Parallel Assembly Synthesis

Jingmei Hu[1][0000−0002−4434−5057], Stephen Chong[1][0000−0002−6734−5383], and Margo Seltzer[2][0000−0002−2165−4658]

[1] Harvard University, Cambridge MA 02138, USA
crystaljmhu@gmail.com, chong@seas.harvard.edu
[2] University of British Columbia, Vancouver BC, Canada
mseltzer@cs.ubc.ca

Abstract. Program synthesis offers an attractive alternative to the intricate and tedious process of writing assembly programs manually. Assembly program synthesis automatically generates implementations, given a high-level formal specification and a machine description. However, its limited scalability prevents widespread adoption. Automatic parallelization improves program synthesis in general, but parallelizing assembly synthesis is nontrivial as the realities that data are untyped and all state is global lead to an enormous search space and prevent straightforward decomposition into separable sub-problems that can be run in parallel. We present PASSES, a Parallel Assembly Synthesis System Exploiting Subspaces. PASSES uses five heuristics to transform an original assembly synthesis problem into a set of sub-problems; it runs multiple synthesis sub-problems in parallel and constructs the final result by combining them. We evaluate PASSES on 26 general bit manipulation assembly programming problems and 140 machine-dependent use cases from two operating systems. Compared to an existing assembly synthesis tool and a state-of-the-art parallel SMT solver, all five heuristics in PASSES significantly improve assembly synthesis scalability.

Keywords: program synthesis, assembly programming, parallel computing.

1 Introduction

Assembly language is used in many mission-critical systems that require direct manipulation of hardware, access to performance-critical instructions, or access to special-purpose accelerators. Assembly language is also common in device drivers, real-time systems, low-level embedded systems, and machine-dependent operating system code. However, assembly language is fundamentally intricate and tedious to write. Writing and debugging assembly programs tends to take more time than debugging higher-level language programs, because each processor architecture has its own assembly (so one might need to debug the same functionality multiple times), and data in assembly programs are untyped and global.

Assembly program synthesis, as an alternative to manual implementation [24, 49, 18], is a promising approach for automated assembly programming. Existing

assembly synthesis systems, in general, take a high-level formal specification, which describes *what* a program should do, and a machine description, which provides the executable model of an instruction set architecture (ISA) semantics, and uses *CounterExample Guided Inductive Synthesis (CEGIS)* [46] to generate a program, i.e., a sequence of assembly instructions that satisfies the specification. However, the limited scalability of synthesis, especially assembly language synthesis, prevents widespread adoption. Fundamentally, assembly synthesis is a search problem: it searches for an assembly program that satisfies the specification from the space of all instruction sequences. Compared to synthesis of higher-level language programs, the search space of possible programs in assembly synthesis is much larger, because data is untyped and global. The space is typically combinatorial in the number of machine state components (including dozens of registers and hundreds of memory locations) and exponential in the number of instructions [23, 24]. To synthesize a single three-operand instruction `OPCODE op1, op2, op3`, the size of the search space is approximately $n \times x \times y \times z$ where n is the number of possible `OPCODE`s and x, y, and z represent the number of choices for `op1`, `op2`, and `op3` respectively; for a program with N instructions, the space size is $(n \times x \times y \times z)^N$. Current tools and approaches are unable to synthesize assembly programs longer than a few instructions in reasonable time [5, 24, 49, 50].

A natural step in improving the performance of program synthesis is parallelization. Parallel program synthesis has been proposed in recent work that enables efficient parallelization of challenging synthesis problems [11, 26, 28]. There are two common approaches for parallelization: One method is to search for a solution over a set of instances with different settings, e.g., running several instances of a sequential solver with different parameters or several different solvers in parallel. If any of the instances succeeds in finding a solution, all the instances are terminated. Another method is to find solutions to subsets of the original problem (i.e., *sub-problems*) and recombine them into a full correct solution, e.g., using divide-and-conquer techniques [2, 3, 14]. Small sub-problems typically have smaller search spaces to explore than does the original problem; we call these smaller search spaces *subspace*s. However, unlike the general parallelization problem, it is not obvious how to conduct parallel search in assembly synthesis. Although the specifications used in assembly synthesis are simple pre- and post-conditions, they lead to an enormous search space for SMT solvers [24, 49], and the SMT expressions generated for assembly synthesis are difficult to decompose into separable conjunctions that can be solved in parallel, because data are untyped and machine state is global in assembly languages.

We present a novel parallel system for assembly synthesis, PASSES (Parallel Assembly Synthesis System Exploiting Subspaces), that uses domain knowledge of assembly language to parallelize synthesis. PASSES exploits this domain knowledge to identify multiple approaches to subspace creation. We first describe three general characteristics of subspaces in parallel synthesis problems: 1) whether the set of subspaces is exhaustive (i.e., *collective exhaustivity*), 2) whether the subspaces overlap (i.e., *mutual exclusivity*), and 3) whether they are

of (approximately) equal size (i.e., *subspace size equality*). Based on those categories, PASSES identifies five complementary and reusable heuristics for creating subspaces, each of which represents a synthesis *sub-problem*. We deploy the techniques by running multiple parallel synthesis tasks and collecting the results from them. To evaluate the effectiveness of these heuristics and PASSES, we collected 26 general bit manipulation assembly programming problems, used in previous work [17], and 140 machine-dependent use cases from two pre-existing OSes with four machine architectures as benchmark examples, also used in previous work [24]. Compared with the state-of-the-art assembly synthesis approach [24, 25], evaluation results show that all of PASSES's heuristics significantly improve the scalability of assembly synthesis, while preserving solution quality. In the best case, PASSES achieves a geometric mean speedup of more than 10× for programs that take the baseline more than 10 seconds to synthesize.

