arguments at the top of the stack (potentially triggering the computation of additional elements), and then applying its associated instruction. In the EVM, this process may entail swapping and duplicating existing elements in the stack; in Wasm, it may involve retrieving values from registers or storing values within them (see Sec. 3.1 and 3.2 for more details). Examples 3.1 and 3.2 illustrate this scenario for EVM and Wasm, resp. Importantly, there can be several sequences that achieve this goal due to commutative operands. Choosing a suitable order in this step is crucial to take advantage of the current state of the stack and thus avoid unnecessary stack manipulation operations or moving stack elements to registers.

Generality: Our greedy algorithm is general as it provides a universal strategy for efficient stack management. In particular, its heuristic for selecting the next element is the same for EVM and Wasm: it favours memory instructions and computations that either reuse topmost stack elements or can be placed in their final position directly. Only its implementation differs due to the EVM and Wasm stack management. EVM prioritises elements with values about to become unreachable by DUP/SWAP. Instead, WASM avoids computing elements with values deep within the stack as this would require moving higher stack values into registers. Other stack-based languages can easily implement the same heuristic with minor tweaks. Sec. 3.1 and 3.2 discuss this adaptation to the EVM and Wasm architectures, resp., by describing how the (purposefully) generic algorithm can be applied to both different stack-based architectures with specific instantiations of steps **(ii)** and **(iii)**.

Soundness and Optimality Results: The greedily optimized sequences are sound but may not be optimal. Soundness and termination follow from the fact that current undergoes a transformation that brings it closer to fin in every step: it either removes an element that is not used, places an element in its position in fin, or computes a needed element while satisfying the dependencies in deplist. Optimality is empirically proven only when the SAT solver (in Sec. 4) returns the same optimal value. Our experiments show this happens often in practice.

3.1 Implementation of EVM Greedy in SuperStack

The implementation of Greedy for the EVM focuses on minimizing the number of SWAPx operations used to produce fin.stack. The key idea is in step **(ii)**, which places the stack elements according to their position *wrt.* the bottom of fin.stack when possible: once an element appears in this position, it does not need to be swapped further. To detect if current.stack[0] is *misplaced* but *canBePlaced* in fin, we compute its corresponding position x in current.stack according to fin.stack. We distinguish two cases: (a) If x = 0, then it is already placed in its position and the *misplaced* check fails. As a small example, consider the initial sequence SWAP1 SWAP2 SWAP1 PUSH(0), with the symbolic states ini = ($[s_0, s_1, s_2]$, [], []) and fin = ($[s_3, s_0, s_2, s_1]$, [], []), along with fin.map = $\{s_3 \mapsto$ PUSH(0)$\}$. The greedy algorithm starts with current set to ini and checks if s_0 is *misplaced*. Using the formula $x = length(\texttt{current.stack}) - length(\texttt{fin.stack}) + pos(\texttt{fin.stack}, s_0) = 3 - 4 + 1 = 0$ (where *pos* indicates the position of s_0 in fin.stack), it determines that s_0 is correctly positioned, being the third element from the bottom in both current.stack and fin.stack. Consequently, the greedy algorithm falls into case **(iii)** and computes PUSH(0). (b) The second case is when x > 0. Then, we can place it in its final position with a SWAPx operation. Otherwise, there are not enough

Table 3. Execution of EVM Greedy for $\texttt{ini}_1$ and $\texttt{fin}_1$ in Example 2.4. The table shows the state before the corresponding iteration I, followed by the case selected based on this state and the resulting sequence of instructions. When the algorithm falls into case **(iii)**, we first show the term chosen by *chooseNextElement* and then the steps followed by *computeElement* to compute it. When the computation is straightforward, both procedures are combined into a single step.

I	current.stack	current.mem	Case	Action	seqGrd
1	$[s_0, s_1, s_2]$	[ ]	**(iii)**	Compute $s_4 \mapsto$ PUSH(0)	[PUSH(0)]
2	$[s_4, s_0, s_1, s_2]$	[ ]	**(ii)**	Swap to position 3	[..., SWAP3]
3	$[s_2, s_0, s_1, s_4]$	[ ]	**(ii)**	Swap to position 1	[..., SWAP1]
4	$[s_0, s_2, s_1, s_4]$	[ ]	**(iii)**	Compute $s_8 \mapsto \texttt{LOAD}_1(s_6)$	[..., PUSH(64),$\texttt{LOAD}_1$]
5	$[s_8, s_0, s_2, s_1, s_4]$ $[s_7, s_8, s_0, s_2, s_1, s_4]$ $[s_0, s_7, s_8, s_0, s_2, s_1, s_4]$	$[\texttt{LOAD}_1(s_6)]$ [...] [...]	**(iii)**	Term: $\texttt{STORE}_4(s_0, s_7)$ Compute $s_7 \mapsto$ PUSH(7) Duplicate s_0 Compute $\texttt{STORE}_4(s_0, s_7)$	 [..., PUSH(7)] [..., DUP3] [..., $\texttt{STORE}_4$]
6	$[s_8, s_0, s_2, s_1, s_4]$	[..., $\texttt{STORE}_4(s_0, s_7)$]	**(iii)**	Compute $s_3 \mapsto \texttt{LOAD}_7(s_5)$	[..., PUSH(128),$\texttt{LOAD}_7$]
7	$[s_3, s_8, s_0, s_2, s_1, s_4]$	[..., $\texttt{LOAD}_7(s_5)$]	**(ii)**	Swap to position 4	[..., SWAP4]
8	$[s_1, s_8, s_0, s_2, s_3, s_4]$	[...]	**(i)**	Pop element	[..., POP]
9	$[s_8, s_0, s_2, s_3, s_4]$ $[s_0, s_8, s_2, s_3, s_4]$	[...] [...]	**(iii)**	Term: $\texttt{STORE}_{13}(s_0, s_8)$ Swap arguments Compute $\texttt{STORE}_{13}(s_0, s_8)$	 [..., SWAP1] [..., $\texttt{STORE}_{13}$]
10	$[s_2, s_3, s_4]$	[..., $\texttt{STORE}_{13}(s_0, s_8)$]			

stack elements to place the topmost element in its final position and we must choose another computation.

Another key aspect of Greedy is dealing with the limitation of the maximum stack depth that can be accessed. This issue is inherent to the design of the EVM instruction set and also occurs in EVM compilers [47]. It is mitigated at step **(iii)** with procedure *chooseNextElement*, which prioritizes computing those terms whose operands depend on values that are at the bottom of the stack when they are about to become unreachable. Using this strategy, our experimental evaluation in Sec. 5 only encountered stack-too-deep errors in 36 sequences out of more than 500,000. In addition, procedure *computeElement* needs to perform extra operations to ensure correctness. Any element in ini.stack needed to obtain v (including v itself, if in ini.stack) must be part of current and be duplicated the number of times given by $uses(\texttt{v}, \texttt{current}, \texttt{fin})$, since such elements cannot be computed. Similarly, every LOAD, once computed and added to current, is treated as an element of ini.stack, since recomputing it could contradict the dependencies. Hence, it should be duplicated when needed. The implementation for step **(iii)** combines heuristics based on the ideas above.

Example 3.1. Consider the execution of Greedy on the $\texttt{ini}_1$ and $\texttt{fin}_1$ states from Example 2.4. We assume deplist forces a strict order among accesses (see deplist(mem) in Example 2.3). Lines 5 and 6 initialize current.stack to $[s_0, s_1, s_2]$ and seqGrd to [], resp. Table 3 shows the evolution of the stacks of current and seqGrd during execution. The algorithm finishes with the state shown for I=10 and all memory operations done in the correct order. While the original sequence $\texttt{seq}_1$ had 16 opcodes, the greedy one seqGrd (displayed as $\texttt{seqGrd}_1$ in Table 2) has 14.

3.2 Implementation of Wasm Greedy in SuperStack

In the context of Wasm, Greedy aims to reduce the number of instructions in a sequence by replacing SETx and GETx by TEEx or by avoiding storing intermediate computations in registers. These situations typically arise when a term v is computed and stored in a register using SETx, even though it will be used later. These optimizations become complex when v has dependencies with other instructions that must be satisfied before v can be used. Another crucial aspect affecting the

Table 4. Execution of Wasm Greedy for $\texttt{ini}_2$ and $\texttt{fin}_2$ in Example 2.4. It follows the same format as Table 3.

