A Max-SMT Superoptimizer for EVM handling Memory and Storage*

Elvira Albert[1,2], Pablo Gordillo[2](✉), Alejandro Hernández-Cerezo[2], and Albert Rubio[1,2]

[1] Instituto de Tecnología del Conocimiento, Madrid, Spain
[2] Complutense University of Madrid, Madrid, Spain
pabgordi@ucm.es

Abstract. Superoptimization is a compilation technique that searches for the optimal sequence of instructions semantically equivalent to a given (loop-free) initial sequence. With the advent of SMT solvers, it has been successfully applied to LLVM code (to reduce the number of instructions) and to Ethereum EVM bytecode (to reduce its gas consumption). Both applications, when proven practical, have left out memory operations and thus missed important optimization opportunities. A main challenge to superoptimization today is handling memory operations while remaining scalable. We present GASOL^{v2}, a gas and bytes-size superoptimization tool for Ethereum smart contracts, that leverages a previous Max-SMT approach for only stack optimization to optimize also *wrt.* memory and storage. GASOL^{v2} can be used to optimize the size in bytes, aligned with the optimization criterion used by the Solidity compiler solc, and it can also be used to optimize gas consumption. Our experiments on 12,378 blocks from 30 randomly selected real contracts achieve gains of 16.42% in gas *wrt.* the previous version of the optimizer without memory handling, and gains of 3.28% in bytes-size over code already optimized by solc.

1 Introduction and Related Work

Superoptimization is an automated technique for code optimization that was proposed back in 1987 [20]. It aims at automatically finding the *optimal* (*wrt.* the considered optimization criteria) instruction sequence —which is semantically equivalent— to a given sequence of loop-free instructions. It differs from traditional optimization techniques in that it uses search rather than applying pre-cooked transformations. However, as it requires exhaustive search in the space of valid instruction sequences, it suffers from high computation demands and it was considered impractical for many years. The first attempts of applying superoptimization were within a GNU C compiler back in the nineties [15] and, later, it has also been applied for an x86-64 assembly language [10, 11].

There is a recent revival of superoptimization due to the availability of SMT solvers which offer powerful techniques to handle enumerative search and

* This research was funded by the Spanish MCIN/AEI/10.13039/ 501100011033/-FEDER "Una manera de hacer Europa" projects RTI2018-094403-B-C31 and RTI2018-094403-B-C33, by the CM project S2018/TCS-4314 co-funded by EIE Funds of the European Union, and by the Ethereum Foundation grant FY21-0372.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 201–219, 2022.
https://doi.org/10.1007/978-3-030-99524-9_11

to check semantic equivalence. The approaches to supercompilation based on SMT can be roughly classified into two types: (1) Those that use an external synthesis algorithm with pruning techniques, such as [9,12,17], and that invoke the SMT solver to solve certain queries. This is the approach of the Souper superoptimizer [22] that relies on the synthesis algorithm for loop-free programs of Gulwani et al. [17]; (2) Those that directly produce an SMT encoding of the problem and use the search engine of the solver. This is the approach of [18], EBSO [21] and SYRUP [7]. Both types of approaches have been proven to be practical on their own settings and optimization criteria: the analysis of blocks does not reach the timeout of 10 sec in 90% of the cases [7] in SYRUP, and Souper optimized three million lines of C++ in 88 minutes [22]. The optimizations achieved vary for the considered criteria, Souper reported around 4.4% reduction in number of instructions, and SYRUP reported 0.58% in the global Ethereum gas usage. Scalability has been partly achieved because challenging features have been left out of the encoding: *memory operations have been excluded both in Souper and SYRUP*. While EBSO included a basic encoding for memory operations, its practicality was not proven: EBSO times out in 82% of the blocks and achieves optimization in less than 1% of all analyzed blocks. Leaving out memory operations dismisses optimization opportunities of two kinds: (a) as it works on smaller blocks of instructions (since the optimizer stops when finding a memory operation), the stack optimization is more limited, and (b) besides it misses possible optimizations on the memory operations themselves (e.g., eliminating unnecessary accesses).

The Ethereum Virtual Machine (EVM) has two areas where it can store items (besides the stack): (1) the *storage* is where all *contract state* variables reside, every contract has its own storage and it is persistent between external function calls (transactions) and has a higher gas cost to use; (2) the *memory* is used to hold temporary values, and it is erased between transactions and thus is cheaper to use. For conciseness, we often use "memory" to include both storage and memory, as their treatment for optimization is identical except for their associated costs. Our big challenge is to be able to handle memory operations while remaining practical, i.e., not reaching the timeout in the optimization of the vast majority of the blocks. This is achieved by leveraging SYRUP's two-staged method [7] to handle memory: (i) the first stage is devoted to synthesize a stack specification from the bytecode and apply simplification rules to it, and (ii) in a second stage a Max-SMT solver is used to perform the search for the optimal solution. When lifting such two-staged method to handle memory operations, we make two important extensions: in stage (i), we now synthesize a stack *and* memory specification from the bytecode on which we detect dependencies among memory operations and possibly remove redundant operations; (ii) this dependency information is included in our second stage as part of the encoding so that the SMT solver only needs to consider the dependence among such memory instructions when performing the search. Our two-staged approach allows isolating the dependency analysis process from the search itself, reducing the effort the SMT solver does in order to find the optimal sequence. The approach of Bansal and Aiken [10] to handle memory operations differs from ours on the superoptimization scope and the search process itself. Their tool considers multiple target sequences from a training set simultaneously and generates a database of (possibly) millions of optimizations. They enumerate all well-formed instructions sequences up to a certain size, including memory operations, and test the equivalence among them via a hash function. Our tool considers each sequence of instructions to optimize independently and the search is done via the search engine of an SMT solver.

GASOL^{v2} can be considered a successor of SYRUP [7], as it adopts its two-staged process and reuses part of its components, but it incorporates three fundamental extensions, and a new experimental evaluation, that constitute the main contributions of this paper: (1) GASOL^{v2} starts from the assembly json [1] generated by the solc compiler, rather than being used as a standalone optimization tool as SYRUP. This is fundamental to achieve a wide use of the tool since it is already linked to one of the most used compilers for coding Ethereum smart contracts. (2) It optimizes memory and storage operations using on one hand rule simplifications at the level of a specification synthesized from the bytecode, and on the other hand, a new SMT encoding which enables achieving a great balance between the accuracy and the overhead of the process. (3) While SYRUP is a tool that only optimizes the gas consumption of the bytecode, we have generalized some of its components to enable other optimization criteria. Currently we have included as well size in bytes, but other criteria can be easily incorporated now to the superoptimizer. (4) Besides we have performed a thorough experimental evaluation of our tool and have compared the results *wrt.* those obtained by SYRUP. The main conclusion of our evaluation is that handling memory operations in superoptimization pays off: it can achieve gains of 16.42% in gas over SYRUP, and reductions of 0.1% in gas and 3.28% in size (on already optimized code). If we assume that these savings are uniformly distributed, and the gas data obtained from Etherscan is constant, the 0.1% gas saved wrt the SYRUP [7] would amount nearly to 9.5 Million dollars in 2021.

