Speeding Up Machine-Code Synthesis *

Venkatesh Srinivasan
University of Wisconsin–Madison, USA
venk@cs.wisc.edu

Tushar Sharma
University of Wisconsin–Madison, USA
tsharma@cs.wisc.edu

Thomas Reps
University of Wisconsin–Madison and Grammatech, Inc., USA
reps@cs.wisc.edu

Abstract

Machine-code synthesis is the problem of searching for an instruction sequence that implements a semantic specification, given as a formula in quantifier-free bit-vector logic (QFBV). Instruction sets like Intel's IA-32 have around 43,000 unique instruction schemas; this huge instruction pool, along with the exponential cost inherent in enumerative synthesis, results in an enormous search space for a machine-code synthesizer: even for relatively small specifications, the synthesizer might take several hours or days to find an implementation. In this paper, we present several improvements to the algorithms used in a state-of-the-art machine-code synthesizer MCSYNTH. In addition to a novel pruning heuristic, our improvements incorporate a number of ideas known from the literature, which we adapt in novel ways for the purpose of speeding up machine-code synthesis. Our experiments for Intel's IA-32 instruction set show that our improvements enable synthesis of code for 12 out of 14 formulas on which MCSYNTH times out, speeding up the synthesis time by at least 1981X, and for the remaining formulas, speeds up synthesis by 3X.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; I.2.2 [*Artificial Intelligence*]: Automatic Programming – Program Synthesis

Keywords Machine-code synthesis, flow independence, flattening deep terms, pruning heuristics, move-to-front heuristic, IA-32 instruction set

1. Introduction

Binary analysis and rewriting has received an increasing amount of attention from the academic community in the last decade (e.g., see references in [24, §7], [3, §1], [6, §1], [8, §7]), which has led to the development and wider use of binary analysis and rewriting tools. Recently [26], it has been observed that a machine-code synthesizer[1] can play a key role in several such tools, including partial evaluators [25], slicers [27], and superoptimizers. A machine-code synthesizer is a tool that synthesizes a straight-line instruction sequence that implements a specification, which is often given as a formula in QFBV.

A key challenge in synthesizing machine code from a specification is the enormous size of the synthesis search-space: for example, Intel's IA-32 instruction-set architecture (ISA) has around 43,000 unique instruction schemas; this huge instruction pool, along with the exponential cost inherent in enumerative synthesis, results in an enormous search-space for a synthesizer.

To cope with the enormous synthesis search-space, a state-of-the-art machine-code synthesizer MCSYNTH [26] uses (i) a *divide-and-conquer* strategy to split the synthesis task into several independent smaller sub-tasks, and (ii) *footprint-based* search-space pruning heuristics to prune away candidates during synthesis. However, MCSYNTH times out for several larger QFBV formulas; even for smaller formulas, MCSYNTH takes several minutes to find an implementation. Consequently, if a binary-rewriter client supplies a formula as input to MCSYNTH, the client has to wait several minutes or hours before MCSYNTH finds an implementation. This delay might not be tolerable for a client that has to invoke the synthesizer multiple times to rewrite an entire binary. For example, the machine-code partial evaluator WIPER [25] uses MCSYNTH for the purpose of residual-code generation. For a small binary, WIPER calls MCSYNTH tens to hundreds of times. If each formula supplied to MCSYNTH is relatively large, partial evaluation of an entire binary might take several hours or days.

* Supported, in part, by a gift from Rajiv and Ritu Batra; by AFRL under DARPA MUSE award FA8750-14-2-0270, and DARPA STAC award FA8750-15-C-0082; and by the UW-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies. T. Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology reported in this publication.

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4444-9/16/11...$15.00
http://dx.doi.org/10.1145/2983990.2984006

[1] We use the term "machine code" to refer generically to low-level code, and do not distinguish between the actual machine-code bits/bytes and the assembly code to which it is disassembled.

In this paper, we present several techniques to improve the synthesis algorithm used in McSynth. Some of our techniques improve the "divide" phase of McSynth so that McSynth can better split a synthesis task into smaller sub-tasks; the remaining techniques improve the "conquer" phase of McSynth so that McSynth can find an implementation for a sub-task faster. Our techniques are not restricted to machine code in particular, and can be applied to speed up other enumerative program synthesizers as well. We have implemented our techniques in McSynth to create a newer version of it called McSynth++. Like McSynth, McSynth++ can be parameterized by the instruction set of the target instruction-sequence.

Because McSynth++ is much faster than McSynth (see §6), McSynth++ should improve the speed of existing binary-rewriting clients that use McSynth [25, 27]. McSynth++ should also allow existing clients to work on larger QFBV formulas for the purpose of obtaining output binaries of better quality. For example, WiPEr currently specializes individual instructions with respect to static inputs. With McSynth++, instead of residuating code for several specialized instructions in a basic block, WiPEr could perform specialization at a basic-block level, and thereby create more optimized code. McSynth++ should also facilitate building of new clients that were impractical to build with McSynth.

McSynth++ improves the "divide" phase of McSynth by addressing the two principal limitations of McSynth's algorithm in the following manner:

- McSynth treats memory conservatively while attempting to split the input formula into independent sub-formulas. McSynth++ reasons about individual memory locations, and thus allows splits that were conservatively discarded by McSynth.
- If a sub-formula has a "deep" term (a term with a deep abstract-syntax tree), McSynth does not attempt to split the synthesis task into sub-tasks. McSynth++ flattens the deep term into a sequence of sub-terms, synthesizes code for the sub-terms, and stitches the code fragments in the correct order to obtain the final implementation.

McSynth++ improves the "conquer" phase of McSynth via an additional pruner: in addition to the footprint-based pruner in McSynth, McSynth++ prunes away candidates based on the pre-state bits lost/destroyed by a candidate instruction-sequence when it transforms a state. Additionally for pragmatic purposes, McSynth++ uses a "move-to-front" heuristic that moves instructions that occur in synthesized code to the front of the instruction pool for the next synthesis task. This heuristic enables McSynth++ to find more quickly implementations of input formulas in terms of more common instructions. Collectively, our techniques work together to significantly reduce synthesis time, sometimes by a factor of over 1981X.

Our techniques incorporate a number of ideas known from the literature, including

- alias testing to check for flow independence between elements of an array [10, 17],
- flattening of an abstract-syntax tree (AST) via scratch-register allocation for code-generation purposes [1],
- the observation that during synthesis, it is advantageous to try out sooner the components that are more frequently used in a codebase [20, 21].

In this paper, we use/adapt these ideas in novel ways for the purpose of speeding up machine-code synthesis.

Contributions. The paper's contributions include the following:

- We show how ideas related to array dependence-testing furnish a better method for splitting a formula into independent sub-formulas for the purposes of synthesis (§4.1.2.1).
- We show how one can use flattening and scratch locations to convert a single large synthesis task involving a "deep" term into multiple smaller synthesis tasks involving "flat" terms (§4.1.2.2).
- We propose a novel way of pruning candidates and prefixes during synthesis based on information about the pre-state bits lost/destroyed by an instruction sequence when it transforms a state (§4.2.2).
- We show how a simple "move-to-front" heuristic can be used to prioritize instructions that are commonly used to implement operations in an instruction set (§4.3).

Our techniques have been implemented in McSynth++, an improved synthesizer for IA-32. Our experiments show that McSynth++ synthesizes code for 12 out of 14 formulas on which McSynth times out, speeding up the synthesis time by at least 1981X, and for the remaining formulas, McSynth++ speeds up synthesis by 3X.

2. Background

In this section, we briefly describe the logic in which input formulas are expressed (§2.1). The logic allows a client to specify some desired state change in a specific hardware platform—in our case, Intel IA-32 (also known as x86). We also give an overview of a state-of-the-art machine-code synthesizer McSynth: we briefly describe how binary-rewriter clients use McSynth to rewrite binaries, and summarize the algorithm used in McSynth (§2.2).

2.1 QFBV Formulas for Expressing Specifications

Input specifications to McSynth++ can be expressed formally by QFBV formulas. Consider a quantifier-free bit-vector logic L over finite vocabularies of constant symbols and function symbols. We will be dealing with a specific instantiation of L, denoted by L[IA-32]. (L can also be instantiated for other ISAs.) In L[IA-32], some constants represent IA-32's registers (*EAX*, *ESP*, *EBP*, etc.), some represent flags (*CF*, *SF*, etc.), and some are free constants (*m*, *n*, *x*, *y*, etc.). L[IA-32] has only one function symbol "*Mem*," which

$$T \in \mathit{Term},\ \varphi \in \mathit{Formula},\ \mathit{FE} \in \mathit{FuncExpr}$$
$$c \in \mathit{Int32} = \{\ldots, \text{-}1, 0, 1, \ldots\} \quad b \in \mathit{Bool} = \{\mathit{True}, \mathit{False}\}$$
$$I_{\mathit{Int32}} \in \mathit{Int32Id} = \{\mathit{EAX}, \mathit{ESP}, \mathit{EBP}, \ldots, m, n, \ldots\}$$
$$I_{\mathit{Bool}} \in \mathit{BoolId} = \{\mathit{CF}, \mathit{SF}, \ldots, x, y, \ldots\}$$
$$F \in \mathit{FuncId} = \{\mathit{Mem}\}$$
$$op \in \mathit{BinOp} = \{+, -, \ldots\} \quad bop \in \mathit{BoolOp} = \{\wedge, \vee, \ldots\}$$
$$rop \in \mathit{RelOp} = \{=, \neq, <, >, \ldots\}$$
$$T ::= c \mid I_{\mathit{Int32}} \mid T_1\, op\, T_2 \mid ite(\varphi, T_1, T_2) \mid F(T_1)$$
$$\varphi ::= b \mid I_{\mathit{Bool}} \mid T_1\, rop\, T_2 \mid \neg\varphi_1 \mid \varphi_1\, bop\, \varphi_2 \mid F = \mathit{FE}$$
$$\mathit{FE} ::= F \mid \mathit{FE}_1[T_1 \mapsto T_2]$$

Figure 1: Syntax of L[IA-32].

denotes memory. The syntax of L[IA-32] is defined in Fig. 1. A term of the form $ite(\varphi, T_1, T_2)$ represents an if-then-else expression. A *FuncExpr* of the form $\mathit{FE}[T_1 \mapsto T_2]$ denotes a *function-update* expression.