I	current.stack	current.regs	current.mem	Case	Action	seqGrd
1	[ ]	$[s_0, s_1, s_2]$	[ ]	**(iii)**	Compute $s_3 \mapsto \texttt{LOAD}_0^*$	$[\texttt{LOAD}_0^*]$
2	$[s_3]$	$[s_0, s_1, s_2]$	$[\texttt{LOAD}_0^*]$	**(iii)**	Term: $s_8 \mapsto \texttt{I32.SHL}(s_7, s_6)$ Subterm: $s_6 \mapsto \texttt{I32.REM}(s_1, s_3)$ Retrieve s_1	[..., GET1]
	$[s_1, s_3]$	$[s_0, s_1, s_2]$	[...]		Compute $s_6 \mapsto \texttt{I32.REM}(s_1, s_3)$	[..., I32.REM]
	$[s_6]$	$[s_0, s_1, s_2]$	[...]		Compute $s_7 \mapsto \texttt{PUSH(2)}$	[..., PUSH(2)]
	$[s_7, s_6]$	$[s_0, s_1, s_2]$	[...]		Compute $s_8 \mapsto \texttt{I32.SHL}(s_7, s_6)$	[..., I32.SHL]
3	$[s_8]$	$[s_0, s_1, s_2]$	[...]	**(ii)**	Place s_8 in reg. 0 with SET0	[..., SET0]
4	[]	$[s_8, s_1, s_2]$	[...]	**(iii)**	Term: $\texttt{STORE}_4(s_5, s_4)$ Compute $s_4 \mapsto \texttt{PUSH(0)}$	[..., PUSH(0)]
	$[s_4]$	$[s_8, s_1, s_2]$	[...]		Compute $s_5 \mapsto \texttt{PUSH(1)}$	[..., PUSH(1)]
	$[s_5, s_4]$	$[s_8, s_1, s_2]$	[...]		Place s_5 in reg. 2 with TEE2	[..., TEE2]
	$[s_5, s_4]$	$[s_8, s_1, s_5]$	[...]		Compute $\texttt{STORE}_4(s_5, s_4)$	[..., $\texttt{STORE}_4$]
5	[ ]	$[s_8, s_1, s_5]$	[..., $\texttt{STORE}_4(s_5, s_4)$]			

design of GREEDY is that Wasm uses the operational stack for direct computation or returning values in a function. Thus, ini.stack is usually empty, and fin.stack contains at most one element (to be returned or checked in a conditional jump). As a result, our main focus is to compute fin.regs and fin.mem, while the selection of elements in fin.stack is delayed in *chooseNextElement*.

The key for the Wasm implementation is to only apply step **(ii)** when current.stack[0] is in register x in fin.regs and x is *free*, i.e. its value is not needed (formally, current.regs[x] $\neq$ fin.regs[x] and $uses(\texttt{current.regs[x]}, \texttt{current}, \texttt{fin}) = 0$), which is checked by procedure *canBePlaced*. Otherwise, we move to step **(iii)** where *chooseNextElement* prioritizes selecting elements that use current.stack[0] and, as said, considers first elements in fin.regs and fin.mem. Additionally, once the element is chosen, *computeElement* checks if there is some order of operands in commutative terms that could use the topmost element directly. It also examines if v uses a stack element in current.stack with depth > 1. If so, it accesses the corresponding element by placing the elements in-between in (auxiliary) registers. On top of that, *computeElement* stores any intermediate stack element s_i in a register immediately after its computation if it has not been previously stored in a register and must be used elsewhere (i.e. $uses(s_i, \texttt{current}, \texttt{fin}) > 1$). In those cases, a TEEx instruction is always applied to maintain s_i in the stack for the computation of v. Lastly, *solvePermutation* places any element in the stack that should be in a register directly, since all such registers must be free now, and reorders the stack if needed.

Example 3.2. Consider the execution of GREEDY on the $\texttt{ini}_2$ and $\texttt{fin}_2$ states from Example 2.4. As discussed in Ex. 2.7, we assume deplist = $[\texttt{LOAD}_0^* < \texttt{STORE}_4(s_5, s_4)]$. Table 4 depicts the execution of the greedy algorithm. The algorithm finishes in state I=5, as there is no need to reorder values with *solvePermutation*. The algorithm returns $\texttt{seqGrd}_2$ in Table 2 which manages to save 3 instructions: SET2 and GET2 are saved by avoiding storing the result of $\texttt{LOAD}_0^*$ in an intermediate register; the second time PUSH(1) is computed in the initial sequence is saved thanks to TEE2 both updating the corresponding register in fin.regs and placing the element in the stack to be used afterwards.

4 TACKLING THE SYNTHESIS PROBLEM WITH SAT

While state-of-the-art methods in bytecode superoptimization [2, 32, 44, 46] typically use SMT engines [2, 3], the structure of the problem naturally fits propositional satisfiability (SAT) reasoning. Indeed, SAT is known to be a good choice to tackle well-defined and constrained synthesis problems [21, 25, 30, 39, 48, 67], as the required decisions are intrinsically Boolean. For example, a Boolean variable can represent whether a specific opcode is applied at a given step or not; and whether a specific value is stored in a stack cell at a given step. Moreover, the transition step from one

symbolic state to another, which is defined by the semantics of the opcode applied at that step, can be easily encoded using Boolean logic connectives. This suitability has been recognized by a number of recent successful applications of SAT and MaxSAT to the efficient synthesis of *minimal* representations of systems as well as planning problems [16, 21, 25, 26, 29, 30, 39, 40, 48, 52, 59, 60, 67]. The main differences between our SAT approach and existing SMT encodings of the problem (e.g., [4, 32, 46]) are: (1) a redefinition of the problem using the (SAT) encoding described in Sec. 4.1 which incorporates in its definition dependencies among operations not considered in the previous frameworks, (2) a new objective function for length minimization proposed in Sec. 4.2 which turns out to be very relevant in practice, and (3) new dominance and redundant constraints proposed in Sec. 4.3 and exploited by our implementation.

This section details the procedure SearchSAT called in line 10 of Alg. 1 with the following input arguments: a flag c indicating which dominance and redundant constraints should be enabled; the selected objective function f; the initial state ini; the final state fin obtained by executing $\texttt{ini} \rightarrow^*_{\texttt{seq}}$ and which has associated sets fin.map, fin.Terms, and fin.StackElems; an upper bound $\texttt{s}^{max}$ on the length of the stack needed to execute an optimal sequence; an upper bound $\texttt{r}^{max}$ on the number of registers required; an upper bound $\texttt{l}^{max} = min(\texttt{g}^{max},\texttt{n}^{max})$ on the length of an optimal sequence (where $\texttt{g}^{max}$ is the length of the solution found by the greedy algorithm); and the optimal sequence found by the greedy algorithm seqGrd. SearchSAT returns a solution seqSAT if found, and a value isOptimal indicating whether seqSAT has been proven optimal. The value of $\texttt{l}^{max}$ significantly affects the encoding size and, hence, the difficulty of the SAT oracle calls. Note that since the greedy solution seqGrd is fed to the SAT solver, it does not have to find *a* solution of the given size. In the remainder of the section we first overview the Boolean variables used in the propositional encoding of the problem as well as exemplify how the high-level constraints can be translated to the *clausal* level of formulas in CNF. Second, we demonstrate how the objective function f can be naturally represented as a set of weighted soft clauses, thus reducing the synthesis problem to weighted partial MaxSAT, and we explain how program length minimization can be efficiently handled with the use of iterative SAT solving. Lastly, we present the dominance and redundant constraints.

4.1 Propositional Encoding

We use the standard definitions in *propositional satisfiability* (SAT) and *maximum satisfiability* (MaxSAT) solving [7, 41]. SAT and MaxSAT formulas are assumed to be propositional. Propositional formula φ is in *conjunctive normal form* (CNF) if it is a conjunction of clauses, where a *clause* is a disjunction of literals, and a *literal* is either a Boolean variable b or its *negation* $\neg b$. Whenever convenient, a clause is treated as a set of literals. A *truth assignment* μ is a mapping from the set of variables in φ to $\{0, 1\}$. A clause is *satisfied* by truth assignment μ if μ assigns value 1 to one of its literals; and *falsified* otherwise. If all clauses of formula φ are satisfied by assignment μ then μ also satisfies φ; otherwise, φ is falsified by μ. Formula φ is *satisfiable* if there is an assignment μ satisfying φ; otherwise, φ is *unsatisfiable*. In the context of unsatisfiable formulas, the maximum satisfiability problem is to find a truth assignment that maximizes the number of satisfied clauses. We use a variant of MaxSAT called Partial (Weighted) MaxSAT [7]. Formula φ in MaxSAT is a conjunction of *hard* clauses $\mathcal{H}$, which must be satisfied, and *soft* clauses $\mathcal{S}$, each with a numeric weight representing a preference to satisfy it, i.e. $\varphi = \mathcal{H} \wedge \mathcal{S}$. When convenient, soft clause c with weight w is denoted by (c, w). The aim is to find a truth assignment that satisfies $\mathcal{H}$ while maximizing the total weight of satisfied soft clauses.