We summarize our contributions as follows:

- We develop a novel parallel assembly synthesis system, PASSES, to improve the scalability of assembly language synthesis.
- We design and implement five domain-specific heuristics, to transform an assembly language synthesis problem into a set of *sub-problems*, synthesize each sub-problem individually in parallel, and construct a solution from them.
- We evaluate the effectiveness of PASSES on general bit manipulation programs and use cases derived from the machine-dependent parts of two operating systems. Our evaluation demonstrates that PASSES greatly improves assembly synthesis scalability.

2 Background and Related Work

Program synthesis is used to automatically generate target executable programs that satisfy a given high-level formal specification, which is typically non-executable [6, 9, 15, 32, 37, 55]. Modern program synthesis techniques fall into two main categories: deductive synthesis, which synthesizes programs by constructively proving a theorem, employing logical inference and constraint solving [29, 32], and inductive synthesis, which finds programs matching a set of input-output examples and generalizes them to work for every input [12, 36, 53]. We focus more on the more relevant latter category of program synthesis.

In the area of inductive synthesis, there are two major directions: synthesis from informal and typically incomplete descriptions, e.g., Programming by Example [12, 16, 36, 54] and synthesis from formal specifications, which began in 2006 with the introduction of Sketch [48, 45], and Syntax Guided Synthesis (SyGuS) [1]. Later, Solar-Lezama et al. introduced Counterexample Guided Inductive Synthesis (CEGIS) [47], which uses an iterative process to perform inductive generalization for all possible inputs. Our work lies in this area of synthesis, where we take formal specifications and use CEGIS to generate assembly programs satisfying the desired behaviors.

In the area of fully automated synthesis with parallelism, there are two main topics: using divide-and-conquer to decompose a synthesis problem into simpler

ones, such as CYPRESS [43, 44], FlashMeta [38], and EUSOLVER [4], and using parallel Boolean-satisfiability/satisfiability-modulo-theories (SAT/SMT) solvers to expedite synthesis solving [31], such as PaInleSS [30], PBoolector [40] (which is the parallel implementation of Boolector [8] splits a bit-vector formula and solves the subproblems in parallel; in Section 5, we compare it to our subspace decomposition approach) and the *parallel portfolio approach* such as ManySat [19, 20, 52]. The traditional divide-and-conquer approaches for program synthesis focus more on the *division of the specification*. However, assembly synthesis typically takes a relatively straightforward specification that describes program behavior and synthesizes instruction sequences with a complicated machine model. The machine model contains all possible machine states, including registers and memory locations, and all possible instructions that must be considered; it is one of the key performance bottlenecks for assembly synthesis: it leads to an enormous search space with these global states and untyped data. It is challenging to separate or decompose them for parallelization, especially at the specification level. Thus, applying divide-and-conquer only to specifications might not produce as significant a performance improvement as we might like. Rather than dividing the specification, we *divide the search space* into smaller sub-spaces, reducing the impact of the exponential state space explosion. In contrast, parallel SAT/SMT solvers directly incorporate parallel algorithms by modifying the state-of-the-art solvers accordingly; in contrast, we propose and evaluate parallelism in the synthesis procedure rather than in the solver implementation.

3 Preliminaries and Terminology

3.1 Baseline Assembly Synthesizer: Aquarium

We implement PASSES on top of a state-of-the-art assembly synthesis engine. To the best of our knowledge, there are two main assembly synthesis related tools: STOKE [42] and Aquarium [24, 25]. STOKE [42] is a stochastic super-optimizer [33] that starts from an existing implementation, not an abstract specification, and optimizes the code to improve performance or reduce size. As such, it does not solve the problem we are trying to solve and is suitable neither as a starting point nor as a baseline comparison. Aquarium is a CEGIS-based assembly synthesis system designed to synthesize the machine-dependent parts of operating systems [24, 25]. It takes as input a functional specification (pre- and post-condition) and a machine model description, and produces a sequence of satisfied assembly instructions. As such, it seems well-matched to our setting.

3.2 Assembly Instruction Types

Though different machine architectures have distinctive assembly syntax and semantics, we can categorize assembly instructions into types. The following six types are used throughout the rest of this paper; in the list below, we provide examples of the types from ARMv7[3]: 1) `ARITH`: arithmetic, such as addition

[3] The conditional execution feature in ARM does not affect the corresponding types.

(`add`); 2) `LOGIC`: logical, such as shifts(`lsl`); 3) `MEMOP`: memory handling, such as loads/stores (`ldr`/`str`); 4) `DATAOP`: register-to-register data transfer (`mov`); 5) `JMP`: conditional and unconditional branches (`b`); and 6) `COPROC`: coprocessor handling (`mcr`/`mrc`).