To write formulas that express state transitions, all *Int32Id*s, *BoolId*s, and *FuncId*s can be qualified by primes (e.g., *Mem′*). The QFBV formula for a specification is a restricted 2-vocabulary formula that specifies a state transformation. It has the form

$$\bigwedge_m (I'_m = T_m) \wedge \bigwedge_n (J'_n = \varphi_n) \wedge \mathit{Mem}' = \mathit{FE},$$

where I'_m and J'_n range over the constant symbols for registers and flags, respectively. The primed vocabulary is the post-state vocabulary, and the unprimed vocabulary is the pre-state vocabulary. For example, the QFBV formula for the specification "push the 32-bit value in the frame-pointer register EBP onto the stack" is given below. (Note that the IA-32 stack pointer is register ESP.)

$$\mathit{ESP}' = \mathit{ESP} - 4 \ \wedge\ \mathit{Mem}' = \mathit{Mem}[\mathit{ESP} - 4 \mapsto \mathit{EBP}]$$

In this section, and in the rest of the paper, we show only the portions of QFBV formulas that express how the state is *modified*. QFBV formulas actually contain identity conjuncts of the form $I' = I$, $J' = J$, and $\mathit{Mem}' = \mathit{Mem}$ for constants and functions that are *unmodified*. Because we do not want the synthesizer output to be restricted to an instruction sequence that is located at a specific address, specifications do not contain conjuncts of the form $\mathit{EIP}' = T$. (EIP is the program counter for IA-32.)

Expressing semantics of instruction sequences. In addition to input specifications, MCSYNTH++ also uses L[IA-32] formulas to express the semantics of the candidate instruction-sequences it considers. The function $\langle\langle \cdot \rangle\rangle$ encodes a given IA-32 instruction-sequence as a QFBV formula. While others have created such encodings by hand (e.g., [22]), we use a method that takes a specification of the concrete operational semantics of IA-32 instructions and creates a QFBV encoder automatically. The method reinterprets each semantic operator as a QFBV formula-constructor or term-constructor (see [15]).

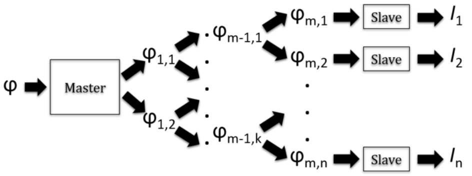

Figure 2: Master-slave architecture of MCSYNTH.

Certain IA-32 string instructions contain an implicit microcode loop, e.g., instructions with the `rep` prefix, which perform an *a priori* unbounded amount of work determined by the value in the ECX register at the start of the instruction. In other applications that use the infrastructure on which MCSYNTH++ is built, this implicit microcode loop is converted into an explicit loop whose body is an instruction that performs the actions performed by the body of the microcode loop. (More details about this conversion is available elsewhere [15, §6].) However, the semantics cannot be expressed as a single QFBV formula. Because of this expressibility limitation, neither MCSYNTH nor MCSYNTH++ tries to synthesize instructions that use the `rep` prefix.

2.2 Machine-Code Synthesis using MCSYNTH

In this section, we give an overview of a state-of-the-art machine-code synthesizer MCSYNTH [26]. We also summarize MCSYNTH's algorithm while highlighting its key limitations.

MCSYNTH synthesizes a straight-line machine-code instruction sequence from a semantic specification of the desired behavior, given as a QFBV formula. The synthesized instruction-sequence implements the input formula (i.e., is equivalent to the formula). MCSYNTH is parameterized by the ISA of the target instruction-sequence.

Binary rewriters transform binaries via semantic transformations. Using MCSYNTH, one can create multiple binary rewriters using the following recipe:

- convert instructions in the binary to QFBV formulas,
- use analysis results to transform QFBV formulas, and
- use MCSYNTH to produce an instruction sequence that implements each transformed formula.

Examples of semantics-based binary rewriters that can be created using the above recipe include offline optimizers, partial evaluators [25], slicers [27], and binary translators [5]. For example, the machine-code partial evaluator WIPER [25] specializes the QFBV formulas of instructions with respect to a static partial state, and uses MCSYNTH to produce residual instructions from the specialized formulas.

MCSYNTH uses enumerative strategies for synthesis. However, an ISA like IA-32 has around 43,000 unique instruction schemas, which, when combined with the exponential cost inherent in enumeration, results in an enormous search space for synthesis. MCSYNTH attempts to cope with the enormous search space using a *master-slave* architecture. The design of MCSYNTH is depicted in Fig. 2. Given a QFBV formula φ, McSynth synthesizes an instruction sequence for φ in the following way:

1. The master uses a *divide-and-conquer* strategy to split φ into independent sub-formulas, and hands over each sub-formula to a slave synthesizer.
2. The slave uses enumeration, along with an instantiation of the *counterexample-guided inductive synthesis* (CEGIS) framework and *footprint-based* search-space pruning heuristics to synthesize code for a sub-formula.
3. If a slave times out, the master uses an alternative split. (For example in Fig. 2, if synthesis of code for $\varphi_{\text{m, 1}}$ times out, the master tries out an alternative split for $\varphi_{\text{m-1, 1}}$.) If all candidate splits for a sub-formula time out, the master hands over the entire sub-formula to a slave. (For example in Fig. 2, if all candidate splits of $\varphi_{\text{m-1, 1}}$ time out, the master supplies $\varphi_{\text{m-1, 1}}$ as an input to a slave.)
4. The master concatenates the results produced by the slaves, and returns the final instruction sequence.

In the remainder of this section, we present an example to illustrate McSynth's algorithm.

Consider the following QFBV formula φ:

$$\varphi \equiv \mathit{EAX}' = \mathit{Mem}(\mathit{ESP}+4) \wedge \mathit{Mem}' = \mathit{Mem}[\mathit{ESP} \mapsto \mathit{EAX}] \wedge$$
$$\mathit{EBX}' = ((\mathit{EAX} * 2) \gg 2) + \mathit{EAX}$$

φ performs three updates on an IA-32 state: (i) it copies the 32-bit value in the memory location pointed to by ESP + 4 to the EAX register, (ii) copies the 32-bit value in the EAX register to the memory location pointed to by the stack-pointer register ESP, and (iii) updates the EBX register with a value computed using the value in the EAX register.

The divide-and-conquer strategy tries to split the updates in φ across a sequence of sub-formulas $\langle \varphi_1, \varphi_2, \ldots, \varphi_k \rangle$ such that if one were to synthesize an instruction sequence I_i for each φ_i *independently*, and concatenate the synthesized instruction-sequences in the same order, the result will be equivalent to φ. Such a split is called a *legal* split. A sufficient condition for a legal split is *flow independence*—if we can split the updates in φ across the sequence $\langle \varphi_1, \varphi_2, \ldots, \varphi_k \rangle$ such that there is no flow dependence from φ_i to any successor sub-formula φ_j $(i < j)$, the split is legal. The reason is as follows: because I_j is equivalent to some sub-formula φ_j of φ, and I_j does not read any of the locations that could be modified by any predecessor instruction-sequence I_i $(i < j)$, $I_1; I_2; \ldots; I_k$ performs the same state transformations as φ.

McSynth uses a very conservative one-sided decision procedure to check if a split $\langle \varphi_1, \varphi_2, \ldots, \varphi_k \rangle$ is flow-independent. The decision procedure returns MAYBE if either (or both) of the following conditions hold:

C1: φ_j might use a register or flag, and a predecessor sub-formula φ_i $(i < j)$ might modify the same register or flag

C2: φ_j might use *some* memory location, and a predecessor sub-formula φ_i $(i < j)$ might modify *some* memory location

If neither of the conditions hold, the decision procedure returns NO (meaning the split is flow independent). For our example, divide-and-conquer identifies $\langle \varphi_1, \varphi_2 \rangle$ as a legal split of φ, where φ_1 and φ_2 are defined below.

$$\varphi_1 \equiv \mathit{EBX}' = ((\mathit{EAX} * 2) \gg 2) + \mathit{EAX}$$
$$\varphi_2 \equiv \mathit{EAX}' = \mathit{Mem}(\mathit{ESP} + 4) \wedge \mathit{Mem}' = \mathit{Mem}[\mathit{ESP} \mapsto \mathit{EAX}]$$

However, because the decision procedure returns MAYBE for the split given below, divide-and-conquer does not identify $\langle \varphi_3, \varphi_4 \rangle$ as a legal split, although it actually is one.

$$\varphi_3 \equiv \mathit{Mem}' = \mathit{Mem}[\mathit{ESP} \mapsto \mathit{EAX}]$$
$$\varphi_4 \equiv \mathit{EAX}' = \mathit{Mem}(\mathit{ESP}+4) \wedge \mathit{EBX}' = ((\mathit{EAX}*2) \gg 2) + \mathit{EAX}$$

One can see that condition C2 is overly conservative: φ_3 might update the value in a memory location (*Mem*(*ESP*)) that is always disjoint from memory location that might be used by φ_4 (*Mem*(*ESP* + *4*)). Consequently, $\langle \varphi_3, \varphi_4 \rangle$, and many other legal splits, are missed by McSynth.

McSynth's master produces the legal split $\langle \varphi_1, \varphi_2 \rangle$ of φ. McSynth cannot split either φ_1 or φ_2 any further, so it hands over φ_1 and φ_2, respectively, to slave synthesizers.

Given a sub-formula φ_i and—for pragmatic reasons—a timeout value, McSynth's slave synthesizer either synthesizes an instruction sequence that implements φ_i, or returns FAIL if it could not find such an instruction sequence before the timeout expires. The slave synthesizes an instruction sequence for φ_i using the following method:

1. The slave enumerates *templatized* instruction-sequences of increasing length. A templatized instruction-sequence is a sequence of instructions with template operands (or holes) instead of one or more constant values.
2. The slave attempts to find an instantiation of a candidate templatized instruction-sequence that is logically equivalent to φ_i using CEGIS. If an instantiation is found, the slave returns it. Otherwise, the next templatized sequence is considered.
3. The slave uses heuristics based on the *footprints* of QFBV formulas (see below) to prune away useless candidates during enumeration.

In the remainder of this section, we illustrate the working of a slave synthesizer using φ_1 as an example.

The slave starts enumerating templatized one-instruction sequences. To prune away candidates during enumeration, McSynth uses an overapproximation of locations that could potentially be touched by a formula. We define the *abstract semantic use-footprint* $\text{SFP}^{\#}{}_{\text{USE}}$ (*kill-footprint* $\text{SFP}^{\#}{}_{\text{KILL}}$) as an overapproximation of the locations used (modified) by a formula. Concretely, an abstract semantic-footprint of a formula φ is a subset of the set of constant symbols in φ and a special symbol "*Mem*," which denotes the entire memory. The symbols in $\text{SFP}^{\#}{}_{\text{KILL}}$ are primed.