Let $\texttt{OP} = \{op(\texttt{term}) | \texttt{term} \in \texttt{fin.Terms}\} \cup \{\texttt{POP}\} \cup \{\texttt{SWAP}x \mid 1 \leq x \leq \texttt{s}^{max} - 1\} \cup \{\texttt{DUP}x \mid 1 \leq x \leq \texttt{s}^{max}\} \cup \{\texttt{GET}x, \texttt{SET}x, \texttt{TEE}x \mid 0 \leq x \leq \texttt{r}^{max} - 1\}$ be the set of all opcodes available (op returns the opcode associated to term). We add the NOP instruction to the encoding to allow for some steps in a sequence not to have an opcode. This is needed for building solutions shorter than $\texttt{l}^{max}$: those with

NOPs at the end. For every symbolic execution step $i \in \{1, \ldots, 1^{max}\}$, we introduce Boolean variables $x_{io}, o \in \{0, 1, \ldots, |\texttt{OP}|\}$ with $o = 0$ reserved for NOP and $1 \leq o \leq |\texttt{fin.Terms}|$ reserved for the opcodes of fin.Terms, to indicate whether the opcode with the corresponding o-index is applied at step i ($x_{io} = 1$) or not ($x_{io} = 0$). As a slight abuse of notation, we will use $x_{io_{\texttt{OPCODE}}}$ to represent the fact that a specific OPCODE is applied at step i and, thus, appears at position i in opt. We also introduce Boolean variable $y_{ijv}, i \in \{0, 1, \ldots, 1^{max}\}$, $j \in \{0, \ldots, \texttt{s}^{max} - 1\}$, and $v \in \{0, 1, \ldots, |\texttt{fin.StackElems}|\}$ with $v = 0$ reserved for value nil, to indicate whether the j'th position of the stack at step i of the symbolic execution (where $i = 0$ corresponds to ini.stack) contains the value represented by index v ($y_{ijv} = 1$) or not ($y_{ijv} = 0$). By an abuse of notation, we also denote the nil value by $v_{\texttt{nil}}$. We introduce Boolean variable w_{irv}, $i \in \{0, 1, \ldots, 1^{max}\}$, $r \in \{0, 1, \ldots, \texttt{r}^{max} - 1\}$, and $v \in \{0, 1, \ldots, |\texttt{fin.StackElems}|\}$ to represent the value v stored in register r after step i. We use x_{io}, y_{ijv} and w_{irv} to define the following constraints:

$$\sum_{o=0}^{|\texttt{OP}|} x_{io} = 1, \quad i \in \{1, \ldots, 1^{max}\} \tag{1}$$

$$\sum_{v=0}^{|\texttt{fin.StackElems}|} y_{ijv} = 1, \quad i \in \{0, \ldots, 1^{max}\}, j \in \{0, \ldots, \texttt{s}^{max} - 1\} \tag{2}$$

$$\sum_{v=0}^{|\texttt{fin.StackElems}|} w_{irv} = 1, \quad i \in \{0, \ldots, 1^{max}\}, r \in \{0, \ldots, \texttt{r}^{max} - 1\} \tag{3}$$

which ensure each step i of the symbolic execution uses exactly one available opcode (Eq.1) – the one appearing at position i of opt; and each cell j of the stack and each register at each step i contains exactly one possible value v (Eqs.2–3). Note that these constraints are easy to represent in CNF by applying any of the well-known encodings of cardinality constraints [5, 8, 20, 50, 54, 58]. We ensure the y_{ijv} variables correctly represent the values in ini.stack and fin.stack as follows:

$$y_{0j\texttt{ini.stack}[j]}{=}1, \quad 0 \leq j < |\texttt{ini.stack}| \quad \wedge \quad y_{0j0}{=}1, \quad |\texttt{ini.stack}| \leq j < \texttt{s}^{max}$$

$$y_{1^{max}j\texttt{fin.stack}[j]}{=}1, \quad 0 \leq j < |\texttt{fin.stack}| \quad \wedge \quad y_{1^{max}j0}{=}1, \quad |\texttt{fin.stack}| \leq j < \texttt{s}^{max}$$

Note we use nil ($v = 0$) as filler once all stack values are enforced. The operational semantics of each opcode is encoded in CNF by, for every step (stack,regs, mem)$\rightarrow^{i}_{\texttt{OPCODE}}$ (stack',regs',mem'), imposing the appropriate constraints. For example, NOP is encoded as:

$$x_{io_{\texttt{NOP}}} \rightarrow (y_{i-1jv} \leftrightarrow y_{ijv}), i \in \{0, \ldots, 1^{max} - 1\}, j \in \{0, \ldots, \texttt{s}^{max} - 1\}, v \in \{0, \ldots |\texttt{fin.StackElems}|\}$$

$$x_{io_{\texttt{NOP}}} \rightarrow (w_{i-1rv} \leftrightarrow w_{irv}), i \in \{0, \ldots, 1^{max} - 1\}, r \in \{0, \ldots, \texttt{r}^{max} - 1\}, v \in \{0, \ldots |\texttt{fin.StackElems}|\}$$

which enforces stack' and regs' to be the same as stack and regs whenever OPCODE is NOP. Since we only want NOPs to occur at the end of the optimized sequence, we add the clauses:

$$x_{io_{\texttt{NOP}}} \rightarrow x_{i+1o_{\texttt{NOP}}}, \quad i \in \{1, \ldots, 1^{max} - 1\} \tag{4}$$

As only SETx and TEEx modify the registers, the encoding of any other instruction includes the second set of clauses for NOP (changing the first literal), thus keeping the registers unchanged. Similarly, POP is encoded by the following clauses (plus those needed to keep registers unchanged):

$$x_{io_{\texttt{POP}}} \rightarrow \left(y_{i-1jv} \leftrightarrow y_{ij-1v}\right), i \in \{1, \ldots, 1^{max}\}, j \in \{1, \ldots, \texttt{s}^{max} - 1\}, v \in \{0, \ldots |\texttt{fin.StackElems}|\}$$

$$x_{io_{\texttt{POP}}} \rightarrow y_{i\texttt{s}^{max}-1v_{\texttt{nil}}}, i \in \{1, \ldots, 1^{max}\}$$

which enforces stack' to have all values of stack shifted by one towards the topmost (thus removing the topmost in stack), and nil at the bottom. As yet another example, SETx can be encoded as a copy of the POP stack manipulation clauses together with:

$$x_{io_{\texttt{SET}x}} \rightarrow (y_{i-10v} \leftrightarrow w_{ixv}), i \in \{1, \ldots, 1^{max}\}, v \in \{0, \ldots |\texttt{fin.StackElems}|\}$$

$$x_{io_{\texttt{SET}x}} \rightarrow (w_{i-1rv} \leftrightarrow w_{irv}), i{\in}\{1, \ldots, 1^{max}\}, r{\in}\{0, 1, \ldots, \texttt{r}^{max} - 1\}\backslash\{x\}, v{\in}\{0, \ldots |\texttt{fin.StackElems}|\}$$

As a final example, the PUSH(k) stack manipulation operations are encoded as:

$$x_{io_{\texttt{PUSH}}} \rightarrow \left(y_{i-1jv} \leftrightarrow y_{ij+1v}\right), i \in \{1, \ldots, 1^{max}\}, j \in \{0, \ldots, \texttt{s}^{max} - 2\}, v{\in}\{0, \ldots |\texttt{fin.StackElems}|\}$$

$$x_{io_{\texttt{PUSH}}} \rightarrow y_{i0v}, \text{where } \texttt{fin.map}(s_v) = k, \text{i.e. the element created by } \texttt{PUSH}(k)$$

All other opcodes can be encoded similarly from their operational semantics. If the clauses encoding the specification of the problem are aggregated into a CNF formula φ, a SAT solver can be used to decide φ's satisfiability and compute an assignment satisfying φ. Reconstructing opt and the associated symbolic states from this assignment and variables x_{io}, y_{ijv} and w_{irv} is straightforward.

Example 4.1. The optimal sequences $\texttt{seqSAT}_1$ and $\texttt{seqSAT}_2$, resp., found by procedure SearchSAT described in this section for $\texttt{seq}_1$ and $\texttt{seq}_2$ appear in Table 2. Note that the SAT search has managed to reduce the solution found by Greedy ($\texttt{seqGrd}_1$) to 13 by cleverly using a single swap operation to both place s_3 in its final position and leave on top of the stack the right arguments for the final $\texttt{STORE}_{13}(s_0, s_8)$ computation. For $\texttt{seq}_2$, it finds a different solution than Greedy ($\texttt{seqGrd}_2$) but of the same length since this length is already optimal.

4.2 Objective Functions

The standard way to minimize a cost function is to add a set of *soft clauses* that represent our preferences to satisfy them. Let us assume each available opcode with index $o \in \{1, \ldots, |\texttt{OP}|\}$ has associated cost $w_o \in \mathbb{R}_{>0}$. The objective is to minimize cost $\sum_{i \in \{1,\ldots,1^{max}\}, o \in \{1,\ldots,|\texttt{OP}|\}} w_o \cdot x_{io}$ subject to φ constructed as described above. This is reduced to a MaxSAT problem with hard clauses φ and weighted soft clauses $\{(\neg x_{io}, w_o) \mid i \in \{1, \ldots, 1^{max}\}, o \in \{1, \ldots, |\texttt{OP}|\}\}$. Any off-the-shelf MaxSAT solver can be applied to this MaxSAT formulation to get a sequence of opcodes satisfying the specification and minimizing the cost function.