GASOL^{v2} is part of the GASOL project [3], a GAS Optimization tooLkit for Ethereum smart contracts. The initial GASOL tool (i.e., GASOL^{v1}), presented in [5], aimed at detecting gas-expensive patterns within program loops (using resource analysis) and made a program transformation (which does not rely on SMT solvers) at the source code level. Hence, it contains a *global* (inter-block) optimization technique that is orthogonal to our superoptimizer, in which we perform *local* (or intra-block) transformations on loop-free code, and besides we work at bytecode rather than at source level. Both complementary techniques will be integrated within the GASOL toolkit, hence their names. In what follows, we drop $v2$ and use GASOL to refer to the tool presented in this paper.

2 The Architecture of GASOL

Figure 1 displays the architecture of GASOL, white components are borrowed from other tools, while gray components correspond to the new developments of this paper (either completely new, like **DEP**, or novel extensions for memory handling of previous SYRUP's implementations, like **SPEC**, **SIMP** and **SMS**).

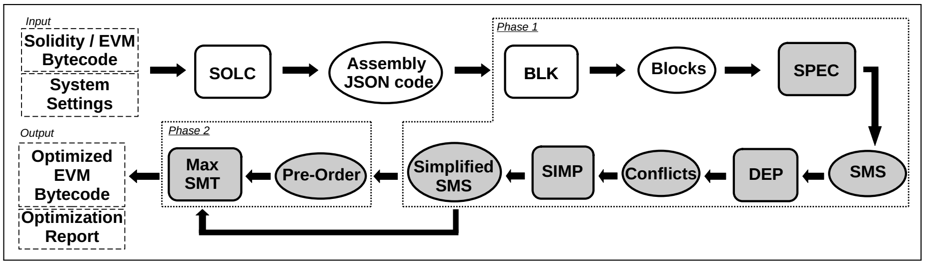

Fig. 1: Architecture of GASOL^{v2}

The **input** to GASOL is a smart contract (either its source in Solidity or its compiled EVM bytecode [23]), a selection of the optimization criteria (currently we are supporting gas consumption and size in bytes), and system settings (this includes compiler options for invoking the solc compiler and GASOL settings like the timeout per block of instructions). The **output** of GASOL is an optimized bytecode program and optionally a report with detailed information on the optimizations achieved (e.g., number of blocks optimized, number of blocks proven optimal, gas/size reduction gains, optimization time, among others).

The first component, labeled **SOLC** in the figure, invokes the Solidity compiler solc to obtain the bytecode in their assembly json exchange format [1]. Working on this exchange format has many advantages, one is that we can enable the optimizer of solc [4] and start the superoptimization from an already optimized bytecode. Besides, the format has been designed to be a usable common denominator for EVM 1.0, EVM 1.5 and Ewasm. Hence, we argue it is a good source for superoptimization as different target platforms will be able to use our tool equally. The assembly json format provides the EVM bytecode of the smart contract, metadata that relates it with the source Solidity code, and compilation information such as the version used to generate the bytecode. The output yield by GASOL can also be returned in assembly json format so that it can be used by other tools working on this format in the future. From the assembly json, the next component **BLK** partitions the bytecode given by solc into a set of sequences of loop-free bytecode instructions, named *blocks*, which correspond to the blocks of the CFG and also computes the size of the stack when entering each block.[3] We omit details of this step as it is standard in compiler construction and, for the case of the EVM, has been already subject of other analysis and optimization papers (see, e.g. [8, 14, 16] and their references).

The next component **SPEC** synthesizes a functional specification of the operand stack and of the memory and storage (SMS for short) for each block of bytecode instructions. This is done by symbolically executing the bytecodes in the block to extract from them what the contents of the operand stack and of the memory/storage are after executing them. The description of this component is given in Sec. 3.1. Next, **DEP** establishes the dependencies among the memory accesses from which a pre-order, that determines when a memory access needs

[3] In EVM, it is possible to reach a block with different stack sizes, and all such sizes can be statically computed. We will refer to the minimum or maximum when needed.

(1) $\tau(\texttt{MLOAD}, <\mathcal{S},\mathcal{M},\mathcal{S}t>) := <[\texttt{MLOAD}(\mathcal{S}[0])] + \mathcal{S}[1:n], \mathcal{M} + [\texttt{MLOAD}(\mathcal{S}[0])], \mathcal{S}t>$
(2) $\tau(\texttt{MSTORE}, <\mathcal{S},\mathcal{M},\mathcal{S}t>) := <\mathcal{S}[2:n], \mathcal{M} + [\texttt{MSTORE}(\mathcal{S}[0],\mathcal{S}[1])], \mathcal{S}t>$
(3) $\tau(\texttt{SLOAD}, <\mathcal{S},\mathcal{M},\mathcal{S}t>) := <[\texttt{SLOAD}(\mathcal{S}[0])] + \mathcal{S}[1:n], \mathcal{M}, \mathcal{S}t + [\texttt{SLOAD}(\mathcal{S}[0])]>$
(4) $\tau(\texttt{SSTORE}, <\mathcal{S},\mathcal{M},\mathcal{S}t>) := <\mathcal{S}[2:n], \mathcal{M}, \mathcal{S}t + [\texttt{SSTORE}(\mathcal{S}[0],\mathcal{S}[1])]>$
(5) $\tau(\texttt{SWAPX}, <\mathcal{S},\mathcal{M},\mathcal{S}t>) := \text{let temp} = \mathcal{S}[0] \;\; <\mathcal{S}[0/X][X/\text{temp}], \mathcal{M}, \mathcal{S}t>$
(6) $\tau(\texttt{POP}, <\mathcal{S},\mathcal{M},\mathcal{S}t>) := <\mathcal{S}[1:n], \mathcal{M}, \mathcal{S}t>$

Fig. 2: SMS Synthesis by Symbolic execution

to be performed before another one, is generated. For instance, subsequent load accesses, which are not interleaved by any store, do not have dependencies among them, while they do have with subsequent write accesses to the same positions. This phase is described in Secs. 3.2 (dependencies) and 3.3 (pre-order). In the next component **SIMP**, we apply simplification rules on the SMS. We include all stack simplification rules of SYRUP [7], as well as the additional rules we have developed for memory/storage simplifications. For instance, successive write accesses that overwrite the same memory position are simplified to a single one provided the same memory location is not read by any other instruction between them. The description of this component is given in Sec. 3.2. Finally, we generate a **Max-SMT** encoding from the (simplified) SMS that incorporates the pre-order established by the component **DEP** and from which the optimized bytecode is obtained. The description of this component is given in Sec. 4.

3 Synthesis of Stack and Memory Specifications

This section describes the first stage of the optimization (components **SPEC**, **SIMP** and **DEP**) that consists in synthesizing from a loop-free sequence of bytecode instructions a *simplified* specification of the stack and of the memory/storage (with the dependencies) that the execution of such bytecodes produces.