Let us assume that the first candidate enumerated by the slave is $\text{C}_1 \equiv$ "`mov eax, ⟨Imm32⟩`." The formula ψ_1 for C_1 is $\mathit{EAX}' = m$. The abstract semantic footprints for φ_1 and ψ_1, respectively, are given below.

$$\mathrm{SFP}^{\#}{}_{\mathrm{USE}}(\varphi_1) = \{\mathit{EAX}\} \quad \mathrm{SFP}^{\#}{}_{\mathrm{KILL}}(\varphi_1) = \{\mathit{EBX}'\}$$
$$\mathrm{SFP}^{\#}{}_{\mathrm{USE}}(\psi_1) = \{\} \quad \mathrm{SFP}^{\#}{}_{\mathrm{KILL}}(\psi_1) = \{\mathit{EAX}'\}$$

One can see that the abstract semantic KILL-footprint of ψ_1 is outside that of φ_1, and thus C_1 can never implement φ_1 without possibly modifying a value in a location that is otherwise unmodified by φ_1. Therefore, the slave prunes away C_1 because it is a *useless candidate*.

Suppose that the next candidate is $C_2 \equiv$ "`mov ebx, [eax]`." The formula ψ_2 for C_2 is $\mathit{EBX}' = \mathit{Mem(EAX)}$. The abstract semantic footprints for ψ_2 are given below.

$$\mathrm{SFP}^{\#}{}_{\mathrm{USE}}(\psi_1) = \{\mathit{EAX}, \mathit{Mem}\} \quad \mathrm{SFP}^{\#}{}_{\mathrm{KILL}}(\psi_1) = \{\mathit{EBX}'\}$$

The abstract semantic USE-footprint of ψ_2 is outside that of φ_1 because ψ_2 might use some memory location, but φ_1 does not use any memory location. Therefore, the slave also prunes away the useless candidate C_2.

Suppose that the slave has exhausted all one-instruction sequences, and the next candidate is $C_3 \equiv$ "`lea ebx, [eax + eax]; lea ebx, [ebx + ⟨Imm32⟩]`." The formula ψ_3 for C_3 is $\mathit{EBX}' = 2 * \mathit{EAX} + m$. The abstract semantic USE/KILL-footprints of ψ_3 are within those of φ_1. So the slave uses a CEGIS-based loop to check if there exists an instantiation of the candidate that implements φ_1. The CEGIS loop says that no such instantiation exists, and so the slave discards C_3.

The slave eventually enumerates the candidate C_4 $\equiv$"`imul ebx, eax, ⟨Imm32⟩; shr ebx, ⟨Imm32⟩; lea ebx, [ebx + eax]`." (For this example, we pretend that the `shr` instruction does not set flags.) The slave returns the instantiation "`imul ebx, eax, 2; shr ebx, 2; lea ebx, [ebx + eax]`" of C_4 as the synthesized code for φ_1. Because of the depth of the AST of φ_1, and the large number of templatized instructions in IA-32 (around 43,000), the slave takes a few days to synthesize this instruction sequence for φ_1 via enumerative synthesis.

In a similar manner, the slave synthesizes the instruction sequence "`mov [esp],eax; mov eax,[esp+4]`" for φ_2. McSynth concatenates the results produced by the slaves and returns the resulting instruction sequence. McSynth takes a few days to complete the overall synthesis task.

In summary, McSynth uses a divide-and-conquer strategy in combination with footprint-based search-space pruning to combat the exponential cost of enumerative synthesis. However, McSynth suffers from the following limitations:

1. McSynth's one-sided decision procedure for flow independence treats memory conservatively. Consequently, McSynth loses opportunities to find legal splits.
2. If a synthesis task involves a "deep" term (a term whose AST is deep), McSynth does not attempt to split the task into smaller sub-tasks.

The overall effect of these limitations is the high synthesis time for even relatively small QFBV formulas.

3. Overview

At a high level, McSynth++ has the same design as McSynth: McSynth++'s master splits the input QFBV formula into a sequence of independent sub-formulas, and hands over each sub-formula to a slave synthesizer. However, McSynth++'s master and slave are improved versions of those of McSynth. McSynth++'s master uses an improved divide-and-conquer strategy that addresses the limitations of McSynth in the following ways:

1. McSynth++ uses an improved one-sided decision procedure for flow independence. McSynth++'s decision procedure is capable of reasoning about flow independence between different memory locations.
2. If the input formula contains a conjunct with a "deep" term/sub-formula, McSynth++ flattens the deep term/sub-formula into a sequence of sub-formulas.

Compared to the master used in McSynth, the improved master identifies more and finer-grained legal splits. Each of the sub-formulas in a split is given to a slave synthesizer, which synthesizes an instruction sequence for the sub-formula. McSynth++'s improved slave uses an additional pruner based on the bits lost/destroyed by a candidate instruction-sequence to prune away candidates during synthesis. Additionally, the slave uses a "move-to-front" heuristic to boost the priority of instructions that have already been used in synthesized code. McSynth++'s master concatenates the results produced by the slaves, and returns the concatenated instruction-sequence as the synthesized code. Because of the aforementioned improvements, McSynth++ typically finishes the synthesis task much faster than McSynth.[2]

This section presents an example to illustrate the workings of McSynth++. Consider the same QFBV formula φ that was used in §2.2 to illustrate McSynth's algorithm.

$$\varphi \equiv \mathit{EAX}' = \mathit{Mem}(\mathit{ESP}{+}4) \wedge \mathit{Mem}' = \mathit{Mem}[\mathit{ESP} \mapsto \mathit{EAX}] \wedge$$
$$\mathit{EBX}' = ((\mathit{EAX} * 2) \gg 2) + \mathit{EAX}$$

McSynth++'s master first flattens the $\mathit{EBX}' = \ldots$ conjunct into a sequence of conjuncts using *interface* constants. An interface constant is a symbolic constant that facilitates the flow of data between conjuncts created by McSynth++. Each interface constant will be replaced by a concrete location (register, flag, or memory location) in a later step. For example, McSynth++ rewrites φ as φ' by adding interface constants m and n.

$$\begin{aligned} \varphi' \equiv\ & \mathit{EAX}' = \mathit{Mem}(\mathit{ESP} + 4) \wedge \\ & \mathit{Mem}' = \mathit{Mem}[\mathit{ESP} \mapsto \mathit{EAX}] \wedge \\ & m = \mathit{EAX} * 2 \wedge n = m \gg 2 \wedge \mathit{EBX}' = n + \mathit{EAX} \end{aligned} \tag{1}$$

Note that φ' and φ are equisatisfiable. (They are *equisatisfiable* instead of *equivalent* because the vocabulary of φ'

[2] Note that McSynth++ is not guaranteed to be faster because it has more legal splits to consider.

contains extra constant symbols. However, if we disregard these constants in the meaning of φ', then $\varphi' \Leftrightarrow \varphi$.)

MCSYNTH++'s master now uses the improved divide-and-conquer strategy to identify legal splits of φ'. (Recall from §2.2 that a legal split splits the updates in φ' across a sequence of sub-formulas $\langle \varphi_1, \varphi_2, \ldots, \varphi_k \rangle$ such that if one were to synthesize instructions for each φ_i independently, and concatenate the synthesized instruction-sequences in the same order, the result will be equivalent to φ'.) Because the updates in φ' contain interface constant-symbols, flow independence is no longer a sufficient condition for a legal split. If the following two conditions hold, then a split $\langle \varphi_1, \varphi_2, \ldots, \varphi_k \rangle$ is legal:

- *Flow independence* for registers, flags, and memory locations: there is no flow dependence through registers, flags, or memory locations from a sub-formula φ_i to any successor sub-formula φ_j $(i < j)$
- *Mandatory flow dependence* for interface constants: each interface constant that gets used in a sub-formula φ_j is defined in some predecessor sub-formula φ_i $(i < j)$

Given a candidate split, it is straightforward to check the second property. To check the first property, MCSYNTH++ uses an improved one-sided decision procedure. We illustrate the improved decision-procedure using examples. Consider the candidate split $\langle \varphi_1, \varphi_2 \rangle$ given below.

$$\varphi_1 \equiv m = EAX * 2 \;\wedge\; n = m \gg 2 \;\wedge\; EBX' = n + EAX \;\wedge$$
$$EAX' = Mem(ESP + 4)$$
$$\varphi_2 \equiv Mem' = Mem[ESP \mapsto EAX]$$

The decision procedure for flow independence first checks whether condition C1 from §2.2 holds. If it does, the decision procedure returns MAYBE; otherwise, it moves to the next step. C1 holds for $\langle \varphi_1, \varphi_2 \rangle$ because of register EAX, and so the decision procedure returns MAYBE. (Note that the one-sided decision procedure in MCSYNTH also returns MAYBE for this split.)

Consider another candidate split $\langle \varphi_3, \varphi_4 \rangle$ given below.

$$\varphi_3 \equiv Mem' = Mem[ESP \mapsto EAX]$$
$$\varphi_4 \equiv m = EAX * 2 \;\wedge\; n = m \gg 2 \;\wedge\; EBX' = n + EAX \;\wedge$$
$$EAX' = Mem(ESP + 4)$$

C1 does not hold for $\langle \varphi_3, \varphi_4 \rangle$. The improved decision procedure now collects the terms that denote the addresses of memory locations that might be modified by φ_3 (denoted by MemUpdateTerms(φ_3)), and the set of terms that denote the addresses of memory locations that might be used by φ_4 (denoted by MemAccessTerms(φ_4)). The sets are given below.

$$\text{MemUpdateTerms}(\varphi_3) = \{ESP\}$$
$$\text{MemAccessTerms}(\varphi_4) = \{ESP + 4\}$$

If any term T_1 in MemUpdateTerms might alias with any term T_2 in MemAccessTerms, the decision procedure returns MAYBE; otherwise, it returns NO. The decision procedure checks if T_1 and T_2 might be aliases of each other by testing the satisfiability of $T_1 = T_2$. In our example, $ESP = ESP{+}4$ is UNSAT, so the decision procedure returns NO, meaning that there is no flow dependence from φ_3 to φ_4. (Note that the one-sided decision procedure in MCSYNTH returns MAYBE for this split.)

The master recursively splits φ_4 in a similar manner. Ultimately, the master splits φ into the following sequence of sub-formulas:

$$\varphi_3 \equiv Mem' = Mem[ESP \mapsto EAX] \quad \varphi_5 \equiv m' = EAX * 2$$
$$\varphi_6 \equiv n' = m \gg 2 \qquad \varphi_7 \equiv EBX' = n + EAX$$
$$\varphi_8 \equiv EAX' = Mem(ESP + 4)$$

Note that when the two occurences of an interface constant are put in different sub-formulas of a split, the occurence on the left of the = operator is primed. The master adds the primes when it enumerates the candidate splits of a formula.