The objective function above contains $1^{max} \times |\texttt{OP}|$ weighted components, which can make the MaxSAT call quite expensive in practice. While this may be unavoidable in general, it is for program length minimization, where $w_o = 1$ for all o. Namely, instead of stating a preference for opcodes $o \in \{1, \ldots, |\texttt{OP}|\}$ not to be used at each step $i \in \{1, \ldots, 1^{max}\}$ (causing it to minimize their use when possible), we simply state the preference to apply NOP at step i. This can be seen as maximizing the objective function $\sum_{i \in \{1,\ldots,1^{max}\}} x_{io_{\texttt{NOP}}}$, which can be represented as a set of unweighted unit soft clauses $\{(x_{io_{\texttt{NOP}}}) \mid i \in \{1, \ldots, 1^{max}\}\}$. Further, thanks to the NOP propagation constraints (4), we avoid using a full-blown MaxSAT solver and instead use a series of pure-SAT oracle calls deciding the satisfiability of $\varphi \wedge (x_{io_{\texttt{NOP}}})$ with the gradually decreasing value of step i while the oracle reports the formula to be satisfiable. If the oracle reports satisfiability for step i^* but not for i^*-1, the optimal problem length is proven to be i^*.

4.3 Dominance and Redundant Constraints

Let c be a constraint, C a set of constraints interpreted as their conjunction, $sol(C)$ the set of solutions of C, and f an objective function over C's variables. Constraint c is *redundant* for C if $sols(C) = sols(C \cup \{c\})$. Redundant constraints will speedup the search if they help solvers detect failure earlier. Constraint c is said to be a *dominance constraint* for C and f if adding c to C only removes solutions whose objective value is equal or worse than that of others not eliminated, i.e., $\forall s \in sols(C) \setminus sols(C \cup \{c\}), \exists s' \in sols(C \cup \{c\})$ s.t. $f(s)$ is equal or worse than $f(s')$. Dominance constraints will speedup the search if they help solvers eliminate search areas that do not lead to better solutions. The following constraints for stack-bytecode already appear in related work:

- **a.** POP must be preceded by an instruction that creates no fresh stack element [3];
- **b.** Every expensive *op*(term), with term ∈ fin.Terms, must appear at most once in seqSAT [3];
- **c.** Every *op*(term), with term ∈ fin.Terms, must appear at least once in seqSAT [4];
- **d.** Opcodes creating fresh stack elements that appear as arguments of terms in fin.Terms must be placed earlier than the opcodes of those terms [2].

Since **a** is a dominance constraint while **c** and **d** are redundant, they all ensure optimality, i.e., guarantee an optimal solution is kept. While **b** does not, in practice the loss of optimality is

compensated by enabling the optimization of larger blocks (thanks to the extra pruning it achieves), thus having an overall beneficial effect [3]. All four are included in our implementation.

The following presents our four new constraints (parameter c in Alg. 1 enables/disables constraints a-h selectively). Their definition imposes constraint $\texttt{seqSAT}_j = op(\texttt{term})$, which holds if the opcode associated to $\texttt{term} \in \texttt{fin.Terms}$ is the opcode at step j in solution seqSAT. It also calls function $uses(s_i, \texttt{ini}, \texttt{fin})$ defined in Sec. 3, to count how many times stack element $s_i \in \texttt{fin.StackElems}$ is needed to compute fin from ini.

e. POP unused stack tops (non-optimal): If ini.stack contains elements that are not in fin, pop them as soon as they are on top. Let UNUSED be the set of such stack elements, i.e., $\{s_i \mid s_i \in \texttt{ini.stack}, uses(s_i, \texttt{ini}, \texttt{fin}) = 0\}$. The constraint is: $\bigwedge_{0 \le j \le length-1} \texttt{stack}_j[0] \in \texttt{UNUSED} \Rightarrow \texttt{seqSAT}_{j+1} = \texttt{POP}$, where $\texttt{stack}_j$ denotes the contents of the stack after the j'th operation. This is optimal unless the unused element can be swapped with others to get them into a better position, before being popped.

f. Create arguments of unary terms exactly where needed (optimal under certain conditions): As some elements are created *de novo* by terms with nullary opcodes, e.g. PUSH, we can insert them in the stack wherever needed if their cost is reasonably low.[1] We denote the set of fresh stack elements that are used exactly once by $\texttt{SINGLE} = \{s_j \mid s_j \in \texttt{fin.StackElems}, k \le j \le m, uses(s_j, \texttt{ini}, \texttt{fin}) = 1\}$, and the set of elements created by a nullary opcode cheaply by CHEAP0. If the argument s_i of unary term $\texttt{UF}(s_i) \in \texttt{fin.Terms}$ is only used once, or can be created cheaply with a nullary opcode, then we force its opcode (i.e., $op(\texttt{fin.map}(s_i))$) to appear directly before that of the unary operation (i.e., $op(\texttt{UF}(s_i))$). The constraint is: $\bigwedge_{1 \le j \le length} \texttt{seqSAT}_j = op(\texttt{UF}(s_i)) \wedge s_i \in \texttt{SINGLE} \cup \texttt{CHEAP0} \Rightarrow \texttt{seqSAT}_{j-1} = op(\texttt{fin.map}(s_i))$. Optimality is preserved if $s_i \in \texttt{SINGLE}$, as this would then be a dominance constraint.

g. Create arguments of binary terms exactly where needed (non-optimal): If the first argument s_0 of a binary term $\texttt{UF}(s_0, s_1) \in \texttt{fin.Terms}$, or the second if UF is commutative, is used only once or can be created cheaply with a nullary operation, we similarly force the creating opcode to appear directly before the binary operation. The constraint is defined as:

$$\bigwedge_{2 \le j \le length} \texttt{seqSAT}_j = op(\texttt{UF}(s_0, s_1)) \wedge s_0 \in \texttt{SINGLE} \cup \texttt{CHEAP0} \Rightarrow \texttt{seqSAT}_{j-1} = op(\texttt{fin.map}(s_0))$$

$$\bigwedge_{2 \le j \le length} \texttt{seqSAT}_j = op(\texttt{UF}(s_0, s_1)) \wedge commutative(\texttt{UF}) \wedge s_1 \in \texttt{SINGLE} \cup \texttt{CHEAP0} \wedge s_0 \notin \texttt{SINGLE} \cup \texttt{CHEAP0} \Rightarrow \texttt{seqSAT}_{j-1} = op(\texttt{fin.map}(s_1))$$

Note that if both $s_0, s_1 \in \texttt{SINGLE} \cup \texttt{CHEAP0}$, only s_0 is created immediately before the binary operation. We can do better by working with the *two* positions immediately before the binary operation:

$$\bigwedge_{3 \le j \le length} \texttt{seqSAT}_j = op(\texttt{UF}(s_0, s_1)) \wedge commutative(\texttt{UF}) \wedge s_0 \in \texttt{SINGLE} \wedge s_1 \in \texttt{SINGLE} \Rightarrow \texttt{seqSAT}_{j-1} \in \{op(\texttt{fin.map}(s_0)), op(\texttt{fin.map}(s_1))\}$$

$$\bigwedge_{3 \le j \le length} \texttt{seqSAT}_j = op(\texttt{UF}(s_0, s_1)) \wedge s_0 \in \texttt{CHEAP0} \wedge s_1 \in \texttt{CHEAP0} \Rightarrow \texttt{seqSAT}_{j-1} = op(\texttt{fin.map}(s_0)) \wedge \texttt{seqSAT}_{j-2} = op(\texttt{fin.map}(s_1))$$

This constraint is non-optimal for the same reason as **e** and because it enforces an order on the creation of the arguments of the binary operation.

h. Propagate the stack elements for predecessor and successor stacks (optimal): Each operation can add σ elements to the top of the stack or remove δ elements from it, where δ is the max arity of any opcode and σ is the maximum number of stack elements an opcode can introduce. Note that σ might be > 0 for call instructions in Wasm. Thus, any element in position $j > 0$ of the current stack must come from positions $j - \sigma, \ldots, j - 1, j, j + 1, \ldots, j + \delta$ or 0 (by SWAP operation) in the previous stack (ensuring they must be > 0). Similarly, any element in position $j > \delta$ of the current stack will end up in position $j - \delta, \ldots, j - 1, j, j + 1, \ldots, j + \sigma$ or 0 in the next stack.

[1] For the EVM objective size-in-bytes, if PUSH(i) is expensive it is better to use it once and then duplicate its value.