3.3 Subspace Creation

We introduce the following notation to describe subspaces. Given a synthesis problem Q, let $M = SearchSpace(Q)$ be the entire search space of Q, i.e., the set of all possible programs. Our goal is to produce a set of n subspaces $P_1 \dots P_n$, each representing a sub-problem, $q_1 \dots q_n$ of Q, i.e., $P_i = SearchSpace(q_i)$. Note that $\forall i, P_i \subseteq M$ and a solution to q_i is a solution to Q.

We describe three (generally desirable) characteristics of subspaces: *collective exhaustivity*, *mutual exclusivity*, and *subspace size equality.*

Collective Exhaustivity. A set of n sub-problems are *collectively exhaustive* if their subspaces P_i cover the entire search space M of the problem Q; that is, $M = \bigcup_{i=1}^{n} P_i$. We write $\sigma \vDash Q$ if a program σ satisfies Q, i.e., σ is a solution to Q. Then the following two statements hold for *collective exhaustivity*: $\exists\ \sigma \vDash Q \implies \exists\ i \in [1, n]\ \sigma \vDash q_i$ and $\forall \sigma \nvDash Q \implies \forall i \in [1, n]\ \sigma \nvDash q_i$. Otherwise, non-exhaustivity leaves *unsearched* portions in M that may contain possible solutions.

While collective exhaustivity is generally desirable, it is not required. A subspace creation technique that is not collectively exhaustive means that failure to find a solution in subspaces *does not* imply that the problem is unsatisfiable.

Mutual Exclusivity. A set of subspaces is *mutually exclusive*, or *disjoint*, if they are non-intersecting, i.e., $\forall i \in [1, n]\ \forall j \in [1, n] \quad P_i \cap P_j \neq \emptyset \implies i = j$.

Mutual exclusivity is generally desirable as it means that searching the subspaces does not duplicate work: any possible solution appears in at most one subspace. Indeed, it could be that the restrictions required to make subspaces disjoint (e.g., additional constraints) increases synthesis times, so it might be more efficient to allow subspaces to overlap.

Subspace Size Equality. In general, synthesis time is positively correlated to the size of the search space: the larger the search space, the longer it takes to find a solution. *Subspace size equality* means that all created subspaces have approximately the same size; in other words, $\forall i \in [1, n], |P_i| \approx |M|/n$ ($|P_i|$ is the number of possible programs in P_i). It equally divides M, which leads to approximately the same synthesis time for each sub-problem.

Subspace size equality is generally desirable, but not required. Under some conditions, such as some characteristics of the original synthesis problem, unequal division may achieve better performance; we discuss this in Section 4.2.

4 PASSES: Parallel Assembly Synthesis System Exploiting Subspaces

Using insight specific to assembly synthesis, we develop five complementary and reusable heuristics to create subspaces. These heuristics are central to the

design of PASSES, our novel parallel assembly synthesis system. PASSES uses these heuristics to create sets of subspaces, tries to synthesize a solution in each subspace in parallel, and collects a final solution from them. We refer to the sub-problems that will be synthesized in parallel as *instance*s. We built PASSES on top of an existing CEGIS-based assembly synthesizer, Aquarium [24]. We explain PASSES's algorithm in Appendix A.1. Table 1 includes the subspace creation categories for those five heuristics; we discuss more detail about each heuristic in the following sections.

	Constraint-SC	RandSimpl-SC	TypeSimpl-SC	Inc-SC	PriorInc-SC
Subspace Creation	static	static	static	incremental	incremental
Collective Exhaustivity	✓	✓	✓		
Mutual Exclusivity	✓	✓	(✓)	(✓)	(✓)
Subspace Size Equality	✓	(✓)			

Table 1: Five heuristics in PASSES. "(✓)" denotes that only some sub-problems belong to the corresponding category.

4.1 Constraint-based Subspace Creation (Constraint-SC)

As the search space grows exponentially in the number of instructions in a program, so too does synthesis time. *Constraint-SC* creates subspaces by introducing *constraints* that require specific instructions to appear in specific locations in the target program; for example, if there is a pre-condition on a value stored in memory, perhaps it's reasonable to try programs where the first instruction loads a value from memory. The constraints applied to each instance determine not only which instructions should be considered but also, where they should appear, thus, there is no overlap among the subspaces and the sum of all subspaces covers the entire search space of the original problem. In practice, rather than select specific instructions, *Constraint-SC* constructs constraints using the six instruction types mentioned in Section 3 (Algorithm explained in Appendix A.2).

4.2 Model-Simplified Subspace Creation (Simpl-SC)

A complete machine model includes descriptions of all instructions; it contains more information than is strictly necessary to facilitate the synthesis of any single program (i.e., practically no program uses every instruction in an ISA). Hence, instead of considering the entire ISA, synthesizing with only a partial instruction set should dramatically improve scalability. *Simpl-SC* simplifies the given machine model and creates multiple subspaces, each representing a sub-problem with a different machine model containing fewer instructions (Algorithm explained in Appendix A.3). The main challenge is selecting appropriate partial instruction sets. Removing the instructions from the machine model might end up with something simple and convenient to consider, but the remaining machine model might not be capable of producing a correct target program.

Randomized Model-Simplified Subspace Creation (RandSimpl-SC) shuffles the whole instruction set and evenly divides them into groups as sub-models, each containing the same number of instructions with various types.