Before giving the sub-formulas in a legal split to slave synthesizers, MCSYNTH++'s master assigns concrete locations to the interface constants. We illustrate how MCSYNTH++ assigns concrete locations to the split $\langle \varphi_3, \varphi_5, \varphi_6, \varphi_7, \varphi_8 \rangle$. (Note that concrete-location assignment is done on a per-split basis.) Suppose that the registers {EAX, EBX} are dead at the point where code is to be synthesized, and these registers are supplied as scratch registers to MCSYNTH++. MCSYNTH++ can assign an interface constant any scratch register that is definitely not used in a downstream sub-formula. For example, MCSYNTH++ cannot assign *m* the register EAX because EAX might be used in φ_7. If MCSYNTH++ were to assign *m* the register EAX, then it introduces a flow dependence that was originally not present in the split, making the split illegal. So MCSYNTH++ assigns *m* the register EBX. MCSYNTH++ also assigns the interface constant *n* the register EBX. The sub-formulas φ_5, φ_6, and φ_7 after register assignment are given below.

$$\varphi_5 \equiv EBX' = EAX * 2 \qquad \varphi_6 \equiv EBX' = EBX \gg 2$$
$$\varphi_7 \equiv EBX' = EBX + EAX$$

MCSYNTH++ supplies the sub-formulas as inputs to the improved slave-synthesizers. (Note that, as in MCSYNTH, if synthesis for any sub-formula in a split times out, MCSYNTH++'s master tries an alternative split; if MCSYNTH++ fails to synthesize code for all candidate splits, the entire formula is given to a slave.) In the remainder of this sub-section, we illustrate the working of the improved slave-synthesizer using φ_7 as an example.

MCSYNTH++'s slave—just like MCSYNTH's slave—enumerates templatized instruction-sequences of increasing length, and prunes away candidates based on abstract semantic-footprints. However unlike MCSYNTH, candidates enumerated by MCSYNTH++ pass through an additional *bits-lost-based* pruner before reaching the CEGIS loop.

The role of the bits-lost-based pruner is illustrated by the following example: suppose that the slave is currently enumerating templatized one-instruction candidates, and the current candidate is $C_1 \equiv$ "`mov ebx, eax`." The formula ψ_1 for C_1 is $EBX' = EAX$. (Note that the abstract semantic-footprints of C_1 are within those of φ_7, and so C_1 does not get pruned away by the footprint-based pruner.) φ_7 requires the pre-state bits in register EBX for the computation it performs. However, when C_1 transforms a pre-state to a post-state, C_1 loses the pre-state bits that were in EBX because they were overwritten by the pre-state bits that were in register EAX. (Note that this overwrite is explicitly shown in the QFBV formula ψ_1.) This observation has two implications: (i) C_1 cannot implement φ_7 because it has lost some of the pre-state bits that are required to implement φ_7, and (ii) no matter what instruction sequence we append to C_1, we can never get back the pre-state bits in register EBX. Consequently, McSynth++ discards C_1.

Suppose that the next candidate enumerated by the slave is $C_2 \equiv$ "`sub ebx,eax`." The formula ψ_2 for C_2 is $EBX' = EBX - EAX$. (To simplify the presentation of this example, we pretend that the `sub` instruction does not set any flags.) One requires the pre-state bits in registers EAX and EBX to implement φ_7. When C_2 transforms a pre-state to a post-state, C_2 overwrites the pre-state bits in register EBX with a value computed from the pre-state bits in registers EAX and EBX. Even though C_2 has overwritten the pre-state bits in register EBX per se, those bits are latent in the post-state value in EBX (denoted by the term "$EBX - EAX$"), and could be recovered from that post-state value. In this specific example, one can restore the pre-state bits in register EBX by appending the instruction "`add ebx,eax`" to C_2. However, recovery of the pre-state bits is not always possible, e.g., consider $C_3 \equiv$ "`and ebx,eax`." When the pre-state bits required to implement φ_7 are *possibly* latent in the current candidate, such as C_2 or C_3, McSynth++ *conservatively* assumes that they can be recovered by additional instructions, and does not prune the candidate. (The algorithm used in the bits-lost-based pruner is described in §4.2.2.)

Eventually, the slave enumerates $C_4 \equiv$ "`add ebx, eax`." The formula ψ_4 for C_4 is $EBX' = EBX + EAX$. (To simplify the presentation of this example, we pretend that the `add` instruction does not set any flags.) C_4 is not discarded by the footprint-based pruner and the bits-lost-based pruner, and enters the CEGIS loop. CEGIS tests equivalence and returns C_4 as the implementation of φ_7. McSynth++ also moves C_4 to the front of the instruction pool for the next synthesis task. In clients like partial evaluators where McSynth++ is invoked multiple times, this heuristic allows McSynth++ to try out sooner the instructions that are commonly used in synthesized code.

McSynth++'s slaves synthesize code for the remaining sub-formulas in a similar manner. Each slave finishes the synthesis task in a few seconds, and McSynth++'s master returns the concatenated instruction-sequence given below as the synthesized code.

```
mov [esp], eax; imul ebx, eax, 2;
     shr ebx, 2; add ebx, eax;
         mov eax, [esp + 4]
```

The entire synthesis task finishes in under a minute. For this example, in comparison to McSynth, McSynth++ speeds up synthesis by *over four orders of magnitude*.

Algorithm 1 Algorithm McSynthMaster

```
Input: $\varphi$, timeout
Output: $C_{conc}$ or FAIL
 1: splits ← EnumerateSplits($\varphi$)
 2: for each split $\langle\varphi_1, \varphi_2\rangle \in$ splits do
 3:     if IsFlowDependent($\langle\varphi_1, \varphi_2\rangle$) = MAYBE then
 4:         continue
 5:     end if
 6:     $ret_1$ ← McSynthMaster($\varphi_1$, timeout)
 7:     if $ret_1$ = FAIL then
 8:         continue
 9:     end if
10:     $ret_2$ ← McSynthMaster($\varphi_2$, timeout)
11:     if $ret_2$ = FAIL then
12:         continue
13:     end if
14:     ret ← Concat($ret_1$, $ret_2$)
15:     return ret
16: end for
17: return McSynthSlave($\varphi$, timeout)
```

4. Algorithm

In this section, we describe the algorithms used by McSynth++. First, we present the algorithms for the improvements to the "divide" phase. Second, we present the algorithm for the improvement to the "conquer" phase. Third, we describe the "move-to-front" heuristic. Finally, we provide the correctness guarantees for McSynth++'s algorithm.

4.1 "Divide" Phase

McSynth++'s master implements the "divide" phase of synthesis. The goal of the "divide" phase is to split the input formula into as many smaller independent sub-formulas as possible. We start by briefly presenting the base algorithm used by McSynth's master; we then present the algorithms for the improvements to the "divide" phase in McSynth++.

4.1.1 Base Algorithm

The algorithm used by McSynth's master is given as Alg. 1. The algorithm takes a formula φ and a timeout value as inputs, and either returns an implementation C_{conc} or FAIL. Alg. 1 first enumerates all possible splits $\langle\varphi_1, \varphi_2\rangle$ of φ via EnumerateSplits (Line 1). EnumerateSplits effectively divides the updates (to registers, flags, and memory lo-

Algorithm 2 Algorithm IsFlowDependent

```
Input: $\langle \varphi_1, \varphi_2 \rangle$
Output: NO or MAYBE
1: killed ← DropPrimes(SFP#_KILL($\varphi_1$))
2: used ← SFP#_USE($\varphi_2$)
3: if killed ∩ used = ∅ then
4:     return NO
5: else
6:     return MAYBE
7: end if
```

cations) in φ between φ_1 and φ_2 in all possible ways.[3] Alg. 1 then uses the one-sided decision procedure IsFlowDependent to test if a split is legal (Lines 3–5). (Recall from §2.2 that a sufficient condition for legality of a split in MCSYNTH is flow independence.) Alg. 1 discards illegal splits; for a given legal split $\langle \varphi_1, \varphi_2 \rangle$, Alg. 1 tries to synthesize code for φ_1 and φ_2, respectively, by recursively calling McSynthMaster (Lines 6–13). If Alg. 1 synthesizes code for all subformulas in a split, it returns the concatenated instruction-sequence (Line 15). If none of the splits work out, Alg. 1 hands over the entire formula to a slave synthesizer (Line 17). In Alg. 1, McSynthSlave invokes the slave synthesizer.

The one-sided decision procedure IsFlowDependent is given as Alg. 2. Alg. 2 checks for possible flow dependence by computing the abstract semantic-footprints of the input formulas. If there are no common symbols in $\text{SFP}^{\#}{}_{\text{KILL}}(\varphi_1)$ and $\text{SFP}^{\#}{}_{\text{USE}}(\varphi_2)$, then Alg. 2 returns NO (meaning that there is definitely no flow dependence from φ_1 to φ_2); otherwise, Alg. 2 returns MAYBE (Lines 3–6). In Alg. 2, DropPrimes drops the primes from the symbols in the kill footprint.

4.1.2 Improvements

MCSYNTH++'s master improves upon MCSYNTH's master in two ways:

- MCSYNTH++ uses an improved one-sided decision procedure for flow dependence, and
- MCSYNTH++ flattens "deep" terms before splitting a formula into subformulas.

In this sub-section, we describe these improvements in greater detail.

4.1.2.1 Improved one-sided decision procedure for flow dependence. One can see that IsFlowDependent (Alg. 2) is overly conservative in its treatment of memory. Because IsFlowDependent uses abstract semantic-footprints, which in turn overapproximate all memory locations by a single symbol "*Mem*," IsFlowDependent can return MAYBE for splits that are actually flow-independent.

MCSYNTH++ uses a more precise one-sided decision procedure (ImprovedIsFlowDependent) to test split $\langle \varphi_1, \varphi_2 \rangle$ for possible flow dependence. The decision procedure is given as Alg. 3. To check for the absence of flow dependences via registers and flags, Alg. 3 uses abstract semantic-footprints (Lines 1–7). If there are no flow dependences introduced through registers or flags, Alg. 3 collects the terms denoting memory locations that might be modified by φ_1 via CollectMemUpdateTerms (Line 8), and the terms denoting memory locations that might be accessed by φ_2 via CollectMemAccessTerms (Line 9). Alg. 3 then uses an SMT solver to test if any modified location might overlap with any used location, and returns MAYBE if that is the case; otherwise, it returns NO (Lines 10–17). (In Alg. 3, SAT checks satisfiability of a formula.) Consequently, ImprovedIsFlowDependent returns NO (meaning that the split is flow independent) only when each memory location possibly modified by φ_1 is guaranteed to not overlap with any memory location that might be used by φ_2.