Importantly, most of the dominance and redundant constraints discussed above are expressed in CNF either directly as clauses with no auxiliary variables, or as cardinality constraints. The constraints enforcing each opcode from `fin.Terms` to be used at most (**b**) and at least once (**c**) are encodable as cardinality constraints $\sum_{i=1}^{1^{max}} x_{io} = 1, o \in \{1, \ldots, |\texttt{fin.Terms}|\}$, similar to (1) and (2). This may require the introduction of auxiliary variables, depending on the cardinality encoding to use. The only constraint that requires the explicit introduction of additional variables is the one enforcing precedences (**d**). For this, we introduce the variables z_{io}, such that $z_{io} = 1$ iff opcode o has been *applied before or at* symbolic execution step i, as follows:

$$\begin{array}{ll} z_{1o} \leftrightarrow x_{1o}, & o \in \{1, \ldots, |\texttt{OP}|\} \\ z_{io} \leftrightarrow (x_{io} \vee z_{i-1o}), & o \in \{1, \ldots, |\texttt{OP}|\}, i \in \{2, \ldots, 1^{max}\} \end{array}$$

Then, we can easily express a dependency that requires opcode o' to precede opcode o as:

$$z_{io} \rightarrow z_{i-1o'}, \quad i \in \{2, \ldots, 1^{max}\}.$$

5 EXPERIMENTAL EVALUATION

SuperStack can optimize a Wasm or EVM bytecode program b to reduce its length if b is written in Wasm, and to reduce its length, gas, or size-in-bytes if b is in EVM. Its implementation follows Alg. 1 and includes components for *sequences*, *symbolic* and Greedy, all programmed in Python. For SearchSAT it makes incremental use of the Glucose 3.0 SAT solver [6] interfaced through the well-known PySAT toolkit [27]. This choice is motivated by the incrementality features of Glucose 3.0 and the fact they are well supported in PySAT, in contrast to some of the more recent SAT solvers like Kissat [12] or Intel(R) SAT Solver [45]. For the objective functions, it uses either the RC2 MaxSAT solver [28] or a series of pure-SAT calls. This section reports on its experimental evaluation and the comparison with competing systems: `GASOL` for EVM [2] and the WebAssembly superoptimizer [15] (abbreviated as `wsouper`). Our experimental evaluation aims at answering the following questions:

Q1: Which are the gains and time savings obtained by the greedy algorithm?
Q2: Which are the gains and time savings obtained by the dominance and redundant constraints?
Q3: What is the effectiveness and efficiency of SuperStack compared to `GASOL` and `wsouper`?
Q4: How does our SAT approach scale? Is it efficient enough to be used in a compiler?
Q5: What is the real impact of the optimization achieved?

To answer them we considered five sets of benchmarks: (1) For a fair comparison with `wsouper`, we use the benchmarks set and the results from their paper (no optimization times were given, marked as "-" in the table),[2] i.e., 11 C programs compiled to Wasm by first compiling them to LLVM using the Clang compiler with aggressive optimizations on, and then using the LLVM to Wasm backend to optimize and generate Wasm code. We have excluded the Baggage program because it was optimized using global analysis, namely dead code elimination, which is outside the scope of superoptimization (see Sec. 6). The number of sequences in this set is 947. (2) We also use a set of 47 programs[3] from the library of Circom [11], a DSL to create arithmetic circuits for zero-knowledge proofs, used by many projects in production. They are compiled into

Table 5. Distribution on the number of instructions per sequence and the total number of instructions, where Q_i denotes the i-th quartile.

Set	mean	std	min	Q_1	Q_2	Q_3	max	# Instructions
(1)	9.49	47.80	1	3	5	11	1458	10,680
(2)	26.90	15.95	1	10	40	40	40	5.05M
(3)	9.45	16.02	1	3	6	11	1370	151k
(4)	8.61	9.97	1	3	5	10	1286	5.17M
(5)	8.84	11.18	1	3	6	10	745	586k

[2]Source code available at https://github.com/ASSERT-KTH/slumps/tree/master/superoptimizer
[3]Source code available at https://github.com/iden3/circomlib/tree/master/test/circuits

Table 6. Experimental results and comparison to GASOL/Wasm-Souper (k in S_x denotes more than 1,000). Three SuperStack configurations are studied: (1) *Greedy* standalone optimizer; (2) *Basic SAT*, the Pure-SAT approach without constraints **e-h** in Sec. 4.3; and (3) *Enhanced SAT*, with all constraints. Each configuration showcases the optimization time in minutes (*Time*), the gains for the corresponding objective (*Gains*), the percentage over the total cost of the original code (*%Gains*), the number of sequences that reach the time limit with no solution (*#timeout*) and the avg. number of instructions in these sequences (*Seq sz*). Columns *Speedup* and *Boost* depict SuperStack's improvement over the competing systems as $\frac{\text{Time}_{\text{WG}}}{\text{Time}_{\text{SS}}}$ and $100 * \frac{\text{Gains}_{\text{SS}} - \text{Gains}_{\text{WG}}}{\text{Gains}_{\text{WG}}}$.

Benchmark Set	wsouper / GASOL (WG)					SuperStack (SS)						WG vs SS	
(Language/Criteria)	Time	Gains	%Gains	#timeout	Seq sz	Configuration	Time	Gains	%Gains	#timeout	Seq sz	Speedup	Boost
(1) wsouper's						Greedy	1	95	0.89	0	-	-	-14.41
benchmark	-	111	1.04	-	-	Basic SAT	164	119	1.11	9	204.44	-	7.21
(Wasm/length)						Enhanced SAT	97	121	1.13	3	512.33	-	9.01
(2) Circom						Greedy	12	737k	14.68	0	-	-	-
library	-	-	-	-	-	Basic SAT	47k	1,179k	23.47	7,301	40	-	-
(Wasm/length)						Enhanced SAT	4,752	1,218k	24.25	230	40	-	-
(3) GASOL's						Greedy	2	21k	0.24	0	-	604.5	94.05
benchmark	1,209	11k	0.12	1,143	35.47	Basic SAT	413	23k	0.28	295	43.97	2.93	129.72
(EVM/gas)						Enhanced SAT	212	26k	0.29	17	73.64	5.7	140
(4) 1,000 most						Greedy	150	777k	0.26	0	-	249.64	86.7
recent deployed	37,378	416k	0.14	36,316	32.96	Basic SAT	13,610	910k	0.3	10,279	41.13	2.75	118.7
(EVM/gas)						Enhanced SAT	6,089	947k	0.31	241	68.78	6.14	127.59
(5) 100 most						Greedy	11	214k	1.17	0	-	361.25	188.17
called	3,748	74k	0.41	3,923	34.1	Basic SAT	1,785	231k	1.26	1,770	41.8	2.1	210.57
(EVM/gas)						Enhanced SAT	961	234k	1.28	343	57.4	3.9	214.59
(3) GASOL's						Greedy	2	4k	2.09	0	-	663.5	9.8
benchmark	1,327	4k	1.91	1,141	35.87	Basic SAT	413	6k	3.09	295	43.97	2.98	61.78
(EVM/size)						Enhanced SAT	212	6k	2.86	17	73.64	6.26	50.08
(4) 1,000 most						Greedy	150	194k	2.48	0	-	277.87	22.95
recent deployed	41,605	158k	2.01	35,822	33.38	Basic SAT	13,610	269k	3.43	10,279	41.13	3.06	70.3
(EVM/size)						Enhanced SAT	6,089	253k	3.22	241	68.78	6.83	60.03
(5) 100 most						Greedy	11	50k	5.12	0	-	398.83	47.32
called	4,138	33k	3.47	3,883	34.3	Basic SAT	1,785	58k	5.97	1,770	41.8	2.32	71.77
(EVM/size)						Enhanced SAT	961	57k	5.88	343	57.4	4.3	69.24

Wasm by the Circom compiler which, as it was recently created does not incorporate aggressive optimizations such as those of Clang and the LLVM to Wasm backend. Since these sequences can have up to 15k opcodes, we split them in subsequences of at most 40, which we have experimentally determined as the upper limit of tractability for SearchSAT in Wasm. Thus, we work with a weaker notion of optimality by applying superoptimization to smaller sequences, see Sec. 6. The resulting set has 181,806 sequences. (3) For a fair comparison with GASOL, we use the benchmark set from its latest paper[4] [2], i.e., the last 30 verified smart contracts (downloaded using Etherscan [61]) compiled using some 0.8.x version of `solc` and whose source code was available as of June 21, 2021. The number of sequences in this set is 12,927. (4) To further increase our testbed, on the 14th of March 2023 we used Etherscan to download the 1,000 most recent deployed and verified contracts compiled using some 0.8.x version of `solc`, with source code in Solidity [63]. We then compiled them using version 0.8.19 of `solc` with flag *–optimize* on (default bytecode optimizer) to measure the extra optimizations gained by our approach. We have 476,974 sequences in this set. (5) We downloaded the deployed code of the 100 most-called contracts on Ethereum (hence the most relevant ones to be optimized) that were compiled with some version 0.8.x of solc, using BigQuery [22] and Etherscan. The number of sequences in this set is 51,682. Therefore, in total we have optimized 724,336 sequences. More information on the benchmark sets is provided in Table 5.