The subspaces are mutually exclusive, but may not cover the space. We add *RandSimpl-SC* to include the complete model to ensure that the entire space is covered.

Type-based Model-Simplified Subspace Creation (TypeSimpl-SC) uses the six instruction types mentioned in Section 3 to construct sub-models. *TypeSimpl-SC* creates $\binom{6}{2} = 15$ sub-models with 2 different instruction types. Note that programs with a single instruction type are rare in real-world situations, thus, we do not create further detailed subspaces for brevity. Similarly, the complete-model instance (i.e., the possible program should contain at least three different types of instructions) explores the rest of the search space.

4.3 Incremental Subspace Creation (Inc-SC)

The previous heuristics create subspaces statically; they begin with a fixed number of divided subspaces that exhaustively cover the search space with the complete-model instance. In contrast, *Inc-SC* divides the search space dynamically without considering the complete-model; it starts with a non-exhaustive division and incrementally expands the search to the entire space. We implement *Inc-SC* on top of *TypeSimpl-SC*, with algorithm explained in Appendix A.4.

4.4 Prioritized Incremental Subspace Creation (PriorInc-SC)

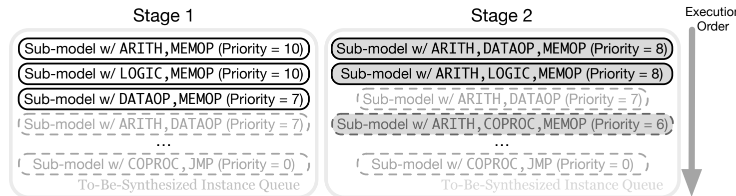


Fig. 1: Two stages in *PriorInc-SC*.

So far, all our heuristics select instructions with equal probability. However, sometimes the specification suggests that some instructions are more likely than others. Given this fact, *PriorInc-SC* extends *Inc-SC* with *Instruction Prioritization* using the following two-stage approach (Figure 1).

First, it statically analyzes the specification, placing instructions into the following three classes: *must-class* instructions, i.e., there is a high possibility that the target program has them, *may-class* instructions, i.e., they may be used in the target program, and *not-class* instructions, i.e., the possibility of their usage is low. For example, if the postcondition uses data from memory locations and calculates on the value, `MEMOP` is *must-class*, `ARITH` and `LOGIC` are *may-class* (they can sometimes achieve equivalent behavior), and `COPROC` is *not-class*, since no coprocessor handling involved. We leave the exploration of other prioritization algorithms for future work. Figure 1 shows an instance initialization example with *PriorInc-SC*, where `ARITH`, `LOGIC`, and `MEMOP` get higher priority, and the top three instances get synthesized first (solid black lines).

In Stage 2, *PriorInc-SC* generates 20 cases with initial states that satisfy the precondition and presents the generated candidate in each CEGIS iteration with a *score*, i.e., the number of cases for which it satisfies the postcondition. For each failed instance, we record the highest score achieved so far and update the priority of each instruction type involved. We always choose to execute instances with the highest priority, i.e., the instructions involved are most likely to be present in the target program. In Figure 1, after the instance with `ARITH` and `MEMOP` failed, newly added instances (gray boxes) might get higher priorities than the previous, not-executed ones (white boxes). Due to its dynamic enlargement, *PriorInc-SC* does not preserve *collective exhaustivity*, and for each newly added instance, we apply mutually exclusive constraints to avoid search space reconsideration.

5 Evaluation

We implemented PASSES in about 3500 lines of OCaml using *Aquarium* [21, 24], implementing each of the five subspace creation heuristics. To demonstrate the effectiveness and efficacy of search space reduction and parallelization in PASSES, we evaluate its performance on 26 bit manipulation programming tasks and 140 machine-dependent operating system (OS) code examples, compared to *Aquarium* [21, 24, 25], as a baseline. We also consider two different SMT parallelization techniques. As mentioned in Section 2, we use *PBoolector* [40], the parallel SMT solver algorithm for Boolector [8]. We also implemented a synthesizer wrapper that runs multiple (sequential) Boolectors in parallel with different random seeds (hereinafter referred to as *RandomSeed*).

5.1 Benchmarks

We select the following two different categories of benchmark programs.

Bit Manipulation Benchmarks Inspired by Gulwani et al. [17] and based on the scalability of current assembly synthesis approaches, we select 26 examples from the book *Hacker's Delight* [51], more detailed appears in Appendix B. We provided the specification of the desired behavior for each example in 32-bit MIPS, by specifying the pre- and post-condition. We first ran unmodified Aquarium [24], which is guaranteed to synthesize the shortest program satisfying the specification. All examples can be implemented with between 2 and 5 assembly instructions. Table 5 in Appendix B reports the number of instructions in each synthesized implementation, indicating the minimum length requirement for each benchmark.

Operating System Related Benchmarks We obtained more complex examples using the 140 use cases from Aquarium [24], including 35 individual procedures from two pre-existing OSes, Barrelfish [7] and OS/161 [22], implementing machine-dependent OS functionality. They consist of machine-level system call trap and kernel entry code in Barrelfish, C standard library function `setjmp` and `longjmp`, user-level program startup code, system call stub, interrupt disable code, and kernel-level thread switch in OS/161. Each procedure is implemented with four different machine architectures: 32-bit ARM, 32-bit MIPS, 32-bit RISC-V, and 64-bit x86_64. Their length varies from 1 to 14 instructions.