Algorithm 3 Algorithm ImprovedIsFlowDependent

```
Input: $\langle \varphi_1, \varphi_2 \rangle$
Output: NO or MAYBE
1: killed ← DropPrimes(SFP#_KILL($\varphi_1$))
2: used ← SFP#_USE($\varphi_2$)
3: killedRegsFlags ← killed − {Mem}
4: usedRegsFlags ← used − {Mem}
5: if killedRegsFlags ∩ usedRegsFlags ≠ ∅ then
6:     return MAYBE
7: end if
8: killedMem ← CollectMemUpdateTerms($\varphi_1$)
9: usedMem ← CollectMemAccessTerms($\varphi_2$)
10: for $T_1$ ∈ killedMem do
11:     for $T_2$ ∈ usedMem do
12:         if SAT($T_1 = T_2$) then
13:             return MAYBE
14:         end if
15:     end for
16: end for
17: return NO
```

4.1.2.2 Flattening "deep" terms. During the "divide" phase, MCSYNTH splits the conjuncts (updates to registers, flags, and memory locations) in φ between φ_1 and φ_2. However, the master does not attempt to split the terms *within* a conjunct, and the smallest formula that a MCSYNTH slave can receive as input is an entire conjunct in φ. If such a conjunct contains a term/sub-formula with a deep AST that can only be implemented by three or more (IA-32) instructions, searching for an implementation by a slave might take days. Consequently to speed up synthesis, the master must attempt to split such "deep" terms/sub-formulas in φ. (In the remainder of this sub-section, we use "deep term" as shorthand for "deep term/sub-formula.")

MCSYNTH++ flattens such "deep" terms using interface constants. An interface constant is an extra symbolic constant that MCSYNTH++ adds to the vocabulary of φ. Given a "deep" term *T*, MCSYNTH++ iteratively picks a term/sub-formula *t* in *T*, replaces *t* by a fresh interface constant n in *T*,

[3] For the interested reader, the algorithm for EnumerateSplits is given as Alg. 6 in [26] (excluding lines 13–15).

Algorithm 4 Algorithm McSynth++Master

```
Input: Flattened $\varphi$, $\rho$, L, $I_{def}$, timeout
Output: $C_{conc}$ or FAIL, updated $\rho$
 1: $\varphi \leftarrow$ Substitute($\varphi$, $\rho$)
 2: splits $\leftarrow$ EnumerateSplits($\varphi$)
 3: for each split $\langle \varphi_1, \varphi_2 \rangle \in$ splits do
 4:     if ImprovedIsFlowDependent($\langle \varphi_1, \varphi_2 \rangle$) = MAYBE then
 5:         continue
 6:     end if
 7:     $I^1_{use} \leftarrow$ UsedInterfaceConsts($\varphi_1$)
 8:     $I^1_{def} \leftarrow$ DefinedInterfaceConsts($\varphi_1$)
 9:     $I^2_{use} \leftarrow$ UsedInterfaceConsts($\varphi_2$)
10:     if $I^1_{use} - I_{def} \neq \emptyset$ or $I^2_{use} - I_{def} - I^1_{def} \neq \emptyset$ then
11:         continue
12:     end if
13:     $L^1 \leftarrow L \cup$ UsedRegs($\varphi_2$)
14:     $\langle ret_1, \rho_1 \rangle \leftarrow$ McSynth++Master($\varphi_1$, $\rho$, $L^1$, $I_{def}$, timeout)
15:     if $ret_1$ = FAIL then
16:         continue
17:     end if
18:     $\langle ret_2, \rho_2 \rangle \leftarrow$ McSynth++Master($\varphi_2$, $\rho_1$, L, $I_{def} \cup I^1_{def}$, timeout)
19:     if ret2 = FAIL then
20:         continue
21:     end if
22:     ret $\leftarrow$ Concat($ret_1$, $ret_2$)
23:     return $\langle ret, \rho_2 \rangle$
24: end for
25: for each interface constant $c \in \varphi$ do
26:     $r \leftarrow$ PickRegister(L)
27:     $\rho[c] = r$
28: end for
29: $\varphi \leftarrow$ Substitute($\varphi$, $\rho$)
30: return $\langle$McSynth++Slave($\varphi$, timeout), $\rho \rangle$
```

and appends the conjunct $n = t$ to φ. After flattening all such "deep" terms in φ, the resultant formula φ' and φ are equi-satisfiable. (In fact, φ and φ' have identical sets of models if we disregard interface constants in models.) An example of a flattened formula φ' is given in Eqn. (1) in §3. MCSYNTH++ supplies φ' as input to its master.

The algorithm for MCSYNTH++'s master is given as Alg. 4. Because the input formula now contains interface constants, MCSYNTH++'s master must (i) take into account interface constants while checking if a split is legal (Lines 7–12), and (ii) assign interface constants concrete locations before invoking slave synthesizers (Lines 25–28). (To simplify the presentation of Alg. 4, we assume that MCSYNTH++'s master assigns interface constants scratch registers. It is straightforward to use memory locations as scratch locations in Alg. 4.) To solve the aforementioned tasks, Alg. 4 has additional inputs and outputs in comparison to Alg. 1. To solve task (i), Alg. 4 takes as input the set of interface constants defined in predecessor sub-formulas (I_{def}). To solve task (ii), Alg. 4 takes as input a map ρ that maps interface constants to assigned scratch registers, and the set of registers L that are live at the point where code is to be synthesized. Alg. 4 also returns an updated version of ρ as an additional output.

For each interface constant c that has been assigned a register r in predecessor sub-formulas, Alg. 4 first substitutes r for c in φ via `Substitute` (Line 1). This step ensures that each interface constant is uniformly replaced by the same register in all sub-formulas of a split. Alg. 4 then enumerates the candidate splits of φ, and uses ImprovedIsFlowDependent (Alg. 3) to check if a split $\langle \varphi_1, \varphi_2 \rangle$ of φ is flow-independent (Line 4). Once Alg. 4 finds a flow-independent split, it checks if each use of an interface constant is preceded by its definition: each interface constant used in φ_1 must be defined in a predecessor sub-formula, and each interface constant used in φ_2 must be defined either in a predecessor sub-formula or φ_1 (Line 10). (In Alg. 4, `UsedInterfaceConsts` returns the set of interface constants used in a formula; `DefinedInterfaceConstants` returns the set of interface constants that are defined in a formula.) At this point, Alg. 4 has found a legal split of φ.

Alg. 4 computes the set of registers L^1 that are live after φ_1 (Line 13). (In Line 13, `UsedRegs` returns the set of unprimed registers in a formula.) Alg. 4 then recursively calls itself on the sub-formulas φ_1 and φ_2, respectively, while passing the suitable sets of live-after registers and defined interface-constants (Lines 14–21). If the algorithm fails to synthesize code for all candidate splits of φ, Alg. 4 assigns dead registers to interface constants via `PickRegister`, and records the assignments in ρ (Lines 25–28). (Note that any register not in the live-after set L is dead at the point where code is to be synthesized.) Finally, Alg. 4 replaces interface constants in φ with the assigned registers via `Substitute`, and invokes the slave synthesizer (Lines 29 and 30).

In summary Alg. 4 recursively splits the input formula into sub-formulas, while assigning dead registers to interface constants, and ensuring legality of splits. If there are sufficient dead registers available at the point where code is to be synthesized, one can also use a more naïve register-assignment technique in Alg. 4, e.g., each interface constant gets a unique dead register.

4.2 Conquer Phase

MCSYNTH++'s slave implements the "conquer" phase of synthesis. The goal of the "conquer" phase is to synthesize an instruction sequence for a given sub-formula. First, we briefly present the base synthesis-algorithm used by MCSYNTH's slave; we then present the algorithm for the improvement to the "conquer" phase in MCSYNTH++.

4.2.1 Base Algorithm

The algorithm used by MCSYNTH's slave is given as the unhighlighted and unboxed lines of Alg. 5. Given a formula φ, the slave enumerates templatized instruction-sequences of increasing length, and uses CEGIS to check if there exists an instantiation of a candidate instruction-sequence that implements φ (Lines 16–21). Before the CEGIS loop, the slave tries to prune away candidates via abstract semantic-footprints: if the abstract semantic-footprints of the candi-

Algorithm 5 Algorithm McSynth++Slave (Unhighlighted and unboxed lines constitute algorithm McSynthSlave)

```
Input: $\varphi$, timeout
Output: $C_{conc}$ or FAIL
 1: instrPool ← ReadInstrPool()
 2: prefixes ← $\{\epsilon\}$
 3: while prefixes $\neq \emptyset$ do
 4:     for each prefix $p \in$ prefixes do
 5:         prefixes ← prefixes − {p}
 6:         for each templatized instruction $i \in$ instrPool do
 7:             C ← Append($p$, $i$)
 8:             $\psi_c \leftarrow \langle\langle C \rangle\rangle$
 9:             if $SFP^{\#}{}_{USE}(\psi_c) \not\subseteq SFP^{\#}{}_{USE}(\varphi) \vee SFP^{\#}{}_{KILL}(\psi_c) \not\subseteq SFP^{\#}{}_{KILL}(\varphi)$ then
10:                 continue
11:             end if
12:             if $BITS^{\#}_{available}(\psi_c) \not\supseteq BITS^{\#}_{required}(\varphi)$ then
13:                 continue
14:             end if
15:             prefixes ← prefixes ∪ {C}
16:             ret = CEGIS($\varphi$, C, $\psi_c$)
17:             if ret ≠ FAIL then
18:                 instrPool ← MoveToFront(ret, instrPool)
19:                 WriteInstrPool(instrPool)
20:                 return ret
21:             end if
22:         end for
23:     end for
24: end while
25: return FAIL
```

date are outside those of φ, the slave prunes the candidate away (Lines 9–11). The slave prunes away only useless candidates (candidates that either do not implement φ, or implement φ but superfluously use or modify locations that are otherwise unused/unmodified by φ). If an instance of any of the remaining candidates implements φ, the slave returns that instance as the implementation of φ. In Alg. 5, `ReadInstrPool` populates an instruction list by reading templatized instructions from a file; `Append` appends an instruction to an instruction sequence.

4.2.2 Improvements

MCSYNTH++'s slave improves upon MCSYNTH's slave by using an additional pruner based on the pre-state bits lost by an instruction sequence when it transforms a pre-state to a post-state.

MCSYNTH uses abstract semantic-footprints to prune away candidates that might either use/modify a location that is otherwise unused/unmodified by the specification φ. While this pruning heuristic prunes away candidates that might use *extra information* in comparison to φ, it does not prune away candidates that *do not have enough information* to implement φ. We now present a pruning heuristic that discovers when candidates have insufficient information to implement φ.

Suppose that $\text{BITS}_{required}(\varphi)$ represents the set of registers and flags that are required to implement a specification φ. (The bits-lost-based pruner in MCSYNTH++ currently handles only registers and flags. Extending this pruner to handle memory locations is a possible direction for future work.) For example, consider the formula $\varphi_1 \equiv EAX' = EAX + EBX$. To implement φ_1, one needs the pre-state bits in registers EAX and EBX, i.e., $\text{BITS}_{required}(\varphi_1) = \{\text{EAX, EBX}\}$. Consider another formula $\varphi_2 \equiv EAX' = EAX \,\&\, \texttt{0x0000ffff}$. To implement φ_2, one needs only the least-significant 16 pre-state bits in register EAX, i.e., $\text{BITS}_{required}(\varphi_1) = \{\text{AX}\}$. (In IA-32, register AX denotes the least-significant half of register EAX.) One can semantically characterize $\text{BITS}_{required}$ as follows:

Definition 1.