Table 6 details the results for the five sets, obtained using an AMD Ryzen Threadripper PRO 3995WX 64-cores and 512 GB of memory, running Debian 5.10.7. As the SAT solvers used are deterministic, the evaluation corresponds to a single execution, with multiple runs yielding similar times. To ensure a fair comparison, we established a time limit based on the competing systems. GASOL gives a larger timeout to larger sequences, while the experiments for `wsouper` in [15] assign

[4]Source code available at https://github.com/costa-group/gasol-optimizer/tree/main/examples/solidity

300 s per sequence. We use 300 s for set (1) and assign the same timeout as GASOL for the others. When the timeout is reached, the SAT solver might have been able to find a solution but optimality has not been proven, or it might have not found any solution. This solution found might be even optimal, but the solver has not been able to prove its optimality within the time limit. Moreover, for set (2) we have not been able to use wsouper since it can only be applied to C/C++ code through the compilation pipeline described above. Another relevant aspect of the evaluation is that configurations *Basic SAT* and *Enhanced SAT* use the pure-SAT based approach due to its superior performance for gas and bytes-size optimization compared to the MaxSAT approach (even though the pure-SAT approach only minimizes program length). It is important to note how gains and costs are determined for each objective function. While the gains for length and size-in-bytes can be easily computed, those for gas can only be approximated statically. We measure such gains as GASOL does: counting once the gas of each instruction in the program and using an average gas cost for storage instructions.

Q1 is answered by comparing *Speedup* and *Boost* columns in *Greedy* configurations, for sets 3-5. This shows reductions in search time by up to 3 orders of magnitude, and increases in gains by up to 3 times by the greedy algorithm on its own *wrt.* GASOL. **Q2** is answered by comparing configurations *Basic SAT* and *Enhanced SAT*, which shows the time decreases by half when using dominance/redundant constraints while the gains are roughly the same. Importantly, we have only found 82 sequences for EVM for which the objective value obtained with the new constraints is worse than without them and 0 for Wasm. For EVM, the loss of optimality is negligible and clearly compensated by their important time reduction. **Q3** is answered by comparing wsouper/GASOL with *Enhanced SAT*. This shows our proposal integrating all components significantly outperforms GASOL (sets 3-5) for both objective functions. In particular, for the largest sample (4), we improve GASOL savings by 127.59 % for gas and 60.03 % for size while reducing the overall time to more than one sixth (for both objectives). For Wasm, the gains for set (1) are not as large as for set (2) because the code in (1) is already highly optimized, not leaving much room for improvement. Still, we outperform wsouper by 9.01 % for set (1) and achieve great optimizations (over 24 % of length reduction, see %Gains for *Enhanced SAT*) for code which is not so optimized in set (2).

Q4 is answered by noting that GASOL generally timeouts when optimizing sequences of >30 instructions, while SuperStack doubles this threshold. This is a huge improvement because the time overhead in superoptimization grows exponentially when the sequence size is increased. In regards to its practical adoption, the results show that using all components proposed in this paper is indeed the best option for achieving optimality with the support of the greedy algorithm for large sequences. To estimate how this combination behaves, we have measured the overall gains and overhead for the largest dataset (4) by applying Greedy alone for sequences of >60 instructions. This subset consists of 1,344 sequences for which SearchSAT reaches #tout in 92.56 % of the cases and does not return any solution in 8.04 %. Greedy reduces the time spent optimizing these sequences by 95 % (from 1.836 min to 10 min), while maintaining 98.56 % of the gains for $(4)_g$ (from 60,681 to 59,809) and 97.05 % for $(4)_s$ (from 11,818 to 11,470). Overall, we argue both approaches (full approach, or only greedy for large sequences) could be included in a compiler, depending on the tradeoff between savings and overhead users want to achieve.

To answer **Q5** we focused on set (5), as these popular contracts manage hundred of thousands and even millions of transactions. We first downloaded all transactions from the set's 100 contracts as a block 17,226,487[5] using Etherscan's Python API [35], consisting of 41,106,276 transactions. Then we combined the transaction fee information with the price per Eth in dollars in the corresponding day (downloaded from [62]), resulting in $656.25M spent in transaction fees. Finally, we have

[5]Produced on 2023-05-10 12:36:23 AM +UTC

removed the costs inherently tied to the transactions (default 21,000 units of gas fee plus the fee for sending the transaction data), which results in a total of $509.1M of execution costs. Assuming these contracts were deployed using the setting *all* which has average gas savings 1.28 (data in S_{all} for set $(5)_g$), and that these gas savings affect all transactions uniformly, this translates to savings of $6.51M just on these 100 contracts, while the average optimization time is around 10 min, which is a reasonable time in the EVM context. Overall, we argue our experimental results prove the significant impact of our approach in the Ethereum ecosystem and the improved scalability *wrt.* previous SMT-based superoptimization. Although not experimentally evaluated by us here, it is known that optimizing Wasm also has significant impact on its performance (see, e.g., [23, 31, 37]).

6 RELATED WORK AND CONCLUSIONS

We have proposed a seamless integration of greedy, constraint-based and SAT techniques within the superoptimization framework that achieves optimality by minimizing the number of movements within the stack for EVM, and between registers and the stack for Wasm. Denali [33] pioneered the use of theorem proving and SAT solving as effective tools for superoptimization. The use of theorem proving focuses on finding equivalences among what we referred to as uninterpreted functions (e.g., arithmetic and bit-wise operations), while that of SAT focuses on considering common subexpressions and other specific features of the architecture. Note that Denali is designed for registered-based ALU operations rather than for a stack-based language. Thus, its propositional encoding is different from ours and considers neither our generic encoding of objective functions using soft clauses, nor our efficient approach to length minimization. The first to propose using an SMT solver was the unbounded superoptimization [32] approach for LLVM, later extended within the Souper superoptimizer [44, 55]. As the target architecture of the LLVM superoptimizers is not stack-based, their SMT encodings again differ significantly from ours. In particular, the representation and manipulation of the stack and the associated dominance constraints are not applicable to LLVM. However, some of our ideas developed for stack-manipulating opcodes could be adapted to the LLVM context, such as the use of a greedy algorithm to find tighter bounds for the length of the optimized code. We believe the gains could be very important as well. Another difference of our framework is that it handles uninterpreted functions in the symbolic analysis phase via peephole optimizations, instead of as part of the encoding. We only lose local opportunities in computations that are semantically equivalent to a cheaper computation but not captured by simplification rules. For instance, assume `PUSH` is encoded by 1 byte. Then, `PUSH(0x2540be40000000)` takes 8 bytes and is equivalent to `PUSH(0x10) PUSH(0x28) EXP` which takes 5. Our approach does not find this because constant `0x2540be40000000` in our symbolic state can only be computed using `PUSH`. Handling these (rarely applicable) optimizations is costly: ebso [46] encoded them and resulted in 80% timeouts (using 1 hour per sequence). The Souper LLVM superoptimizer has been applied to stack-bytecode (namely Wasm) by pipeline of transformations (and associated optimizations) from C/C++ to LLVM and then from LLVM to Wasm. This pipeline misses optimization opportunities that can be achieved using our approach. GreenThumb [51] is a framework for constructing superoptimizers in different ISAs, integrating multiple search techniques and proposing different approaches to scale existing superoptimization techniques. Notably, it introduces the LENS algorithm, which performs a bidirectional enumerative search with selective refinements. Integrating this algorithm into our framework is not feasible since we rely on internal search mechanisms within SAT solvers. However, GreenThumb also introduces a context-aware decomposition technique to scale the search which consists in randomly selecting fixed-length code fragments of the original program for optimization until no further improvement is possible. The idea is to provide the superoptimizer

with preconditions and postconditions for the optimized fragment's execution, that allow incorporating context information and relaxing the notion of correctness. This technique can be adapted to our approach by encoding the context information as part of the initial and final symbolic states.

Despite the above advances, scalability is still a limiting factor of superoptimization, since the search for optimality is exponential in the size of the solution being constructed and becomes intractable for complex code. A practical approach is to give up on optimality and move to more-efficient (possibly non-optimal) code, i.e., to make superoptimization incomplete. Different ways of introducing incompleteness are found in the literature: (1) using a timeout in the search, (2) splitting jump-free sequences into even smaller sequences, (3) using weaker notions of optimality (e.g., not handling memory), (4) using stochastic techniques [56] or machine learning [57] to prune the search process in a possibly incomplete way, and (5) using dominance constraints that can lose optimality in unusual situations but considerably accelerate the process. GASOL uses 1, 2, 3, and optionally 5; Souper uses 1 and 3 (as memory operations are not handled); GreenThumb uses 1, 2, 3 and 4; and SuperStack uses 1 and 3 (non-stack operations are uninterpreted), and optionally 5. We argue our experimental evaluation shows this combination leads to a practical while still very accurate approach. There are other techniques to improve the scalability based on reducing the enumerative search space, which cannot be adopted in our framework for the same reasons as for the LENS algorithm. For instance, [44] introduces a technique that combines different dataflow analyses to prune synthesis candidates before trying to instantiate them with specific values.