3.1 Initial Stack and Memory/Storage Specification

For each block, we synthesize its Stack and Memory Specification (SMS) by symbolically executing the instructions in the sequence. Function τ in Fig. 2 defines the symbolic execution for the memory/storage operations (1-4) and includes two representative stack opcodes (5-6). The first parameter of τ is a bytecode instruction and the second one is the SMS data structure $<\mathcal{S}$,$\mathcal{M}$,$\mathcal{S}t>$ whose first element corresponds to the stack ($\mathcal{S}$), the second one to the memory ($\mathcal{M}$), and the third one to the storage ($\mathcal{S}t$). The stack $\mathcal{S}$ is a list whose position $\mathcal{S}[0]$ corresponds to the top of the stack. At the beginning of executing a block, the stack contains the minimum number of elements needed to execute the block represented by symbolic variables s_i, where s_i models the element at $\mathcal{S}[i]$. The resulting list $\mathcal{M}$ ($\mathcal{S}t$ resp.) will contain the sequence of memory (storage resp.) accesses executed by the block. By abuse of notation, we often treat lists as sequences. Both $\mathcal{M}$ and $\mathcal{S}t$ are empty before executing the block symbolically. As an example, the symbolic execution of `SSTORE` removes the two top-most elements from $\mathcal{S}$, and adds the symbolic expression $\texttt{SSTORE}(\mathcal{S}[0],\mathcal{S}[1])$ to the storage sequence. Similarly, `SLOAD` removes from the top of the stack the position to be read, puts on the top of the stack the symbolic expression $\texttt{SLOAD}(\mathcal{S}[0])$ that represents the value read from the storage position $\mathcal{S}[0]$, and adds the same expression to the storage sequence $\mathcal{S}t$. As a result of applying τ to a sequence of bytecodes, the SMS obtained provides a specification of the target stack after executing the sequence in terms of the elements located in the stack before executing the sequence and, the target memory/storage (given as a sequence of accesses) after executing the sequence in terms of the input stack elements too.

Example 1. Consider the following bytecode that belongs to a real contract (bytecodes 0 to 47 of Welfare [2]). Its assembly json yield by the **SOLC** component contains 4524 bytecodes and after being partitioned by **BLK** we have 437 blocks to optimize. We illustrate the superoptimization of this block that contains in total 48 bytecodes from which 5 are the (underlined) memory/storage accesses:

```
1  PUSH1 80     9  DUP2        17 DUP4        25 PUSH2 3E8   33 PUSH2 FFFF  41 MUL
2  PUSH1 40     10 SLOAD       18 PUSH2 FFFF  26 PUSH1 1     34 MUL         42 OR
3  MSTORE       11 DUP2        19 AND         27 PUSH1 16    35 NOT         43 SWAP1
4  PUSH1 64     12 PUSH2 FFFF  20 MUL         28 PUSH2 100   36 AND         44 SSTORE
5  PUSH1 1      13 MUL         21 OR          29 EXP         37 SWAP1       45 POP
6  PUSH1 14     14 NOT         22 SWAP1       30 DUP2        38 DUP4        46 CALLVALUE
7  PUSH2 100    15 AND         23 SSTORE      31 SLOAD       39 PUSH2 FFFF  47 DUP1
8  EXP          16 SWAP1       24 POP         32 DUP2        40 AND         48 ISZERO
```

As **BLK** returns that the stack is empty when entering the block, we apply τ to the initial state $<[\,],[\,],[\,]>$ and produce the following SMS at the next selected lines:

$L1 : \tau(\texttt{PUSH1 80}, <[\,],[\,],[\,]>) = <[128],[\,],[\,]>$
$L2 : \tau(\texttt{PUSH1 40}, <[128],[\,],[\,]>) = <[64,128],[\,],[\,]>$
$L3 : \tau(\texttt{MSTORE}, <[64,128],[\,],[\,]>) = <[\,],[\texttt{MSTORE(64,128)}],[\,]>$

Finally, we get that at L48 $\mathcal{S} = [\texttt{ISZERO(CALLVALUE)}, \texttt{CALLVALUE}]$, $\mathcal{M} = [\texttt{MSTORE(64,128)}]$, $\mathcal{S}t = [\texttt{SLOAD}_1\texttt{(1)}, \texttt{SSTORE(1,V1)}, \texttt{SLOAD}_2\texttt{(1)}, \texttt{SSTORE(1,V2)}]$ where $\texttt{V1} = \texttt{OR}(\texttt{MUL}(...)), \texttt{AND}(\texttt{NOT}(..)), \texttt{SLOAD}_1\texttt{(1)})$ (omitting subexpressions) and V2 is another similar expression involving arithmetic, binary operations and $\texttt{SLOAD}_2\texttt{(1)}$. Note that we use subscripts to distinguish the SLOAD instructions by their position in $\mathcal{S}t$. The stack specification contains a term that represents the result of the opcode CALLVALUE (executed at line 46, L46 for short), and a term with the result of executing the opcode ISZERO on CALLVALUE, stored on top of the stack. The memory only contains one element that is obtained by symbolically executing the three first instructions. The PUSH instructions at L1 and L2 introduce the values 64 and 128 on the stack, and the MSTORE executed at L3 introduces in $\mathcal{M}$ the symbolic expression MSTORE(40,80). Similarly, $\mathcal{S}t$ contains the sequence of symbolic expressions that represent the storage instructions executed in the block at L10, L23, L31 and L44 respectively. The expressions corresponding to V1 and V2 are also obtained by applying function τ to the corresponding state. These stack expressions can be simplified in the next step using the rules in [7].

We note that the EVM memory is byte addressable (e.g., with instruction MSTORE8) and two different memory accesses may overlap. For simplicity of the presentation, we only consider the general case of word-addressable accesses, but the technique extends easily to the byte addressable case. In what follows, we use LOAD to abstract from the specific memory (MLOAD) and storage (SLOAD) bytecodes (and the same for STORE), when they are treated in the same way.

3.2 Memory/Storage Simplifications

In order to define the simplifications, and to later indicate to the SMT solver which memory instructions need to follow an order, we compute the conflicts between the different load and store instructions within each sequence.

Definition 1. *Two memory accesses A and B* conflict*, denoted as conf(A,B) if:*
(i) A is a store and B is a load and the positions they access might be the same;
(ii) A and B are both stores, the positions they modify might be the same, and they store different values.

Note that in (ii) two store instructions that might operate on the same position do not conflict if the values they store are equal, as we will reach the same memory state regardless of the order in which the stores are executed. Note that two load instructions are never in conflict as the memory state does not change if we execute them in one order or another.

Given the SMS obtained in Sec. 3.1, we achieve simplifications by applying the stack simplification rules of [7] and, besides, the following new memory simplification rules based on Def. 1 to the M and S components (that achieve optimizations of type (b) according to the classification mentioned in Sec. 1):

Definition 2 (memory simplifications). *Let $< \mathcal{S}, \mathcal{M}, \mathcal{S}t >$ be an SMS, we can apply the following simplifications to any subsequence $b_1, \ldots, b_n$ in $\mathcal{M}$ or $\mathcal{S}t$:*

i) if $b_1 =$STORE(p, v) and $b_n =$LOAD(p) and $\nexists b_i =$STORE with $i \in \{2, \ldots, n-1\}$ and conf(b_1,b_i), we simplify it to $b_1, \ldots, b_{n-1}$ and replace b_n by v in the resulting SMS.
ii) if $b_1 =$STORE(p, v) and $b_n =$STORE(p, w) and $\nexists b_i =$LOAD with $i \in \{2, \ldots, n-1\}$ conf(b_1,b_i), we simplify it to $b_2, \ldots, b_n$.
iii) if $b_1 =$LOAD(p) and $b_n =$STORE$(p,$LOAD$(p))$ and $\nexists b_i =$STORE with $i \in \{2, \ldots, n-1\}$ conf(b_1,b_i), we simplify it to $b_1, \ldots, b_{n-1}$.