A bit $b \notin \text{BITS}_{required}(\varphi)$ iff $\forall m, m \models \varphi \Leftrightarrow m_{\bar{b}} \models \varphi$, where $m_{\bar{b}}$ is m with bit b flipped.

Recall from §2.1 that the QFBV formula that specifies the state transformation performed by an instruction sequence is a restricted 2-vocabulary formula of the form

$$\bigwedge_m (I'_m = T_m) \wedge \bigwedge_n (J'_n = \varphi_n) \wedge Mem' = FE,$$

where *FE* is a function-update expression of the form $Mem[L_1 \mapsto V_1][L_2 \mapsto V_2] \ldots [L_p \mapsto V_p]$. (Each L_i is a term that denotes a memory location l, and each V_i is a term that denotes the value in l in the post-state.) In such a 2-vocabulary formula, we call each term/formula T_m, φ_n, and V_p an *r-value* term[4] because they denote the post-state r-values. Given a candidate C whose state transformation is represented by the formula ψ_c, let $\text{BITS}^{\#}_{available}(\psi_c)$ represent the set of unprimed register and flag constant-symbols that appear in r-value terms in ψ_c. $\text{BITS}^{\#}_{available}(\psi_c)$ is really an over-approximation of the registers and flags whose pre-state bits can be recovered from ψ_c. For example, consider the candidate $C_1 \equiv$ "`mov eax, ebx`" whose formula ψ_1 is

$$EAX' = EBX \wedge EBX' = EBX \wedge \ldots \wedge EDI' = EDI \wedge \ldots$$
$$CF' = CF \wedge SF' = SF \wedge \ldots \wedge Mem' = Mem.$$

(Note that we have explicitly shown in ψ_1 the identity conjuncts for parts of the state that are unmodified by C_1.) Then $\text{BITS}^{\#}_{available}(\psi_1) = \{\text{EBX}, \ldots, \text{EDI, CF, SF}, \ldots\}$. (Note that EAX is *not* in $\text{BITS}^{\#}_{available}(\psi_1)$.) MCSYNTH++ computes $\text{BITS}^{\#}_{available}$ for a formula ψ via a syntax-directed translation over ψ. In the following definitions, *RF* is the set of unprimed constant symbols used for registers and flags, *T* is the set of QFBV terms, and *FE* is the set of function-update expressions over the function symbol *Mem*.

[4] In compiler parlance, the l-value of an assignment denotes the location that is assigned to by the assignment; the r-value denotes the value that is assigned to the location.

Definition 2.

$$\mathrm{BITS}^{\#}_{\mathit{available}}(c) = \begin{cases} \{c\} & \mathit{if}\ c \in \mathit{RF} \\ \emptyset & \mathit{otherwise} \end{cases}$$

$$\mathrm{BITS}^{\#}_{\mathit{available}}(\mathit{Mem}(t)) = \emptyset,\ \mathit{where}\ t \in T$$

$$\mathrm{BITS}^{\#}_{\mathit{available}}(\mathit{Mem}) = \emptyset$$

$$\mathrm{BITS}^{\#}_{\mathit{available}}(\mathit{fe}[l \mapsto v]) = \mathrm{BITS}^{\#}_{\mathit{available}}(v) \cup \mathrm{BITS}^{\#}_{\mathit{available}}(\mathit{fe}),\ \mathit{where}\ l,\ v \in T,\ \mathit{and}\ \mathit{fe} \in \mathit{FE}$$

For all other cases, $\mathrm{BITS}^{\#}_{\mathit{available}}$ *is the union of* $\mathrm{BITS}^{\#}_{\mathit{available}}$ *of the constituents.*

In the implementation, if a register R is in $\mathrm{BITS}^{\#}_{\mathit{available}}$, McSynth++ adds the smaller registers enclosed by R to $\mathrm{BITS}^{\#}_{\mathit{available}}$. (For example, if EAX is in $\mathrm{BITS}^{\#}_{\mathit{available}}$, McSynth++ adds AX, AL, and AH to $\mathrm{BITS}^{\#}_{\mathit{available}}$.)

McSynth++ prunes away any candidate C (with QFBV formula ψ_c) that satisfies the following property:

$$\mathrm{BITS}^{\#}_{\mathit{available}}(\psi_c) \not\supseteq \mathrm{BITS}_{\mathit{required}}(\varphi).$$

Suppose that one can obtain $\mathrm{BITS}_{\mathit{required}}(\varphi)$ precisely for an input formula φ. Even when $\mathrm{BITS}^{\#}_{\mathit{available}}$ overapproximates the registers/flags whose pre-state bits are available in ψ_c, if a candidate C satisfies the above property, then it could never implement φ, and thus McSynth++ prunes away C. Moreover, no matter what instruction sequence we append to C, we can never get back the lost bits. Consequently, McSynth++ also does not retain C as a prefix for enumerating future candidates. If $\mathrm{BITS}_{\mathit{required}}$ can be obtained precisely for an input formula, then the bits-lost-based pruner does not affect the completeness properties of the synthesizer.

Because it is difficult to obtain $\mathrm{BITS}_{\mathit{required}}(\varphi)$ precisely for an input formula φ, McSynth++ uses an overapproximation of $\mathrm{BITS}_{\mathit{required}}(\varphi)$: McSynth++ computes $\mathrm{SFP}^{\#}{}_{\mathrm{USE}}(\varphi)$, and disregards the symbol "*Mem*," which denotes the entire memory in abstract semantic-footprints. (i.e., $\mathrm{BITS}^{\#}_{\mathit{required}}(\varphi) = \mathrm{SFP}^{\#}{}_{\mathrm{USE}}(\varphi) - \{\mathit{Mem}\}$). Even though this overapproximation makes McSynth++'s synthesis algorithm incomplete (Thm. 2), the bits-lost-based pruner provides an additional 7–14% improvement on top of the improvements obtained by our other techniques (see §6).

The algorithm used by McSynth++'s slave is given as Alg. 5. The bits-lost-based pruning is shown in the highlighted lines of Alg. 5.

4.3 Pragmatics

The principal use case of McSynth++ is that of a code generator in semantics-based binary-rewriting clients (e.g., the residual-code generator in the machine-code partial evaluator WiPEr [25]). Binary rewriters would typically convert instructions in a program into a formula, modify the formula based on various semantic criteria, and supply the modified formula as input to McSynth++. In programs, certain operations tend to occur more frequently than others (e.g., increment/decrement the stack pointer, write to the stack, etc.) and certain instructions tend to be used more frequently than others to implement such operations. It is beneficial to prioritize during synthesis the instructions that are used to implement common operations in input formulas.

To prioritize useful instructions, McSynth++ uses a "move-to-front" heuristic: whenever a slave finds an implementation, McSynth++ moves the templatized instructions that occur in that implementation to the front of the list that serves as the instruction pool in the next synthesis task; the next synthesis task could be the invocation of a slave on a different sub-formula in the same input formula, or the invocation of a slave on a sub-formula in a new input formula. In Alg. 5, this heuristic is denoted by the call to `MoveToFront` (Line 18); `WriteInstrPool` dumps the instruction list to a file (Line 19), which would be read by the next synthesis task (Line 1).

Both McSynth and McSynth++ also incorporate function caching to reuse implementations of previously seen formulas.

4.4 Correctness

In this sub-section, we present the soundness and completeness properties of McSynth++.

Lemma 1. *Alg. 5 is sound. (The formula* $\langle\!\langle I \rangle\!\rangle$ *for instruction sequence I returned by Alg. 5 is logically equivalent to the input QFBV formula* φ*.)*

Proof. The CEGIS loop of the slave (Line 16 in Alg. 5) returns an instruction sequence *I* only if $\langle\!\langle I \rangle\!\rangle$ is equivalent to φ. □

Lemma 2. *For any legal split* $\langle \varphi_1, \varphi_2 \rangle$ *of* φ*, if* $\varphi_1 \Leftrightarrow \langle\!\langle I_1 \rangle\!\rangle$*,* $\varphi_2 \Leftrightarrow \langle\!\langle I_2 \rangle\!\rangle$*, and l is a set of scratch locations that appear in* I_1 *and* I_2 *but not in* φ*, then* $\varphi \Leftrightarrow \langle\!\langle I_1; I_2 \rangle\!\rangle$ *disregarding scratch locations l.*

Proof. If there are no interface constants in φ, then flow independence is the only criterion for a legal split, and the proof for this lemma under the aforementioned case is available elsewhere (Lemma 2 in [26]). Thus, we only have to prove that the lemma still holds when interface constants are present in a flow-independent split $\langle \varphi_1, \varphi_2 \rangle$. Without loss of generality, let us assume that there is only one interface constant n, which replaces term *T* in φ. Because $\langle \varphi_1, \varphi_2 \rangle$ is a legal split, "$n' = T$" would appear in φ_1, and n would be used in φ_2. McSynth++ assigns n a location l that is dead at the point where code is to be synthesized, and I_1 initializes the location l according to *T*. Because I_2 follows I_1, I_2 always reads the value written by I_1 in l. Consequently, φ and $\langle\!\langle I_1; I_2 \rangle\!\rangle$ are equisatisfiable, and $\varphi \Leftrightarrow \langle\!\langle I_1; I_2 \rangle\!\rangle$ disregarding l. □

Theorem 1. ***Soundness.*** *McSynth++ is sound.*

Proof. Follows from Lemmas 1 and 2. □

With the exception of candidates pruned away because of the imprecision introduced while computing $\text{BITS}^{\#}_{required}$ for an input formula, MCSYNTH++ has the same completeness guarantees as MCSYNTH (Thm. 2 in [26]).

Theorem 2. ***Completeness.*** *Modulo SMT timeouts and candidates that are pruned away because of imprecision in $BITS^{\#}_{required}(\varphi)$, if there exists an instruction sequence I that (i) is equivalent to φ, and (ii) does not superfluously use/-modify locations that are otherwise unused/unmodified by φ, then* MCSYNTH *will find I and terminate.*

Proof. In comparison with MCSYNTH, the only source of incompleteness in MCSYNTH++ is the bits-lost-based pruner. If $\text{BITS}^{\#}_{required}(\varphi)$ is imprecise, then a candidate that actually has enough information to implement φ might be pruned away. Apart from the bits-lost-based pruner there are no other sources of incompleteness: the "move-to-front" heuristic does not prune any candidate; MCSYNTH++'s master tries out all candidate splits of φ, and if MCSYNTH++ fails to synthesize code for all candidate splits, MCSYNTH++ supplies the entire φ as input to a slave synthesizer. □

5. Implementation

MCSYNTH++ uses Transformer Specification Language (TSL) [14] to convert instruction sequences into QFBV formulas. The concrete operational semantics of the integer subset of IA-32 is written in TSL, and the semantics is reinterpreted to produce QFBV formulas [15]. MCSYNTH++ uses ISAL [14, §2.1] to generate the templatized instruction pool for synthesis. MCSYNTH++ uses Yices [7] as its SMT solver. In the examples presented in this paper, we have treated memory as if each memory location holds a 32-bit integer. However, in our implementation, memory is addressed at the level of individual bytes. MCSYNTH++, just like MCSYNTH, is capable of accepting scratch registers for synthesis [26, §4.4].