The generation of optimal code has been studied since the early days of stack machines [14]. The closest to our greedy algorithm would be the *stack scheduling* technique in [38] developed for the JVM. This technique aims at removing load accesses to registers by replacing them with stack duplication and manipulation opcodes, which differs from our optimization goal. It uses code analysis to detect pairs of program points in which load opcodes could be substituted, and then tries to perform as many replacements as possible. The only related idea to our algorithm is that it follows a greedy strategy to decide the order in which these pairs are traversed.

Superoptimization is complementary to other more traditional optimization techniques. In the context of smart contracts, the importance of optimizing them is noted in [13, 46, 49, 68] and guidelines for writing more efficient code have been provided for smart contract developers [17, 18, 34]. It is hence not surprising for optimization of EVM smart contracts to be an active research topic, and to find optimization approaches applied on the (Solidity) source-code level (e.g., [19]). EVM bytecode optimization is also performed by the standard Solidity compiler solc [64] which is able to perform certain types of inlining, of dead code elimination, etc. As regards to Wasm, it is an increasingly important low-level language that can be run in modern browsers and to which multiple source languages can be compiled. Wasm optimization is hence a highly relevant problem [23, 24, 53]. Recent work [37] explores missed optimizations in Wasm optimizers and identifies performance deficiencies [31]. The type of optimizations in [37] require global transformations and analysis; thus they are outside the scope of superoptimization. Summarizing, superoptimization is complementary to all mentioned approaches based on source-level and/or on global transformations (for both EVM and Wasm), and it is usually applied after having enabled them as a final code optimization stage, as we have done in our experiments.

As future work, we are exploring different strategies to enhance the performance of our approach such as the context-aware decomposition method mentioned earlier and more suitable formulas for setting the time limit based on the blocks obtained after partitioning. Additionally, we plan to apply the greedy algorithm without prior partitioning. This could uncover (with minimal computational cost) additional savings that may be missed due to partitioning. This strategy is supported by our findings in Sec. 5, where we demonstrated that, for larger sequences, the greedy algorithm alone achieves most gains consuming a fraction of the time spent by the SAT solver.

ACKNOWLEDGMENTS

Part of this research was done while Elvira Albert and Albert Rubio were visiting the University of Monash funded by the CM project S2018/TCS-4314. The work is also funded partially by the Spanish project PID2021-122830OB-C41 and the GREEN project AOC-1662 by the Ethereum Foundation.

DATA AVAILABILITY STATEMENT

SuperStack's source code and the benchmarks used for the experimental evaluation are available in Zenodo [1].

REFERENCES

[1] Elvira Albert, Maria Garcia de la Banda, Alejandro Hernández-Cerezo, Alexey Ignatiev, Albert Rubio, and Peter J. Stuckey. 2024. *Artifact for "SuperStack: Superoptimization of Stack-Bytecode via Greedy, Constraint-based, and SAT Techniques"*. https://doi.org/10.5281/zenodo.10801691

[2] Elvira Albert, Pablo Gordillo, Alejandro Hernández-Cerezo, and Albert Rubio. 2022. A Max-SMT Superoptimizer for EVM handling Memory and Storage. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 201–219. https://doi.org/10.1007/978-3-030-99524-9_11

[3] Elvira Albert, Pablo Gordillo, Alejandro Hernández-Cerezo, Albert Rubio, and Maria Anna Schett. 2022. Superoptimization of Smart Contracts. *ACM Trans. Softw. Eng. Methodol.* 31, 4 (2022), 70:1–70:29. https://doi.org/10.1145/3506800

[4] Elvira Albert, Pablo Gordillo, Albert Rubio, and Maria Anna Schett. 2020. Synthesis of Super-Optimized Smart Contracts Using Max-SMT. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12224)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 177–200. https://doi.org/10.1007/978-3-030-53288-8_10

[5] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. 2011. Cardinality Networks: a theoretical and empirical study. *Constraints* 16, 2 (2011), 195–221. https://doi.org/10.1007/s10601-010-9105-0

[6] Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. 2013. Improving Glucose for Incremental SAT Solving with Assumptions: Application to MUS Extraction. In *SAT (Lecture Notes in Computer Science, Vol. 7962)*. Springer, 309–317. https://doi.org/10.1007/978-3-642-39071-5_23

[7] Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. 2021. Maximum Satisfiabiliy. In *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, Vol. 336. IOS Press, 929–991. https://doi.org/10.3233/FAIA201008

[8] Olivier Bailleux and Yacine Boufkhad. 2003. Efficient CNF Encoding of Boolean Cardinality Constraints. In *CP*. 108–122. https://doi.org/10.1007/978-3-540-45193-8_8

[9] Sorav Bansal and Alex Aiken. 2006. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, John Paul Shen and Margaret Martonosi (Eds.). ACM, 394–403. https://doi.org/10.1145/1168857.1168906

[10] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. 2021. Satisfiability Modulo Theories. In *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, Vol. 336. IOS Press, 1267–1329. https://doi.org/10.3233/FAIA201017

[11] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina. 2023. Circom: A Circuit Description Language for Building Zero-Knowledge Applications. *IEEE Trans. Dependable Secur. Comput.* 20, 6 (2023), 4733–4751. https://doi.org/10.1109/TDSC.2022.3232813

[12] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximilian Heisinger. 2020. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *SAT Competition*. 51–53.

[13] Tamara Brandstätter, Stefan Schulte, Jürgen Cito, and Michael Borkowski. 2020. Characterizing Efficiency Optimizations in Solidity Smart Contracts. In *IEEE International Conference on Blockchain, Blockchain 2020, Rhodes, Greece, November 2-6, 2020*. IEEE, 281–290. https://doi.org/10.1109/Blockchain50366.2020.00042

[14] John L. Bruno and T. Lassagne. 1975. The Generation of Optimal Code for Stack Machines. *J. ACM* 22, 3 (1975), 382–396. https://doi.org/10.1145/321892.321901

[15] Javier Cabrera-Arteaga, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, and Martin Monperrus. 2020. Superoptimization of WebAssembly bytecode. In *Programming'20: 4th International Conference on the Art, Science, and Engineering of Programming, Porto, Portugal, March 23-26, 2020*, Ademar Aguiar, Shigeru Chiba, and Elisa Gonzalez Boix (Eds.). ACM, 36–40. https://doi.org/10.1145/3397537.3397567

[16] Martin Capek and Pavel Surynek. 2021. DPLL(MAPF): an Integration of Multi-Agent Path Finding and SAT Solving Technologies. In *SOCS*. AAAI Press, 153–155. https://doi.org/10.1609/SOCS.V12I1.18567
[17] Ting Chen, Youzheng Feng, Zihao Li, Hao Zhou, Xiapu Luo, Xiaoqi Li, Xiuzhuo Xiao, Jiachi Chen, and Xiaosong Zhang. 2021. GasChecker: Scalable Analysis for Discovering Gas-Inefficient Smart Contracts. *IEEE Trans. Emerg. Top. Comput.* 9, 3 (2021), 1433–1448. https://doi.org/10.1109/TETC.2020.2979019
[18] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *SANER*. IEEE Computer Society, 442–446. https://doi.org/10.1109/SANER.2017.7884650
[19] Yanju Chen, Yuepeng Wang, Maruth Goyal, James Dong, Yu Feng, and Isil Dillig. 2022. Synthesis-powered optimization of smart contracts via data type refactoring. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 560–588. https://doi.org/10.1145/3563308
[20] Niklas Eén and Niklas Sörensson. 2006. Translating Pseudo-Boolean Constraints into SAT. *JSAT* 2, 1-4 (2006), 1–26. https://doi.org/10.3233/SAT190014
[21] Bishwamittra Ghosh and Kuldeep S. Meel. 2019. IMLI: An Incremental Framework for MaxSAT-Based Learning of Interpretable Classification Rules. In *AIES*. ACM, 203–210. https://doi.org/10.1145/3306618.3314283
[22] Google. 2023. BigQuery. https://cloud.google.com/bigquery.
[23] WebAssembly Group. 2017. Binaryen Optimizations. https://github.com/WebAssembly/binaryen#binaryen-optimizations.
[24] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 185–200. https://doi.org/10.1145/3062341.3062363
[25] Marijn Heule and Sicco Verwer. 2010. Exact DFA Identification Using SAT Solvers. In *ICGI (Lecture Notes in Computer Science, Vol. 6339)*. Springer, 66–79. https://doi.org/10.1007/978-3-642-15488-1_7
[26] Alexey Ignatiev, Edward Lam, Peter J. Stuckey, and Joao Marques-Silva. 2021. A Scalable Two Stage Approach to Computing Optimal Decision Sets. In *AAAI*. AAAI Press, 3806–3814. https://doi.org/10.1609/AAAI.V35I5.16498
[27] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. 2018. PySAT: A Python Toolkit for Prototyping with SAT Oracles. In *SAT (Lecture Notes in Computer Science, Vol. 10929)*. Springer, 428–437. https://doi.org/10.1007/978-3-319-94144-8_26
[28] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. 2019. RC2: an Efficient MaxSAT Solver. *J. Satisf. Boolean Model. Comput.* 11, 1 (2019), 53–64. https://doi.org/10.3233/SAT190116
[29] Alexey Ignatiev, Filipe Pereira, Nina Narodytska, and Joao Marques-Silva. 2018. A SAT-Based Approach to Learn Explainable Decision Sets. In *IJCAR (Lecture Notes in Computer Science, Vol. 10900)*. Springer, 627–645. https://doi.org/10.1007/978-3-319-94205-6_41
[30] Alexey Ignatiev, Alessandro Previti, and Joao Marques-Silva. 2015. SAT-Based Formula Simplification. In *SAT (Lecture Notes in Computer Science, Vol. 9340)*. Springer, 287–298. https://doi.org/10.1007/978-3-319-24318-4_21
[31] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 2019. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. *login Usenix Mag.* 44, 3 (2019). https://www.usenix.org/publications/login/fall2019/jangda
[32] Abhinav Jangda and Greta Yorsh. 2017. Unbounded superoptimization. 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, October 23 - 27, 2017*. 78–88. https://doi.org/10.1145/3133850.3133856
[33] Rajeev Joshi, Greg Nelson, and Yunhong Zhou. 2006. Denali: A practical algorithm for generating optimal code. *ACM Trans. Program. Lang. Syst.* 28, 6 (2006), 967–989. https://doi.org/10.1145/1186632.1186633
[34] Queping Kong, Zi-Yan Wang, Yuan Huang, Xiangping Chen, Xiao-Cong Zhou, Zibin Zheng, and Gang Huang. 2022. Characterizing and Detecting Gas-Inefficient Patterns in Smart Contracts. *J. Comput. Sci. Technol.* 37, 1 (2022), 67–82. https://doi.org/10.1007/s11390-021-1674-4
[35] P.C. Kotsias. 2020. pcko1/etherscan-python. https://doi.org/10.5281/zenodo.4306855
[36] Tim Lindholm and Frank Yellin. 1997. *The Java Virtual Machine Specification*. Addison-Wesley.
[37] Zhibo Liu, Dongwei Xiao, Zongjie Li, Shuai Wang, and Wei Meng. 2023. Exploring Missed Optimizations in WebAssembly Optimizers. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 436–448. https://doi.org/10.1145/3597926.3598068
[38] Martin Maierhofer and M. Anton Ertl. 1998. Local Stack Allocation. In *Compiler Construction, 7th International Conference, CC'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1383)*, Kai Koskimies (Ed.). Springer, 189–203. https://doi.org/10.1007/BFB0026432