5.2 Experimental Setup

We ran all experiments on an *m5.8xlarge* AWS EC2 instance with 32 virtual CPUs and 128 GB of memory; all benchmarks use eight processors for parallel synthesis. We also ran *PBoolector* with the maximum of eight sub-problems generated when splitting bit-vector formulas, and *RandomSeed* with eight different random seeds for synthesis. For bit manipulation examples, the synthesis target is MIPS programs that work on bitvectors of size 32 bits, while for OS-related examples, we synthesize for all four architectures listed above.

5.3 Performance Results

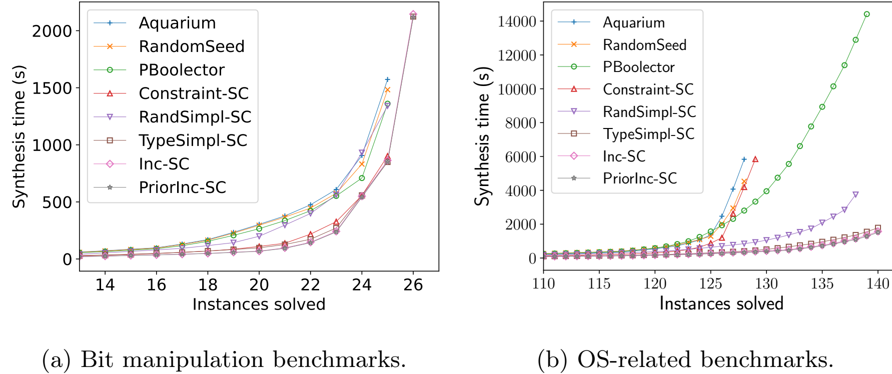
We measured synthesis performance for all of the aforementioned heuristics in PASSES. We ran each use case five times, varying the random number seed each time. We also randomized the variable names used for solver communication by prepending a random five-character alphabetical string, which produces significant variance in solver performance.

We run each synthesis task with a half-hour timeout; several use cases timed out under the baseline setting, denoted with "—" in the following tables.

Use	*Aquarium*	*Constraint-SC*		*RandSimpl-SC*		*TypeSimpl-SC*		*Inc-SC*		*PriorInc-SC*		*PBoolector*	
Case	Time (s)	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
P1b	0.86 (2)	0.73 (2)	1.18	1.73 (2)	0.50	1.72 (2)	0.50	0.83 (2)	1.04	0.68 (2)	1.26	1.97 (2)	0.44
P5a	1.28 (2)	0.54 (2)	2.37	1.12 (2)	1.14	2.86 (2)	0.45	1.91 (2)	0.67	1.82 (2)	0.70	1.88 (2)	0.68
P7b	2.05 (2)	0.60 (2)	3.42	1.77 (2)	1.16	2.99 (2)	0.69	1.99 (2)	1.03	1.97 (2)	1.04	2.12 (2)	0.97
P1a	2.21 (2)	0.83 (2)	2.66	1.34 (3)	1.65	1.75 (2)	1.26	0.82 (2)	2.70	0.77 (2)	2.87	1.51 (2)	1.46
P4a	2.78 (2)	**0.52 (2)**	**5.35**	1.87 (2)	1.49	2.17 (2)	1.28	0.97 (2)	2.87	1.01 (2)	2.75	2.69 (2)	1.03
P2a	2.81 (2)	0.69 (2)	4.07	1.62 (2)	1.73	2.11 (2)	1.33	1.52 (2)	1.85	1.45 (2)	1.94	1.03 (2)	2.73
P2b	3.48 (2)	0.93 (2)	3.74	2.13 (2)	1.63	2.02 (2)	1.72	0.82 (2)	4.24	**0.66 (2)**	**5.27**	2.45 (2)	1.42
P7a	3.69 (2)	**0.69 (2)**	**5.35**	2.08 (2)	1.77	2.90 (2)	1.27	1.58 (2)	2.34	1.47 (2)	2.51	3.93 (2)	0.94
P13a	3.91 (3)	3.01 (3)	1.30	0.96 (3)	4.07	0.82 (3)	4.77	**0.75 (3)**	**5.21**	0.79 (3)	4.95	2.97 (3)	1.32
P8a	6.56 (3)	6.04 (3)	1.09	12.27 (3)	0.53	1.49 (3)	4.40	1.45 (3)	4.52	1.53 (3)	4.29	8.66 (3)	0.76
P3a	9.04 (3)	3.62 (3)	2.50	4.88 (3)	1.85	6.35 (3)	1.42	5.48 (3)	1.65	5.46 (3)	1.66	7.97 (3)	1.13
P5b	10.28 (3)	3.92 (3)	2.62	9.56 (3)	1.08	4.60 (3)	2.23	3.80 (3)	2.71	3.67 (3)	2.80	8.93 (3)	1.15
P9	10.86 (3)	8.58 (3)	1.27	7.26 (4)	1.50	2.46 (3)	4.41	2.51 (3)	4.33	2.67 (3)	4.07	7.49 (3)	1.45
P8b	12.64 (3)	5.46 (3)	2.32	5.28 (3)	2.39	**1.81 (3)**	**6.98**	**1.49 (3)**	**8.48**	**1.49 (3)**	**8.48**	8.09 (3)	1.56
P3b	12.73 (3)	5.71 (3)	2.23	12.43 (4)	1.02	8.58 (3)	1.48	7.10 (3)	1.79	7.09 (3)	1.80	12.77 (3)	1.00
P4b	12.91 (3)	5.63 (3)	2.29	13.83 (3)	0.93	11.88 (3)	1.09	10.00 (3)	1.29	9.01 (3)	1.43	12.19 (3)	1.06
P10	32.56 (3)	24.52 (3)	1.33	25.67 (3)	1.27	**5.26 (3)**	**6.19**	**5.19 (3)**	**6.27**	**5.27 (3)**	**6.18**	26.47 (3)	1.23
P11b	39.01 (3)	10.38 (3)	3.76	13.10 (3)	2.98	12.73 (3)	3.06	**5.67 (3)**	**6.88**	**5.65 (3)**	**6.90**	38.29 (3)	1.02
P11a	63.36 (3)	**11.27 (3)**	**5.62**	23.86 (3)	2.66	**8.03 (3)**	**7.89**	**4.79 (3)**	**13.23**	**4.79 (3)**	**13.23**	52.62 (3)	1.20
P6	70.27 (4)	27.75 (4)	2.53	56.22 (4)	1.25	46.23 (4)	1.52	49.24 (4)	1.43	49.31 (4)	1.43	58.10 (4)	1.20
P11c	73.88 (3)	16.21 (3)	4.56	97.46 (3)	0.76	14.89 (3)	4.96	**9.82 (3)**	**7.52**	**9.84 (3)**	**7.51**	71.19 (3)	1.04
P14b	97.64 (3)	109.61 (3)	0.89	102.05 (4)	0.96	99.05 (3)	0.99	95.48 (3)	1.02	94.07 (3)	1.04	86.49 (3)	1.13
P13b	134.02 (4)	145.82 (4)	0.92	163.17 (4)	0.82	29.06 (4)	4.61	28.75 (4)	4.66	**26.13 (4)**	**5.13**	129.48 (4)	1.04
P12b	297.69 (4)	229.05 (4)	1.30	409.52 (4)	0.73	291.61 (4)	1.02	313.56 (4)	0.95	310.26 (4)	0.96	156.65 (4)	1.90
P12a	666.35 (4)	**79.78 (4)**	**8.35**	370.10 (4)	1.80	286.06 (4)	2.33	307.98 (4)	2.16	305.15 (4)	2.18	651.56 (4)	1.02
P14a	—	—		—		1273.88 (5)	>1.41	1281.60 (5)	>1.40	1267.12 (5)	>1.42	—	