The simplifications can be applied in any order within $\mathcal{M}$ and $\mathcal{S}t$ until the process converges and the resulting sequence cannot be further simplified.

Intuitively, in (i), a load instruction from a position after a store instruction to the same position is simplified in the stack to the stored value provided there is no other store operation in between that might have changed the content of this position. In (ii), two subsequent store instructions to the same position are simplified to a single store if there is no load access on the same position between them. In (iii), a store instruction that stores in a position the result of the load in the same position can be removed, provided there is no other store in between that might have changed the content of this position. Note that such simplification rules can be applied to general-purpose compilers.

Example 2. In the SMS of Ex. 1, we have that conf(`SLOAD`$_1$`(1)`,`SSTORE(1,V1)`), conf (`SLOAD`$_1$`(1)`,`SSTORE(1,V2)`), conf(`SLOAD`$_2$`(1)`,`SSTORE(1,V1)`), conf(`SLOAD`$_2$`(1)`,`SSTORE(1, V2)`) and conf(`SSTORE(1,V1`,`SSTORE(1,V2)`) as all accesses operate on the same location. With these conflicts, we can apply rule i) to `SLOAD`$_2$`(1)`, as the previous `SSTORE` instruction has stored the value `V1` at the same location and there are no other storage instructions with conflict between them. Hence, we eliminate it from $\mathcal{S}t$ and replace it by `V1` in the resulting SMS. After that, we are able to apply rule ii) on the two `SSTORE` instructions as they store a value at the same position without conflict loads in between. Then, we remove `SSTORE(1,V1)` from $\mathcal{S}t$. The resulting SMS has the same $\mathcal{S}$ and $\mathcal{M}$ and $\mathcal{S}t$ is now [`SLOAD`$_1$`(1)`, `SSTORE(1,V2')`] where `V2'` is `V2` replacing `SLOAD`$_2$`(1)` by `V1`.

3.3 Pre-Order for Memory and Uninterpreted Functions

Given the SMS and using the conflict definition above, we generate a pre-order, as defined below, that indicates to the SMT solver the order between the memory accesses that needs to be kept in order to obtain the same memory state as the original one. Clearly, having more accurate conflict tests will result in weaker pre-orders and hence a wider search space for the SMT solver. This in turn will result in potentially larger optimization. Our implementation is highly parametric on the conflict test **DEP** so that more accurate tests can be easily incorporated.

Definition 3. *Let A and B be two memory accesses in a sequence S. We say that B has to be executed after A in S, denoted as $A \sqsubset B$ if:*

i) (store-store) B is a store instruction and A is the closest store instruction predecessor of B in S such that conf(*A*,*B*).
ii) (load-store) A is a load instruction and B is the closest store instruction successor of A in S such that conf(*B*,*A*).
iii) (store-load) B is a load instruction and A is the closest store instruction predecessor of B in S such that conf(*A*,*B*).

Let us observe that we do not compute the closure for the dependencies at this stage, as the SMT solver will infer them, as explained in Sec. 4.2.

Example 3. From the simplified SMS of Ex. 2, we get the following load-store dependency, `SLOAD`$_1$`(1)` $\sqsubset$ `SSTORE(1,V2')`, while the access `MSTORE(64,128)` has no dependencies as it is the unique memory operation.

Importantly, the notion of pre-order between memory instructions can also be naturally extended to all other operations that occur in the specification of the target stack. These operations are handled as uninterpreted functions and have to be called in the right order to build the result that is required in the target stack. Therefore, we propose a novel implementation (both in SYRUP and GASOL) that extends the pre-order $\sqsubset$ to uninterpreted functions by adding $A \sqsubset B$ also when:

iv) (uninterpreted-functions) A and B are uninterpreted functions that occur in the target stack as $B(\ldots, A(\ldots), \ldots)$.

While in the case of uninterpreted functions the pre-order is used for improving performance, for memory operations the use of the pre-order is mandatory for soundness, since it is what ensures that the obtained block after optimization has the same final state (in the stack, memory and storage) than the original block.

3.4 Bounding the Operations Position

As we will show in the next section, a solution to our SMT encoding assigns a position in the final instruction list to each operation such that the target stack is obtained. A key element for the performance of the encoding we propose in this paper is based on extracting from the instruction pre-order $\sqsubset$, upper and lower bounds to the position the operations can take in the instruction list. The lower bound for a given function is obtained by inspecting the subterm where it occurs in the target stack and analyzing its operands to detect the earliest point in which the result of all them can be placed in the stack, taking into account that shared subcomputations can be obtained using a `DUP` opcode. On the other hand, the upper bound for a function is obtained by inspecting the position in the target stack they occur and analyzing the operations that use the term that is headed by this function, to obtain the latest point in which this term could be computed. From this analysis, we obtain both the upper $UB(\iota)$ and lower $LB(\iota)$ bounds for every uninterpreted (which includes the load) and store operation ι, which are extensively used in the encoding provided in the next section.

4 Max-SMT Superoptimization

This section describes the second stage of the optimization process (named **Max SMT** in Fig. 1) that consists in producing, from the SMS and the dependencies, a Max-SMT encoding such that any valid model corresponds to a bytecode equivalent to the initial one and optimized for the selected criterion.

4.1 Stack Representation in the SMT Encoding

The stack representation is the same as in [7]: the stack can hold non-negative integer constants in the range $\{0, \ldots, 2^{256}-1\}$, matching the 256-bit words in the EVM; initial stack variables $s_0, \ldots, s_{k-1}$, represent the initial (unknown) elements of the stack; and fresh variables $s_k, \ldots, s_v$ abstract each different subterm (built from opcodes and the initial stack variables) that appears in the SMS. A stack variable of the form s_i is represented in the encoding as the integer constant $2^{256}+i$, so that all stack elements in the model are integer values. To represent the contents of the stack after applying a sequence of instructions, a bound on the number of operations b_o and the size of the stack b_s must be given. These numbers are statically computed by considering the size of the initial block and the maximum number of stack elements involved. Then, propositional variables $u_{i,j}$, with $i \in \{0, \ldots, b_s-1\}$ and $j \in \{0, \ldots, b_o\}$, are used to denote whether there exists an element at position i in the stack after executing the first j operations, where the element $u_{0,j}$ refers to the topmost element of the stack. Quantified variables $x_{i,j} \in \mathbb{Z}$ are introduced to identify the word at position i after applying j operations, following the same format as $u_{i,j}$.