[39] Dmitry Malioutov and Kuldeep S. Meel. 2018. MLIC: A MaxSAT-Based Framework for Learning Interpretable Classification Rules. In *CP (Lecture Notes in Computer Science, Vol. 11008)*. Springer, 312–327. https://doi.org/10.1007/978-3-319-98334-9_21

[40] Joao Marques-Silva, Mikolas Janota, and Anton Belov. 2013. Minimal Sets over Monotone Predicates in Boolean Formulae. In *CAV (Lecture Notes in Computer Science, Vol. 8044)*. Springer, 592–607. https://doi.org/10.1007/978-3-642-39799-8_39

[41] Joao Marques-Silva, Ines Lynce, and Sharad Malik. 2021. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, Vol. 336. IOS Press, 133–182. https://doi.org/10.3233/FAIA200987

[42] Henry Massalin. 1987. Superoptimizer - A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*. 122–126. https://dl.acm.org/citation.cfm?id=36194

[43] William M. McKeeman. 1965. Peephole optimization. *Commun. ACM* 8, 7 (1965), 443–444. https://doi.org/10.1145/364995.365000

[44] Manasij Mukherjee, Pranav Kant, Zhengyang Liu, and John Regehr. 2020. Dataflow-based pruning for speeding up superoptimization. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 177:1–177:24. https://doi.org/10.1145/3428245

[45] Alexander Nadel. 2022. Introducing Intel(R) SAT Solver. In *SAT*. 8:1–8:23. https://doi.org/10.4230/LIPICS.SAT.2022.8

[46] Julian Nagele and Maria A Schett. 2019. Blockchain Superoptimizer. In *Preproceedings of 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2019)*. https://arxiv.org/abs/2005.05912.

[47] Sand Nallani. 2020. Issue in Solidity's repository reporting stack too deep errors. https://github.com/ethereum/solidity/issues/13158.

[48] Nina Narodytska, Alexey Ignatiev, Filipe Pereira, and Joao Marques-Silva. 2018. Learning Optimal Decision Trees with SAT. In *IJCAI*. ijcai.org, 1362–1368. https://doi.org/10.24963/IJCAI.2018/189

[49] Keerthi Nelaturu, Sidi Mohamed Beillahi, Fan Long, and Andreas G. Veneris. 2021. Smart Contracts Refinement for Gas Optimization. In *3rd Conference on Blockchain Research & Applications for Innovative Networks and Services, BRAINS 2021, Paris, France, September 27-30, 2021*. IEEE, 229–236. https://doi.org/10.1109/BRAINS52497.2021.9569819

[50] Toru Ogawa, Yangyang Liu, Ryuzo Hasegawa, Miyuki Koshimura, and Hiroshi Fujita. 2013. Modulo Based CNF Encoding of Cardinality Constraints and Its Application to MaxSAT Solvers. In *ICTAI*. 9–17. https://doi.org/10.1109/ICTAI.2013.13

[51] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodík, and Dinakar Dhurjati. 2016. Scaling up Superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016*, Tom Conte and Yuanyuan Zhou (Eds.). ACM, 297–310. https://doi.org/10.1145/2872362.2872387

[52] Jussi Rintanen. 2021. Planning and SAT. In *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, Vol. 336. IOS Press, 765–790. https://doi.org/10.3233/FAIA201003

[53] Andreas Rossberg, Ben L. Titzer, Andreas Haas, Derek L. Schuff, Dan Gohman, Luke Wagner, Alon Zakai, J. F. Bastien, and Michael Holman. 2018. Bringing the web up to speed with WebAssembly. *Commun. ACM* 61, 12 (2018), 107–115. https://doi.org/10.1145/3282510

[54] Olivier Roussel and Vasco M. Manquinho. 2021. Pseudo-Boolean and Cardinality Constraints. In *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, Vol. 336. IOS Press, 1087–1129. https://doi.org/10.3233/FAIA201012

[55] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and John Regehr. 2017. Souper: A Synthesizing Superoptimizer. *CoRR* abs/1711.04422 (2017). arXiv:1711.04422 http://arxiv.org/abs/1711.04422

[56] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, Vivek Sarkar and Rastislav Bodík (Eds.). ACM, 305–316. https://doi.org/10.1145/2451116.2451150

[57] Shikhar Singh, Mengshi Zhang, and Sarfraz Khurshid. 2019. Learning Guided Enumerative Synthesis for Superoptimization. In *Model Checking Software - 26th International Symposium, SPIN 2019, Beijing, China, July 15-16, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11636)*, Fabrizio Biondi, Thomas Given-Wilson, and Axel Legay (Eds.). Springer, 172–192. https://doi.org/10.1007/978-3-030-30923-7_10

[58] Carsten Sinz. 2005. Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In *CP*. 827–831. https://doi.org/10.1007/11564751_73

[59] Martin Suda. 2014. Property Directed Reachability for Automated Planning. *J. Artif. Intell. Res.* 50 (2014), 265–319. https://doi.org/10.1613/JAIR.4231

[60] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. 2016. Efficient SAT Approach to Multi-Agent Path Finding Under the Sum of Costs Objective. In *ECAI (Frontiers in Artificial Intelligence and Applications, Vol. 285)*. IOS Press, 810–818. https://doi.org/10.3233/978-1-61499-672-9-810

[61] Etherscan team. 2018. Etherscan. https://etherscan.io.

[62] Milk Road team. 2023. EthereumPrice. https://ethereumprice.org/history/?start=2019-02-28&end=2023-05-10&currency=USD.
[63] Solidity team. 2022. Solidity documentation. https://docs.soliditylang.org/en/v0.8.17/.
[64] Solidity team. 2023. Optimizer of Solidity compiler. https://docs.soliditylang.org/en/latest/internals/optimizer.html.
[65] W3C. 2016. WebAssembly. https://webassembly.org/.
[66] Gavin Wood. 2019. Ethereum: A secure decentralised generalised transaction ledger.
[67] Jinqiang Yu, Alexey Ignatiev, Peter J. Stuckey, and Pierre Le Bodic. 2021. Learning Optimal Decision Sets and Lists with SAT. *J. Artif. Intell. Res.* 72 (2021), 1251–1279. https://doi.org/10.1613/JAIR.1.12719
[68] Ziyi Zhao, Jiliang Li, Zhou Su, and Yuyi Wang. 2023. GaSaver: A Static Analysis Tool for Saving Gas. *IEEE Trans. Sustain. Comput.* 8, 2 (2023), 257–267. https://doi.org/10.1109/TSUSC.2022.3221444

Received 2023-11-16; accepted 2024-03-31