Table 2: Performance of all 26 bit manipulation benchmarks, sorted by Aquarium baseline. For each benchmark, the table shows: (1) *Aquarium* baseline runtime, *PBoolector* runtime, and PASSES with five heuristics (in seconds, averaged across 5 trials), (2) number of instructions synthesized (in parentheses), and (3) the speedup, i.e., the ratio of the baseline time to the time with parallelism. "—" denotes cases where synthesis does not complete within the 1800-second timeout. A light gray background shows cases where a heuristic sped up synthesis ($Speedup > 0$); bold with a dark gray indicates large speedup ($Speedup > 5$).

Performance of Bit Manipulation Benchmarks Table 2 reports the synthesis performance comparison between *Aquarium* and five heuristics for all 26



(a) Bit manipulation benchmarks.

(b) OS-related benchmarks.

Fig. 2: Heuristics performanace relative to *Aquarium* baseline. Each plot shows each benchmark's average runtime across five trials that do not time out; we omit benchmarks in which all trials timed out. We omit the first 13 bit manipulation and 110 OS-related benchmark results to improve readability.

bit manipulation benchmark examples, sorted by the *Aquarium* runtime; Figure 2a adds comparison with the two other parallel implementations (*PBoolector* and *RandomSeed*). All PASSES heuristics, except *RandSimpl-SC*, generate the same length programs as *Aquarium* (reported in Table 5), although the actual synthesized programs are not identical; *RandSimpl-SC* introduces randomness, sometimes generating longer solution programs (with one more instruction) in 5 trials (in P1a, P3b, P6, P9, P14b). The synthesis times vary widely with different use cases; occasionally the heuristics take longer than *Aquarium*. We include the speedup information, i.e., the ratio of the baseline time to the synthesis time with each heuristic in Table 2, and we examine the performance result closely in the following.

Constraint-SC produces a geometric mean speedup of 2.43×. In the best case, it makes the synthesis up to 8.35× faster; it speeds up the synthesis in 23 out of 26 cases. However, it sometimes slows down synthesis due to the overhead caused by parallelism, such as in P14b and P13b. In the worst case, we observe a slowdown of 1.12×. Both *Aquarium* and *Constraint-SC* fail to synthesize P14a within the half-hour time limit. Figure 2a shows that *Constraint-SC* expedites assembly synthesis even though most programs are quite short.