6. Experiments

We tested MCSYNTH++ on QFBV formulas obtained from instruction sequences from the SPECINT 2006 benchmark suite [11], and also in the context of a client—the machine-code partial evaluator WIPER [25]. Our experiments were designed to answer the following questions:

1. In comparison with MCSYNTH, what is the speedup in synthesis time caused by each individual improvement to the "divide" phase? What is the speedup when both improvements are used together?
2. In comparison with MCSYNTH, how many timeouts remain with each individual improvement to the "divide" phase? How many timeouts remain with both improvements together?
3. With both improvements to the "divide" phase turned on, what is the speedup in synthesis time caused by the bits-lost-based pruner and the "move-to-front" heuristic, respectively? What is the speedup when both improvements are used together?

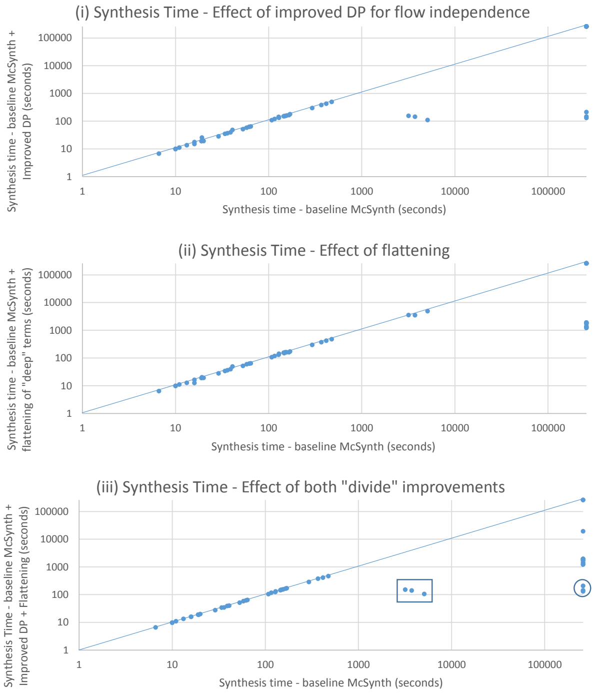

Figure 3: Synthesis times obtained via improvements to the "divide" phase in MCSYNTH++ for the corpus of 50 QFBV formulas.

4. What is the speedup in residual-code-synthesis time when MCSYNTH++ is used in the place of MCSYNTH in WIPER?

All experiments were run on a system with a quad-core, 3GHz Intel Xeon processor; however, MCSYNTH++'s algorithm is single-threaded. The system has 32 GB of memory.

To answer the first three questions, we used the same benchmark suite that was used to test MCSYNTH: QFBV formulas obtained from a representative set of "important" instruction sequences that occur in real programs. We harvested the five most frequently occurring instruction sequences of lengths 1 through 10 from the SPECINT 2006 benchmark suite (50 instruction sequences in total). We converted each instruction sequence into a QFBV formula and used the resulting formulas as inputs for our experiments.

Note that in general there is no restriction on the source of the input formula, and the formula can come from any client; we simply chose to obtain the input formulas from instruction sequences for experimental purposes.

To answer the first two questions, we measured the synthesis time with (i) only the improved decision-procedure for flow independence turned on, (ii) only flattening of "deep" terms turned on, and (iii) both improvements turned on. We compared the numbers against the baseline synthesis-time numbers obtained from MCSYNTH.

The results are shown in Fig. 3(i), (ii), and (iii), respectively. In Fig. 3, the blue lines represent the diagonals of the scatter plots. If a point lies below and to the right of the diagonal, the baseline performs worse. All axes in Fig. 3

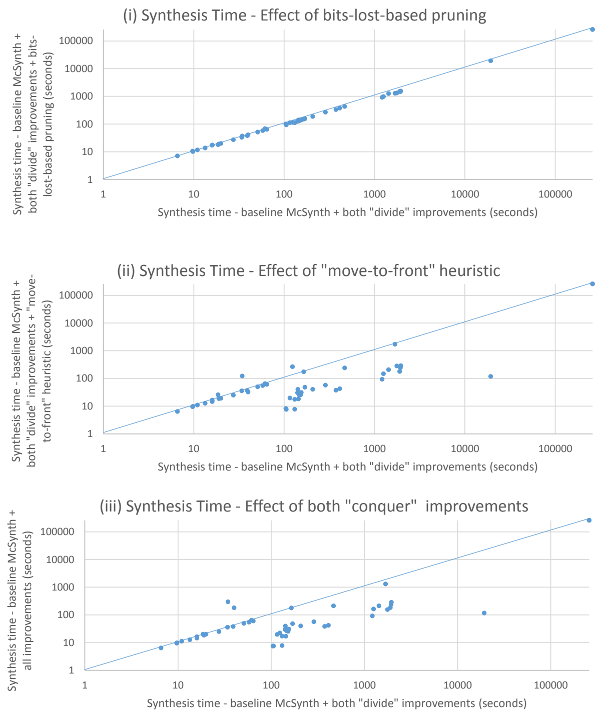

Figure 4: Synthesis times obtained via improvements to the "conquer" phase in McSynth++ for the corpus of 50 QFBV formulas.

use logarithmic scales. McSynth timed out on 14 formulas. (The timeout value was three days.) With only the improved decision-procedure turned on, the number of timeouts was 10; with only flattening turned on, the number of timeouts was 6; with both improvements to the "divide" phase turned on, the number of timeouts was 2. For the formulas that timed out in McSynth but did not timeout in McSynth++, the average speedup in synthesis time was over 233X (computed as a geometric mean). For 3 formulas, the speedup was over three orders of magnitude (surrounded by a circle in Fig. 3(iii)). Among the formulas that did not timeout in McSynth, the two improvements reduced the synthesis time considerably only for three formulas (surrounded by a square in Fig. 3(iii)). For the formulas that did not timeout, the average speedup in synthesis time caused by the two improvements was 1.34X.

To answer the third question, we turned on both improvements to the "divide" phase, and we measured the synthesis time with (i) only the bits-lost-based pruner turned on, (ii) only the "move-to-front" heuristic turned on, and (iii) both improvements turned on. We compared the numbers against the synthesis times obtained when both improvements to the "divide" phase were turned on. The results are shown in Fig. 4(i), (ii), and (iii), respectively. The average speedup in synthesis time caused by the bits-lost-based pruner was 1.07X (computed as the geometric mean). If we consider only formulas whose baseline synthesis-time numbers are 100 seconds or more, the speedup is slightly higher: 1.14X. We believe that the speedup is relatively small because McSynth's footprint-based pruner already prunes away most of the candidates that could potentially be pruned away by the bits-lost-based pruner. For example, consider the input formula $\varphi \equiv EBX' = EAX + EBX$, and the candidate C $\equiv$ "`mov eax, ebx`" with the QFBV formula $\psi \equiv EAX' = EBX$. The candidate loses the pre-state bits in register EAX when it transforms a state. Because φ requires the pre-state bits in EAX for the computation it performs, C will be potentially pruned away by the bits-lost-based pruner. However, $\text{SFP}^{\#}_{\text{KILL}}(\varphi) = \{EBX'\}$ and $\text{SFP}^{\#}_{\text{KILL}}(\psi) = \{EAX'\}$, and the abstract semantic kill-footprint of ψ is outside that of φ. Consequently, C will be pruned away by the footprint-based pruner, and will never reach the bits-lost-based pruner.

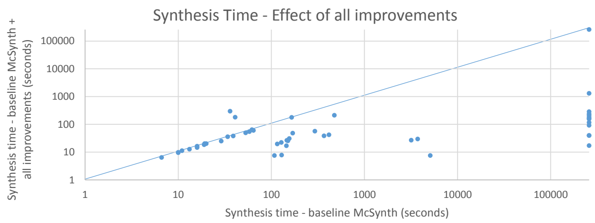

Figure 5: Effect of all improvements in McSynth++ for the corpus of 50 QFBV formulas.

The "move-to-front" heuristic caused more pronounced speedup. The average speedup in synthesis time caused by the heuristic was 3X (computed as a geometric mean). If we consider only formulas whose baseline synthesis-time numbers are 100 seconds or more, the speedup is 6X. Our corpus consists of formulas obtained from the most frequently-occurring instruction sequences, and programs tend to perform certain computations more frequently than others (e.g., increment/decrement the stack pointer, write to a stack location, etc.). Consequently, many formulas in our corpus contain specifications for such frequently performed computations. (Note that this would be the case even if a binary-rewriter client like a partial evaluator were producing the corpus of formulas because the rewriter originally obtains the base formulas from instructions in the binary.) The "move-to-front" heuristic produced a sizeable speedup because it prioritizes instructions that are used to implement common operations in the input formulas. The heuristic caused a slowdown in two formulas because it moved some instructions to the front of the instruction list, pushing later some more-infrequently-used instructions required to implement the formulas.

The comparison of synthesis-time numbers produced by McSynth and McSynth++ is shown in Fig. 5. In summary, with all the improvements turned on and in comparison with McSynth, McSynth++ speeds up the synthesis time by over 1981X for formulas that timed out in McSynth but did not timeout in McSynth++. For the formulas that did not timeout in McSynth, McSynth++ speeds up the synthesis time by 3X. If we consider only formulas whose baseline synthesis-time numbers are 100 seconds or more, the speedup is 11X.

Table 1: Comparison of residual-code-synthesis time using McSynth and McSynth++, respectively, in WiPEr.

Application	No. of calls to the synthesizer	Synthesis time using McSynth (seconds)	Synthesis time using McSynth++ (seconds)	Speedup
power	6	16	13.5	1.19
interpreter	19	30	22.8	1.32
sha1	23	25.4	21	1.21
filter	212	241	177	1.36
dotproduct	306	312	267	1.17

To answer the fourth question, we measured the total time taken to synthesize residual code using McSynth and McSynth++, respectively, while partially evaluating the microbenchmarks used in [25] with WiPEr. The results are shown in Table 1. The average speedup in residual-code-synthesis time caused by McSynth++ is 1.25X (computed as a geometric mean). Note that the microbenchmarks are fairly small programs (see Table 1 in [25]); the specialized formulas given to the synthesizer are also small, and are often implemented by one instruction.