An instruction $\iota \in \mathcal{I}$ in the encoding can be either a basic stack opcode (`POP`, `SWAPk`, ...), a distinct expression that appears in the SMS or the extra instruction `NOP` that represents the possibility no opcode has been applied. A mapping θ is introduced to link every instruction in $\mathcal{I}$ to a non-negative integer in $\{0, \ldots, m_\iota\}$, where $m_\iota + 1 = |\mathcal{I}|$. This way, we can introduce the existentially quantified variables t_j, with $t_j \in \{0, \ldots, m_\iota\}$ and $j \in \{0, \ldots, b_o-1\}$, to denote that the instruction ι is applied at step j when $t_j = \theta(\iota)$. There is a special case to be considered when identifying the instructions from an SMS: each expression containing a single occurrence of an opcode in W_{base} (see [23]) is considered as an independent expression with a different ι. Opcodes in W_{base} consume no operand from top of the stack and have lower gas cost and equal byte count as `DUPk`, so we can safely assume that in an optimal block such expressions are never duplicated. For efficiency reasons, we also apply the reciprocal: any other expression is forced to appear exactly once in our solution, as our experiments show that duplicating the expression is always better than computing it more than once. However, note that this may not hold, in general, when the cost of the expression is low or the size of the operating stack is high, and hence, although is highly unlikely, we may lose some better solutions. From this assumption, we have that every ι we have introduced must appear exactly once in every model, which simplifies greatly both the pre-order encoding and the gas model used. The following example illustrates how the SMS is processed and the relevance of considering W_{base}:

Example 4. Consider a modified version of Ex. 1, in which $\mathcal{S} = [\texttt{ISZERO(CALLVALUE)}, \texttt{CALLVALUE}]$ but $\mathcal{M}, \mathcal{S}t$ are both empty. b_0, b_s are bounded to 3 and 2 resp., as three instructions are enough to compute the given SMS and it reaches a stack size of two elements. Each application of `CALLVALUE` is considered independently, as $\texttt{CALLVALUE} \in W_{base}$. Variables $s_0 := 2^{256}, s_1 := 2^{256}+1, s_2 := 2^{256}+2$ are introduced to represent the stack variable obtained from $\texttt{CALLVALUE}_0, \texttt{CALLVALUE}_1$ and $\texttt{ISZERO(CALLVALUE}_1\texttt{)}$. GASOL creates the following θ map:

$$\theta := \{\texttt{PUSH} : 0, \texttt{POP} : 1, \texttt{NOP} : 2, \texttt{DUP1} : 3, \texttt{SWAP1} : 4,$$
$$\texttt{CALLVALUE}_0 : 5, \texttt{CALLVALUE}_1 : 6, \texttt{ISZERO(CALLVALUE}_1\texttt{)} : 7\}$$

The optimal sequence is `CALLVALUE CALLVALUE ISZERO`, which consumes 7 units of gas. It improves the cost of L46-L48, which consumes 8 due to the use of `DUP1`.

The set of instructions $\mathcal{I}$ can be split in four subsets $\mathcal{I}_S \uplus \mathcal{I}_U \uplus \mathcal{I}_C \uplus \mathcal{I}_{St}$:

- $\mathcal{I}_S$ contains the basic stack operations: `PUSH`, `POP`, `NOP`, $\texttt{DUP}k$, and $\texttt{SWAP}k$, with $k \in \{1, \ldots, min(b_s - 1, 16)\}$. $\texttt{DUP}k$ and $\texttt{SWAP}k$ are restricted by b_s because they cannot deal with elements that go beyond the maximum stack size.
- $\mathcal{I}_U$ contains the non-commutative uninterpreted functions that appear in the SMS. Its subset $\mathcal{I}_L \subseteq \mathcal{I}_U$ denotes the set of load instructions.
- $\mathcal{I}_C$ contains the commutative uninterpreted functions in the SMS.
- $\mathcal{I}_{St}$ contains the write operations in memory structures.

The encoding for subsets $\mathcal{I}_S \uplus \mathcal{I}_U \uplus \mathcal{I}_C$ was already considered in [7], whereas $\mathcal{I}_{St}$ was left out. Instead, blocks were split when an opcode belonging to $\mathcal{I}_{St}$ was found. The inclusion of $\mathcal{I}_{St}$ instructions in the model leads to more savings in gas, as more optimizations can be applied in larger blocks (those correspond to optimizations of type (a) in the classification given in Sec. 1).

For each $\iota \in \mathcal{I}$ and each possible position j in the sequence of instructions, we add a constraint to represent the impact of this combination on the stack. These constraints match the semantics of τ when projecting onto the stack component, so that we encode the elements of the stack after executing ι in terms of the ones before its execution. They follow the structure $t_j = \theta(\iota) \Rightarrow C_\iota(j)$, where $C_\iota(j)$ expresses the changes in the stack after applying ι. The constraints for $\mathcal{I}_S \uplus \mathcal{I}_U \uplus \mathcal{I}_C$ are detailed in [7], our extension in this section is only to include the constraints to reflect the impact of storage operations on the stack. For this purpose, we use an auxiliary predicate *Move* (already used in [7]) to denote that all elements in the stack are moved two positions to the right in the resulting stack state. Thus, we have the following constraint for each position j and each

$\iota \in \mathcal{I}_{St}$, where o_0 and o_1 denote the position and value stored:

$$C_{St}(j,\iota) := \ t_j = \theta(\iota) \Rightarrow u_{0,j} \ \wedge \ u_{1,j} \ \wedge \ x_{0,j} = o_0 \ \wedge \ x_{1,j} = o_1 \ \wedge \\ Move(j, 2, b_s - 1, -2) \ \wedge \ \neg u_{b_s-1,j+1} \ \wedge \ \neg u_{b_s-2,j+1}$$

Finally, we express the contents of the stack before executing the instructions of the block (initial stack) and after having executed them (target stack) by assigning the corresponding values (whether constants or stack variables) to $u_{i,0}, x_{i,0}$ and to u_{i,b_o}, x_{i,b_o} respectively. The overall SMT encoding for the stack representation is denoted as C_{SFS} and it is encoded using *QF_LIA* logic.

Example 5. Following Ex. 4, GASOL generates the constraint shown below to update the contents of the stack after applying $\iota =$ `ISZERO(CALLVALUE`$_1$`)` at step 2:

$$C_\iota(2,\iota) := \ t_2 = 7 \Rightarrow u_{0,2} \ \wedge \ x_{0,2} = 2^{256} + 1 \ \wedge \\ u_{0,3} \ \wedge \ x_{0,3} = 2^{256} + 2 \ \wedge \ u_{1,3} = u_{1,2} \ \wedge \ x_{1,3} = x_{1,2}$$

4.2 Encoding the Pre-order Relation

Once the stack representation has been formalized, we also need to consider the conflicts that appear among memory operations as part of our encoding, as well as the dependencies between uninterpreted functions. All this is made by encoding the pre-order relation given in Sec. 3.3. We consider each pair of instructions ι, ι' s.t. $\iota \sqsubset \iota'$. We aim to prevent conflicting operations from appearing in the wrong order in a model, by imposing that ι cannot occur in the assignment after ι'.