When *Aquarium* is already quite fast, the overhead of PASSES can dominate performance. For example, when *Aquarium* takes less than 10 seconds, *RandSimpl-SC* frequently produces slowdown (up to 2.01×). However, for complicated programs, it makes the synthesis up to 4.07× faster, with a geometric mean speedup of 1.33×. *RandSimpl-SC* also fails to synthesize P14a within the half-hour time limit. Since *RandSimpl-SC* first randomly shuffles the instruction sets for later subspace creation, we also include the mean and standard deviation result across 5 trials in Table 3. This randomness produces large standard deviations on synthesis time for the complicated programs, especially for programs that take more than 100 seconds to synthesize.

Use Case		P1b	P5a	P7b	P1a	P4a	P2a	P2b	P7a	P13a	P8a	P3a	P5b	P9
Aquarium		0.86	1.28	2.05	2.21	2.78	2.81	3.48	3.69	3.91	6.56	9.04	10.28	10.86
RandSimpl-SC	Mean	1.73	1.12	1.77	1.34	1.87	1.62	2.13	2.08	0.96	12.27	4.88	9.56	7.26
	SD	0.12	0.22	0.25	0.81	0.48	0.44	0.49	0.05	0.16	2.18	1.65	4.67	4.27

Use Case		P8b	P3b	P4b	P10	P11b	P11a	P6	P11c	P14b	P13b	P12b	P12a	P14a
Aquarium		12.64	12.73	12.91	32.56	39.01	63.36	70.27	73.88	97.64	134.02	297.69	666.35	—
RandSimpl-SC	Mean	5.28	12.43	13.83	25.67	13.1	23.86	56.22	97.46	102.05	163.17	409.52	370.1	—
	SD	5.58	7.36	3.05	16.58	10.19	23.51	11.57	26.47	29.25	45.77	219.5	100.89	

Table 3: Means and standard deviation (SD) of *RandSimpl-SC* with all 26 bit manipulation benchmarks, sorted by *Aquarium* baseline. "—" denotes cases where synthesis does not complete within the 1800 second timeout.

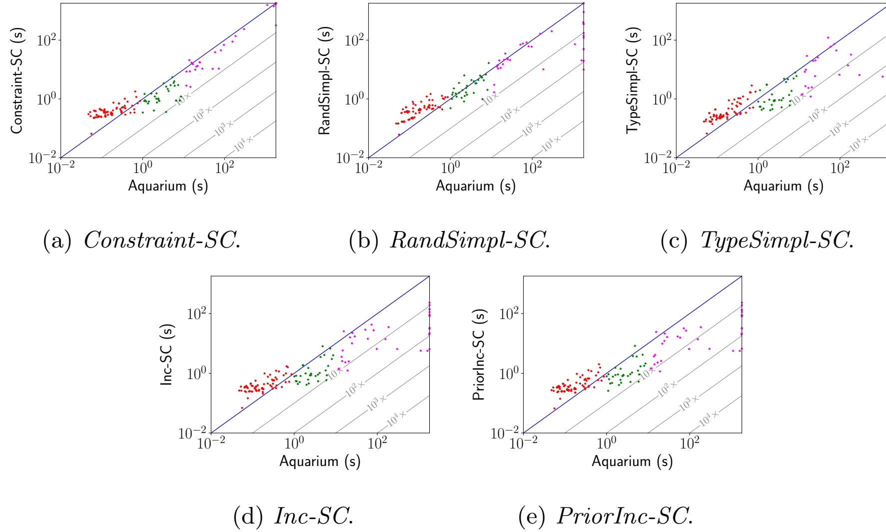
TypeSimpl-SC produces more consistent results: 22 out of 26 cases achieve synthesis speedup. We observe a maximum speedup of 7.89×, with a geometric mean speedup of 1.99× (up to 2.23× slowdown due to the parallel overhead). For the programs that take *Aquarium* more than 3 seconds to synthesize, *TypeSimpl-SC* produces a significant and consistent speedup. It also synthesizes programs that are not accessible to *Aquarium*: it successfully synthesizes an assembly program for P14a in fewer than 1300 seconds; if we treat timeouts as taking 1800 seconds, this conservative speedup is 1.41×.

Inc-SC accelerates synthesis in most cases; it is beneficial even for small programs. It makes the synthesis up to 13.23× faster, with a geometric mean speedup of 2.69× (up to 1.49× slowdown). We also observe that *Inc-SC* induces an overhead due to the dynamic instance launches. Table 2 shows that, *Inc-SC* produces a slowdown for some complicated cases such as P12a and P14a, compared with the original *TypeSimpl-SC*, while it still expedites the synthesis compared with *Aquarium*. It slightly slows down synthesis for only two cases (P5a and P12b), compared to *Aquarium*.