7. Related Work

Superoptimization. Superoptimization aims to find an optimal instruction-sequence for a target instruction-sequence [4, 5, 13, 16, 18, 19, 23]. Peephole superoptimization [4] uses "peepholes" to harvest target instruction-sequences, and replace them with equivalent instruction sequences that have a lower cost. A superoptimizer can be implemented by using a machine-code synthesizer that has been enhanced to bias its search toward short instruction sequences. Recall that $\langle\!\langle \cdot \rangle\!\rangle$ converts an instruction-sequence into a QFBV formula. Suppose that SynthOptimize is a client of the synthesizer that is biased to synthesize short instruction sequences. Then a superoptimizer can be constructed as follows:

$$\textsf{Superoptimize}(\mathit{InstrSeq}) = \textsf{SynthOptimize}(\langle\!\langle \mathit{InstrSeq} \rangle\!\rangle)$$

However, one cannot construct a synthesizer from a superoptimizer. Moreover, recent superoptimizers sacrifice completeness for reduced superoptimization times by resorting to stochastic search [18, 23]. In contrast, the techniques used in McSynth++ aim to reduce the synthesis time as much as possible without losing its completeness properties.

From a practical standpoint, certain techniques used in modern superoptimizers like STOKE [23] can be applied to machine-code synthesis. The following points outline how one could potentially adapt STOKE for machine-code synthesis:

- The cost function in STOKE takes into account both correctness and performance. If a client of the synthesizer does not care about performance, one could drop the performance component of the cost function.
- STOKE uses Markov Chain Monte Carlo (MCMC) sampling to search through the space of instruction sequences, and validates candidates by executing the input and candidate instruction-sequences on bare metal using a finite set of test inputs. Executing tests on bare metal allows STOKE to meet the sampling-rate requirements of MCMC sampling. For machine-code synthesis, the specification of the input is a QFBV formula and not an instruction sequence, and for candidate validation, one needs to evaluate the input QFBV formula on test inputs. QFBV evaluation is much slower than executing tests on bare metal, and might not meet the rate requirements of MCMC sampling. However, one could evaluate the input formula using the test inputs a priori, and just use the post-states for comparison inside the MCMC loop.
- While generating candidates, STOKE restricts immediate operands in candidates to take values only from a small set S. If the input formula φ contains values that are not in S, STOKE might not find an implementation for φ. One possible workaround is to add all constants occurring in φ to S, and sample immediate operands from the updated S in the MCMC loop. However, this strategy might preclude STOKE from finding potentially better implementations. For example, suppose that $\varphi \equiv \mathit{EAX}' = (\mathit{EAX}+2)+2$. The aforementioned strategy will not find the implementation "`lea eax, [eax + 4]`" for φ.

It remains for future work to adapt STOKE for machine-code synthesis, evaluate on our set of benchmarks, and compare empirical completeness and performance with McSynth++.

Clients of a machine-code synthesizer. Partial evaluation [12] is a program-specialization technique that optimizes a program with respect to certain static inputs. A machine-code partial evaluator [25] partially evaluates a binary either to specialize it with respect to certain inputs, or to extract an executable component from a binary. In a machine-code partial evaluator, a synthesizer is used to synthesize residual code from formulas of instructions specialized with respect to static inputs. Because McSynth++ is much faster than McSynth, McSynth++ should speed up residual-code synthesis in a partial evaluator. Moreover, McSynth++ should enable the partial evaluator to perform specialization on a per-basic-block basis instead of a per-instruction basis, leading to more compact and optimized residual code.

A machine-code synthesizer also plays a key role in a machine-code slicer [27]. For purposes of precise slicing, a machine-code slicer converts a machine-code program into an intermediate representation (IR) that is at the microcode level (microcode is at an even lower level than machine code), and performs slicing on the microcode IR. However, if a client of the slicer wants a machine-code slice instead of a microcode slice, the slicer has to now reconstitute a machine-code program from the slice. To solve the program-reconstitution issue, a machine-code synthesizer can be used to synthesize instructions from the microcode fragments included in the slice. Because McSynth++ is much faster than McSynth, McSynth++ should reduce slicing times.

Dependence testing in arrays. A parallelizing compiler employs a series of tests to check for flow dependences between array references [10, 17]. The tests are often ordered as a sequence, ranging from cheapest (but approximate) to most expensive (but exact). If the tests say that an ⟨array-update, array-access⟩ pair is flow-independent, the parallelizing compiler proceeds to parallelize the sequential code.

The one-sided decision procedure in McSynth++ aims to solve the same problem as the aforementioned work: test if an ⟨array-update, array-access⟩ pair is flow-independent. However, instead of array variables in programs, McSynth++ deals with the memory array in QFBV formulas. Also, because state-of-the-art SAT solvers are quite efficient, McSynth++ uses SAT for alias testing instead of the series of tests used in the aforementioned work.

Alternatives to machine-code synthesis: QFBV-to-IA-32 compiler. One could try to build a QFBV-to-IA-32 compiler by (i) defining a set of patterns $\mathscr{P}$ that map basic QFBV fragments to IA-32 instruction sequences, and (ii) using a bottom-up rewriting system [2, 9] that computes the least-cost IA-32 cover for the input QFBV formula. However, one finds that the QFBV-to-IA-32 translation problem has some subtleties that make it difficult to create such a compiler.

- Not all QFBV formulas would be straightforward to handle. The easy formulas would be ones that specify a pre-state to post-state transformation (e.g., $\varphi_1 \equiv \mathit{EAX}' = \mathit{EAX}+2 \wedge \mathit{EBX}' = \mathit{EBX}+2$). Such a formula is said to be in *explicit form*. However, a client can supply a formula that expresses a property over pre-states and post-states (e.g., $\varphi_2 \equiv \mathit{EAX}' + \mathit{EBX}' = \mathit{EAX} + \mathit{EBX} + 4$). Such a formula is said to be in *implicit form*. In such cases, the client would expect an instruction sequence whose pre-state-to-post-state transformation satisfies the input formula. For example, "`lea eax, [eax + 4]`" is one instruction sequence that satisfies φ_2. Both McSynth and McSynth++ are capable of searching for an instruction sequence that satisfies a formula in implicit form. (See [26, §4.4], "Synthesizing code that satisfies properties.") However, it would be difficult to create a compiler that handles formulas in implicit form.
- Formulas use operators that are associative and commutative, and consequently different elements in a formula that are relevant for selecting a given instruction can be arbitrarily far apart. This situation prevents one from creating a compiler for formulas based on a bottom-up rewrite system (e.g., iburg [2], Twig [9], etc.).
- Certain clients might want the output instruction-sequence to possess a certain "quality" (small size, short runtime, low energy consumption, etc.). For example, a superoptimizer would like the synthesized code to have a short runtime. Because a QFBV-to-IA-32 compiler would have a fixed set of patterns $\mathscr{P}$, the compiler would not be able to produce instruction sequences with varying qualities: given input formula φ, it would always return the instruction sequence specified by $\mathscr{P}$. In contrast, because a synthesizer searches over the space of instruction sequences, it can find different implementations of φ with varying qualities. The algorithm used by McSynth and McSynth++ to find an implementation with a certain quality is given in [26, §4.4], "Quality of synthesized code."

8. Conclusion

In this paper, we described several improvements to the algorithms used in a state-of-the-art machine-code synthesizer McSynth. We presented McSynth++, an improved synthesizer for IA-32. Our experiments show that McSynth++ synthesizes code for 12 out of 14 formulas on which McSynth timed out, speeding up the synthesis time by over 1981X, and for the remaining formulas, McSynth++ speeds up the synthesis time by 3X.

References

[1] *Compilers: Principles, Techniques, and Tools*, chapter 8: Code Generation. Addison-Wesley, 2007.

[2] A. Aho, M. Ganapathi, and S. Tjiang. Code generation using tree matching and dynamic programming. *TOPLAS*, 35(4), 1989.

[3] G. Balakrishnan and T. Reps. WYSINWYX: What You See Is Not What You eXecute. *TOPLAS*, 32(6), 2010.

[4] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *ASPLOS*, 2006.

[5] S. Bansal and A. Aiken. Binary translation using peephole superoptimizers. In *OSDI*, 2008.

[6] D. Brumley, I. Jager, T. Avgerinos, and E. Schwartz. BAP: A Binary Analysis Platform. In *CAV*, 2011.

[7] B. Dutertre and L. de Moura. Yices: An SMT solver, 2006. http://yices.csl.sri.com/.

[8] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua. Scalable variable and data type detection in a binary rewriter. In *PLDI*, 2013.

[9] C. Fraser, D. Hanson, and T. Proebsting. Engineering a simple, efficient code-generator generator. *LOPLAS*, 1 (3), 1992.

[10] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *PLDI*, 1991.

[11] J. Henning. SPEC CPU2006 Benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[12] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., 1993.

[13] R. Joshi, G. Nelson, and K. Randall. Denali: A goal-directed superoptimizer. In *PLDI*, 2002.

[14] J. Lim and T. Reps. TSL: A system for generating abstract interpreters and its application to machine-code analysis. *TOPLAS*, 35(4), 2013.

[15] J. Lim, A. Lal, and T. Reps. Symbolic analysis via semantic reinterpretation. *Softw. Tools for Tech. Transfer*, 13(1):61–87, 2011.

[16] H. Massalin. Superoptimizer: A look at the smallest program. In *ASPLOS*, 1987.

[17] D. Maydan, J. Hennessy, and M. Lam. Efficient and exact data dependence analysis. In *PLDI*, 1991.

[18] P. Phothilimthana, A. Thakur, R. Bodik, and D. Ghurjati. Scaling up superoptimization. In *ASPLOS*, 2016.

[19] P. Phothilimthana, A. Thakur, R. Bodik, and D. Ghurjati. GreenThumb: Superoptimizer construction framework. UCB/EECS-2016-8, University of California–Berkeley Tech Report, Feb. 2016. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-8.pdf`.

[20] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *PLDI*, 2014.

[21] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from“big code”. In *POPL*, 2015.

[22] H. Saïdi. Logical foundation for static analysis: Application to binary static analysis for security. *ACM SIGAda Ada Letters*, 28(1):96–102, 2008.

[23] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *ASPLOS*, 2013.

[24] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Int. Conf. on Information Systems Security*, 2008.

[25] V. Srinivasan and T. Reps. Partial evaluation of machine code. In *OOPSLA*, 2015.

[26] V. Srinivasan and T. Reps. Synthesis of machine code from semantics. In *PLDI*, 2015.

[27] V. Srinivasan and T. Reps. An improved algorithm for slicing machince code. In *OOPSLA*, 2016.