Our proposed approach consists in introducing a variable $l_{\theta(\iota)}$ for every instruction $\iota \in \mathcal{I}_C \cup \mathcal{I}_U \cup \mathcal{I}_{St} := \mathcal{I}_{lord}$ to track the position it appears in a sequence. This information is useful for specifying multiple conditions in the encoding that are difficult to reflect otherwise. Firstly, these variables implicitly enforce that ι must be tied to exactly one position, and thus, included in every sequence exactly once. Besides, we can narrow the positions in which ι can appear by using $LB(\iota), UB(\iota)$ bounds. Finally, as *QF_LIA* supports ordering among variables, the order between conflicting instructions can be encoded as a plain comparison between their positions. Hence, the following constraints are derived:

$$L_P(\iota) := LB(\iota) \leq l_{\theta(\iota)} \leq UB(\iota) \ \wedge \bigwedge_{LB(\iota) \leq j \leq \ UB(\iota)} (l_{\theta(\iota)} = j) \Leftrightarrow (t_j = \theta(\iota))$$

$$L_{lord}(\iota, \iota') := l_{\theta(\iota)} < l_{\theta(\iota')} \ \text{ where } \iota \sqsubset \iota'$$

Regarding memory operations, there is no need to consider special cases. The whole encoding can be expressed as follows:

$$C_{SMS} := \ C_{SFS} \ \wedge \bigwedge_{\iota \in \mathcal{I}_{lord}} L_P(\iota) \ \wedge \bigwedge_{\iota \sqsubset \iota'} L_{lord}(\iota, \iota')$$

4.3 Optimization using Max-SMT

As in [7], we formulate the problem of finding an optimal block as a partial weighted Max-SMT problem. In this section we show that the same encoding for gas optimization can be used in the presence of memory operations and that other optimization criterion, like bytes-size, can be included as well in our framework. Basically, in our Max-SMT problem, the *hard constraints* that must be satisfied by every model are those constraints for computing the SMS; and the *soft constraints* are used to find the optimal solution: a set of pairs $\{[C_1, \omega_1], \ldots, [C_n, \omega_n]\}$, where C_i denotes an SMT clause and ω_i its weight. The Max-SMT solver minimizes the weights of the falsified soft constraints. The weights of soft constraints presented in [7] match the gas spent for the sequence of instructions, thus ensuring an optimal model corresponds to a block that spends the least possible amount of gas. This gas encoding is also included in GASOL, but instructions in $\mathcal{I}_{lord}$ are removed from the *soft constraints*. Hard constraints already assert the exact number of times these instructions must appear in a sequence and therefore, they only add unnecessary extra cost that may harm the search of an optimality proof.

However, gas consumption is not the only relevant objective to consider when optimizing the code. When a contract is deployed, a fee of 200 units of gas must be paid for each non-zero byte of the EVM binary code. The desired trade-off between the initial deployment cost and invoking transactions can be specified in solc by setting the expected number of contract runs. In some cases, this leads to solc intentionally not fully replacing expressions that have a constant result by the value they represent if this constant is a large number, since the needed PUSH instructions will need many more non-zero bytes and hence will increment the deployment gas cost. For instance, if we want to have $2^{256} - 1$ on the top of the stack we can either push a zero and perform the bitwise NOT operation, which has gas cost 6 and non-zero bytes length 2 or push $2^{256} - 1$ directly which has gas cost 3 but non-zero bytes length 33.

When the bytes-size criterion is selected, we disable the application of the simplification rules of [7] that increase the byte-size and, besides, propose the next approach based on the *bytes-size model* for the Max-SMT encoding. This model is fairly simple except for the handling of the PUSH related instructions, denoted as $\mathcal{I}_P$ in what follows. All instructions that are not in $\mathcal{I}_P$ use exactly one byte. Instead PUSHx instructions take one byte to specify the opcode itself, and x bytes to include the pushed value. A first attempt to encode the weight of the PUSHx we tried was based on precisely describing the size in bytes based on the corresponding 32 options that x can take in terms of number of bytes. (recall that in EVM we have 256-bit words). This encoding is precise, but did not work in practice. An alternative, much simpler encoding, is based on the observation that numerical values can only appear in a model because at least once the corresponding PUSHx instruction is made. Later on, this value can be repeated using DUP, which has a minimal cost *wrt.* size of bytes, but if the block is large, some SWAP operation may also be needed. To make the encoding perform well in practice, we need to associate a single constant weight to all PUSHx operations, that is high enough to avoid models where expensive PUSHx operations are performed more than once instead of duplicating them. Our experiments have shown that a weight of 5 is enough to obtain optimal results for the sizes of blocks that the Max-SMT is able to handle. Then, we can assume NOP instructions cost 0 units, instructions in $\mathcal{I}_p$ costs 5 units and the remaining instructions cost 1 unit. Hence, three disjoint sets are introduced to match previous costs: $W_0 := \{\texttt{NOP}\}$,

$W_5 := \mathcal{I}_p$ and $W_1 := \mathcal{I}_S \setminus (W_0 \uplus W_5)$. Ω' bytes-size model is followed directly:

$$\Omega'_{SMS} := \bigcup_{0 \leqslant j < b_o} \{[t_j = \theta(\texttt{NOP}), 1], [\bigvee_{\iota \in W_0 \uplus W_1} t_j = \theta(\iota), 4]\}$$

Example 6. The optimized bytecode returned by GASOL for the gas criterion is `PUSH24* PUSH 80 PUSH 40 MSTORE PUSH 1 SLOAD PUSH32* AND PUSH21* OR PUSH32* AND OR PUSH 1 SSTORE CALLVALUE CALLVALUE ISZERO` (using * to skip large constants), which achieves a reduction of 5905 units *wrt.* the original version and is proven optimal. For the bytes-size criterion, GASOL times out due to the larger size of the block when size-increasing simplification rules are disabled. This issue will be discussed in Sec. 5.

5 Implementation and Experiments

This section provides further implementation details and describes our experimental evaluation. The GASOL tool is implemented in Python and uses as Max-SMT solver OptiMathSAT (OMS) [13] version 1.6.3 (which is the optimality framework of MathSAT). The aim of the experiments is to assess the effectiveness of our proposal by comparing it with the previous tool SYRUP. A timeout is given to the tools to specify the maximum amount of time that they can use for the analysis of each block. The timeout given to GASOL must be larger than for SYRUP because it works on less and larger blocks in order to analyze the same contract. We have used as timeout for SYRUP 10 sec, and for GASOL, we use 10*(#store+1) sec, as this would correspond to the addition of the times in SYRUP given to the partitioned blocks. It should be noted though that the cost of the search to be performed grows exponentially with the number of additional instructions. Therefore, in spite of giving a similar timeout, GASOL might time out in cases in which it has to deal with rather large blocks, while SYRUP does not on the corresponding smaller partitioned blocks. For this reason, we have implemented two additional versions: gasol_{all} splits the blocks at all stores as SYRUP, and gasol_{24} splits at store instructions only those blocks that have a size larger than 24 instructions. This is because we have observed during experimentation that the SMT search does not terminate in a reasonable time from that size on. The 24-partitioning starts from the end of the block and splits it if it finds a store. If the partitioned sub-block (from the start) still has a size larger than 24, further partitioning is done again if a new store is found from its end, and so on. Still, depending on where the stores are, the resulting blocks can have sizes larger than 24, as it happens in SYRUP as well. Further experimentation will be needed to come up with intelligent heuristics for the partitioning. The gasol versions implement all techniques described in the paper, including the SMT encoding dependencies between uninterpreted functions as described in Sec. 3.3. We have the following versions of GASOL and SYRUP in the evaluation: (1) syrup_{cav} is the original tool from [7], (2) gasol_{all} splits the blocks at all stores as in syrup_{cav}, (3) gasol_{24} performs the 24-partitioning described above, (4) gasol_{none} does not perform any additional partitioning of blocks, and (5) gasol_{best} uses gasol_{all}, gasol_{24}, and gasol_{none}, as a portfolio of possible optimization results (running them in parallel) and keeps the best result.

We run the tools using the gas usage and the bytes-size criteria in Sec. 4.3. As already mentioned, SYRUP in [7] did not include the bytes-size criterion,

	$\mathbf{G}_{normal}$	$\mathbf{G}_{timeout}$	$\%\mathbf{G}_{total}$	$\mathbf{T}_{gas}$	$\mathbf{B}_{normal}$	$\mathbf{B}_{timeout}$	$\%\mathbf{B}_{total}$	$\mathbf{T}_{bytes}$
syrup$_{cav}$	35689	11129	0.62%	142,93	–	–	–	–
gasol$_{all}$	36344	11975	0.64%	120,21	3712	2213	2.64%	200,17
gasol$_{24}$	38765	12336	0.68%	327,36	4315	2238	2.92%	558,48
gasol$_{none}$	39977	0	0.53%	850,75	3871	0	1.72%	1194,38
gasol$_{best}$	41307	13197	0.72%	933,66	4676	2692	3.28%	1313,36

Table 2: Overall gains in gas and bytes-size and overheads

marked as "–" in the figures. Experiments have been performed on an Intel Core i7-7700T at 4.2GHz x 8 and 64Gb of memory, running Ubuntu 16.04.

The benchmark set. We have downloaded the last 30 verified smart contracts from Etherscan that were compiled using the version 8 of solc and whose source code was available as of June 21, 2021. The reason for this selection is twofold: (1) we require version 8 in order to be able to apply the latest solc optimizer and start from a worst-case scenario in which we have the most possible optimized version and, this way, assess if there is room for further optimization and, in particular, for the two types of gains achievable by GASOL (see Sec. 1), (2) we want to make a random choice (e.g., the last 30) rather than picking up contracts favorable to us. The benchmarks in [7] require using an old version of the compiler (at most 4), hence the last solc optimizer cannot be activated. The source code of GASOL as well as the smart contracts analyzed are available at https://github.com/costa-group/gasol-optimizer. We provide the results of analyzing the compiled smart contracts generated by the version 0.8.9 of solc with the complete optimization options. The total number of blocks, given by **BLK**, for the 30 contracts is 12,378. Within them, there are 1,044 `SSTORE` instructions, 6,631 `MSTORE` and 43 `MSTORE8`. These memory instructions are used by SYRUP to split the basic blocks, while GASOL does not split them always as explained above. This results on 15,416 blocks when considering the additional 24-partitioning, 13,030 without partitioning at stores by gasol$_{none}$, and 20,467 blocks by syrup$_{cav}$ and gasol$_{all}$. As in [7] all tools split blocks at instructions like `LOGX` or `CODECOPY` .

Efficiency gains and performance overhead. Table 2 shows the overall gas and size gains and the optimization time (in minutes). The total gas consumed by all contracts before running the optimizers is 7,538,907, and the bytes-size is 224,540. As it is customary, we are calculating such gas (resp. size) as the sum of the gas (resp. size) consumed by all EVM instructions in the considered contracts.[4] For those EVM instructions that do not consume a constant and fixed amount of gas, such as `SSTORE`, `EXP` or `SHA3`, we choose the lower bound that they may consume. Column $\mathbf{G}_{normal}$ refers to the gains for the blocks that do not timeout giving no solution, $\mathbf{G}_{timeout}$ represents the gas saved by the optimized blocks that reached the timeout in gasol$_{none}$ with no result (note that $\mathbf{G}_{normal}$ is the complementary of $\mathbf{G}_{timeout}$), and $\mathbf{G}_{total}$ the total gains computed as the sum of the previous two, given as a percentage *wrt.* the initial gas consumption. Columns **B** have the analogous meanings for size and **T** gives the time in minutes. The first observation is that our proposal of using dependencies in gasol$_{all}$ pays off, as we achieve larger

[4] Estimating the actual gains of executing transactions on the involved contracts is a research problem on its own which has been subject of other work, e.g., [6,16,19,24].

	#B	$\mathbf{Alr}_g$	$\mathbf{Opt}_g$	$\mathbf{Bet}_g$	$\mathbf{Non}_g$	$\mathbf{Tout}_g$	$\mathbf{Alr}_b$	$\mathbf{Opt}_b$	$\mathbf{Bet}_b$	$\mathbf{Non}_b$	$\mathbf{Tout}_b$
syrup_{cav}	20467	70.54	27.01	0.47	0.08	1.9	–	–	–	–	–
gasol_{all}	20467	70.63	27.36	0.64	0.35	1.02	83.25	12.83	1.2	0.69	2.03
gasol_{24}	15416	62.2	33.79	1.47	0.91	1.63	75.48	16.29	3.21	1.78	3.24
gasol_{none}	13030	65.48	25.3	3.81	0.34	5.07	73.44	11.7	3.1	2.57	9.19

Table 3: Optimization report (%) for SYRUP and GASOL

gains than syrup_{cav} in less time. The second observation is that the gains in gas of GASOL are notably larger for blocks that do not time out $\mathbf{G}_{normal}$, as a larger search space can be explored. However, those blocks that would require a larger timeout might behave worse than the syrup_{cav} and gasol_{all} versions working on smaller blocks, as the original bytecode is taken as the optimization result in case of timeout. This sometimes happens in the version gasol_{24}, and more often in gasol_{none}. The problem is exacerbated for the bytes-size criterion because larger blocks are considered as a result of skipping size-increasing simplification rules. Even in $\mathbf{B}_{normal}$ the gain is smaller for gasol_{none} than for gasol_{24}. This is because $\mathbf{B}_{normal}$ includes timeouts for which a solution is found. Our solution to mitigate the huge computation demands required in these cases is in row gasol_{best} that runs in parallel gasol_{all}, gasol_{24} and gasol_{none} and returns the best result. As it can be seen, gasol_{best} clearly outperforms the other systems in gas and size gains. As regards the overhead, it is also the most expensive option, as it reaches the timeout more often than the other systems and these timeouts are accumulated to the time. However, as superoptimizers are often used as offline optimization tools, which are run only prior to deployment, we argue that the gains compensate the further optimization time. Finally, it remains to be investigated the interaction between the two optimization criteria, namely how the reduction in bytes-size affects the gas consumption and vice versa.

Impact of phases 1 and 2. We would also like to estimate how much is gained in gasol_{best} by applying the simplification rules and how much is gained by the SMT encoding. Regarding the simplification rules on memory, gasol_{best} has applied 6 rules on storage and 11 on memory: 15 of them correspond to the rule i) (4 on storage and 11 on memory) described in Def. 2, and 2 to the rule ii) (both on storage). Rule iii) is never applied on this benchmark set, but we have applied it when optimizing other real smart contracts. As regards the percentage of the gains, 14.6% of the gas savings come from applying the memory rules, 34.4% from the stack rules and 51% is saved by the use of the Max-SMT solver. As in [7], the gains due to each phase are roughly half (i.e., 50% each). Regarding the simplification rules on stack for the gas criterion, their application has increased 11.4% in gasol_{best} because it works on larger blocks and has more opportunities to apply them. However, when selecting the bytes-size criteria, there are less simplification rules applied (namely 96% less) as when the rules generate larger code in terms of size they are not applied (see Sec. 4.3).

Optimality results. Table 3 provides additional detailed information, which is also part of the *optimization report* of Fig. 1. Column **#B** shows the total number of blocks analyzed in each case, depending on the partitioning. In the remaining columns, we show the percentages of: Column **Alr** blocks that are already optimal, i.e., those blocks that cannot be optimized because they already consume the minimal amount of gas; **Opt** blocks that have been optimized and the SMT solver has proved the optimality of the solution, i.e., they consume the minimum amount of gas needed to generate the provided SMS; **Bet** blocks that have been optimized and therefore, consume less gas than the original ones, but the solution is not proved to be optimal; **Non** blocks that have not been optimized and the solver has not been able to prove if they are optimal, i.e., the solution found is the original one but it may exist a better one; **Tout** blocks where the solver reached the timeout without finding a model. The subscripts $_b$ are the analogous for the bytes-size criterion. We can observe in the table that gasol_{none} times out in more cases due to the larger sizes of the blocks that it optimizes, but the percentages of blocks for which it finds a better and optimal solution are notably high. It should also be noted that the results of SYRUP (and gasol_{all}) and, to a lesser extent, of gasol_{24} *wrt.* optimality are weaker. This is because they work on strictly smaller blocks and hence they can prove optimality for the partitioned blocks, but when glued together, the optimality may be lost. This is also the reason why the results for gasol_{best} are not included, because it mixes different notions of optimality and the concepts are not well-defined. Due to this weaker optimality, the **Opt** and **Bet** results are only slightly better for GASOL than for SYRUP. However, the truly important aspect is that the actual gas and size gains for GASOL in Table 2 are notably larger.

6 Conclusions and Future Work

We have presented GASOL^{v2}, a Max-SMT based superoptimizer for Ethereum smart contracts that uses the assembly json exchange format of the solc compiler for a direct integration into it. GASOL^{v2} extends the Max-SMT approach of SYRUP [7] with memory and storage operations, which constitute the most challenging and relevant features left out in SYRUP's approach. GASOL^{v2} is part of the GASOL project [3] that aims at developing a GAS Optimization tooLkit that will integrate *inter-block* optimizations [5] as well. Namely, the initial optimizer [5] of the GASOL project uses inter-block analysis to detect storage accesses that can be replaced by cheaper memory accesses, thus making global optimizations that are orthogonal and complementary to our intra-block ones. As part of our future work, we plan to investigate potential synergies among the different proposals to optimization for smart contracts. This includes also the cooperation with the solc optimizer [4] that incorporates classical compiler optimizations (e.g., dead code elimination, constant propagation, etc.) from which our superoptimizer is already benefiting (since we are applying the solc optimizer). In the other order of application, we expect also gains when applying classical analyses after superoptimization. For instance, we have also observed that after applying rule simplification (i) in Def. 2 and eliminating load instructions, we might leave store operations on memory locations that will never be accessed again, and that could be eliminated afterwards by applying an inter-block analysis ensuring that there are no further access to such memory location. The combination of the techniques and tools thus seems a promising direction for future research.

References

1. Compiler Input and Output JSON Description. https://docs.soliditylang.org/en/v0.8.7/using-the-compiler.html#compiler-input-and-output-json-description.
2. Welfare contract. https://etherscan.io/address/0x3E873439949793e8c577E08629c36Ed8c184e7D9#code.
3. GASOL – A GAS Optimization tooLkit, 2021. Funded by the Ethereum Foundation https://blog.ethereum.org/2021/07/01/esp-allocation-update-q1-2021/.
4. The solc optimizer, 2021. https://docs.soliditylang.org/en/v0.8.7/internals/optimizer.html.
5. Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts. In Armin Biere and David Parker, editors, *Proceedings of 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2020*, volume 12079 of *Lecture Notes in Computer Science*, pages 118–125, 2020.
6. Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. Don't run on fumes—parametric gas bounds for smart contracts. *Journal of Systems and Software*, 176:110923, 2021.
7. Elvira Albert, Pablo Gordillo, Albert Rubio, and Maria A. Schett. Synthesis of super-optimized smart contracts using max-smt. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 177–200. Springer, 2020.
8. Elvira Albert, Pablo Gordillo, Albert Rubio, and Ilya Sergey. Running on Fumes: Preventing Out-Of-Gas Vulnerabilities in Ethereum Smart Contracts using Static Resource Analysis. In *13th International Conference on Verification and Evaluation of Computer and Communication Systems, VECoS 2019. Proceedings*, volume 11847 of *LNCS*, pages 63–78. Springer, 2019.
9. Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8. IEEE, 2013.
10. Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In John Paul Shen and Margaret Martonosi, editors, *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*, pages 394–403. ACM, 2006.
11. Sorav Bansal and Alex Aiken. Binary translation using peephole superoptimizers. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 177–192. USENIX Association, 2008.
12. James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with metasketches. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 775–788. ACM, 2016.

13. Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013. Proceedings*, pages 93–107, 2013.
14. F. Contro, M. Crosara, M. Ceccato, and M. Dalla Preda. Ethersolve: Computing an accurate control-flow graph from ethereum bytecode. In *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021*, pages 127–137. IEEE, 2021.
15. Torbjörn Granlund and Richard Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. In Stuart I. Feldman and Richard L. Wexelblat, editors, *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI), San Francisco, California, USA, June 17-19, 1992*, pages 341–352. ACM, 1992.
16. Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. MadMax: surviving out-of-gas conditions in Ethereum smart contracts. *PACMPL*, 2(OOPSLA):116:1–116:27, 2018.
17. Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 62–73. ACM, 2011.
18. Abhinav Jangda and Greta Yorsh. 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*, pages 78–88, 2017.
19. Matteo Marescotti, Martin Blicha, Antti E. J. Hyvärinen, Sepideh Asadi, and Natasha Sharygina. Computing Exact Worst-Case Gas Consumption for Smart Contracts. In *ISoLA*, volume 11247 of *LNCS*, pages 450–465. Springer, 2018.
20. Henry Massalin. 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)*, pages 122–126, 1987.
21. Julian Nagele and Maria A Schett. Blockchain superoptimizer. In *Preproceedings of 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2019)*, 2019. https://arxiv.org/abs/2005.05912.
22. Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. Souper: A Synthesizing Superoptimizer. *arXiv:1711.04422 [cs]*, November 2017.
23. Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2019.
24. Renlord Yang, Toby Murray, Paul Rimba, and Udaya Parampalli. Empirically analyzing ethereum's gas mechanism. In *2019 IEEE European Symposium on Security and Privacy Workshops, EuroS&P Workshops 2019, Stockholm, Sweden, June 17-19, 2019*, pages 310–319, 2019.