PriorInc-SC is more effective.We notice that the previous heuristics sometimes generate unpromising instances that fail synthesis easily; due to our limited degree of parallelism (i.e., we benchmark with eight threads), those instances waste CPU resources during synthesis. *PriorInc-SC* prioritizes instances and executes those most likely to succeed first. Thus, it eliminates the overhead caused by unpromising instances. It produces a geometric mean speedup of 2.77× and a maximum speedup of 13.23× (up to 1.42× slowdown). It also successfully synthesizes the troublesome P14a. Compared to *Inc-SC*, it makes synthesis faster in some cases, especially for complicated programs. *Our high level observation is that Figure 2a indicates that* TypeSimpl-SC*,* Inc-SC*, and* PriorInc-SC *outperform* Aquarium *and solve all benchmarks within the time limit.*

Comparison with Parallel SMT Solver and Different Random Seeds. Table 2 includes the detailed *PBoolector* synthesis runtime and speedup compared with *Aquarium* baseline, for each bit manipulation benchmark. As shown in Figure 2a, for all 26 bit manipulation benchmarks, all five heuristics outperform both *PBoolector* and *RandomSeed*. In the best case, *PriorInc-SC* outperforms *PBoolector* and *RandomSeed* with a maximum speedup of 10.98× and 12.91× for the program that takes *Aquarium* about 60s to synthesize, respectively.

Search Space Comparison. We evaluate all bit manipulation benchmarks in 32-bit MIPS; the complete machine model includes 37 assembly instructions which covers all the basic operations. As a first-order approximation, the overall size of the search spaces for n-instruction programs is $2^{36.6n}$, while *TypeSimpl-SC*, for example, prunes the subspaces for 2-type machine models into $2^{26.7n}$ with `ARITH` and `DATAOP`, $2^{27.7n}$ with `LOGIC` and `MEMOP`, or even $2^{19.8n}$ with `JMP` and `COPROC`.



(a) *Constraint-SC*. (b) *RandSimpl-SC*. (c) *TypeSimpl-SC*.

(d) *Inc-SC*. (e) *PriorInc-SC*.

Fig. 3: Effect of PASSES with heuristics on synthesis runtimes, with all 140 OS-related benchmark examples. Each shows PASSES with heuristics against *Aquarium*. Each data point represents the runtime under both conditions for one OS-related benchmark, averaged over five trials (timeouts are counted as 1800 seconds). $1s-$ *programs* are colored in red, $10s-$ *programs* are colored in green, and $10s+$ *programs* are colored in pink. The blue diagonal line represents equal time under both conditions, so that points below/right of the diagonal line demonstrate better performance with the heuristic. Gray contours provide guidelines for visually estimating the speedup factor. The upper and right boundaries of the plot represent an 1800-second timeout.

Performance of Operating System Related Benchmarks Figure 2b and Figure 3 show the synthesis performance comparison between *Aquarium* and PASSES with all five heuristics for the 140 OS-related benchmarks. We categorize those benchmarks into three groups: (1) $1s-$ *programs*: they take *Aquarium* no more than 1 second to complete, colored with red in Figure 3; (2) $10s-$ *programs*: their *Aquarium* runtime is more than 1 second but no more than 10 seconds, colored with green in Figure 3; (3) $10s+$ *programs*: they take *Aquarium* more than 10 seconds to synthesize, colored with pink in Figure 3. In general, PASSES accelerates synthesis on the majority of the OS-related benchmark examples, especially as the synthesis time increases. In particular, for those $10s+$ programs, the heuristics and PASSES's ability to run them all in parallel reduces the

synthesis time. However, the overhead of subspace creation and parallelism can cause a slowdown in synthesis for small programs, especially those $1s-$ programs. In Figure 3, we observe a cluster of data points (red) above the diagonal line in the lower left-hand corner in all figures, indicating that for those $1s-$ programs, PASSES with heuristics slightly slows the synthesis. Similarly, all PASSES heuristics produce programs of the same length as *Aquarium*, except *RandSimpl-SC*, which may generate solutions with one additional instruction across 5 trials.

	Constraint-SC	*RandSimpl-SC*	*TypeSimpl-SC*	*Inc-SC*	*PriorInc-SC*
$1s-$ Programs	0.43	0.41	0.46	0.42	0.42
$10s-$ Programs	2.15	1.16	2.27	2.46	2.75
$10s+$ Programs	1.49	3.42	9.21	10.71	11.30

Table 4: Geometric mean speedups of all the heuristics compared to *Aquarium* with OS-related benchmarks.

Table 4 reports the geometric mean speedups of these three groups, evaluating with five heuristics against *Aquarium*. This indicates an entirely acceptable trade-off: a slowdown of 2.33× for $1s-$ programs on average is a small price to pay for a speedup of 2.16× for $10s-$ programs and 7.23× for $10s+$ programs on average.

Constraint-SC is often beneficial, although its benefit is not as outstanding as other heuristics. Figure 2b shows that though it slows synthesis for small programs with a maximum slowdown of 6.87×, if we assume a mix of large and small jobs, the speedup on the $10s+$ programs more than compensates for it. For those $10s+$ programs, it makes the synthesis faster with a geometric mean of 1.49×. In the best case, it produces a maximum speedup of 10.35×.

RandSimpl-SC and *TypeSimpl-SC* both outperform *Aquarium*, while, as discussed before, *TypeSimpl-SC* produces more consistent results than does *RandSimpl-SC*. They sometimes slow synthesis for small programs, but for those $10s+$ programs, *RandSimpl-SC* and *TypeSimpl-SC* produce a geometric mean speed up of 3.42× and 9.21×, respectively. In the best case, they synthesize programs that are not accessible to *Aquarium*, such as the SJ-1 and LJ-1 cases; counting timeouts as 1800 seconds, there is a speedup of 180.49× and 288.29×, respectively. Due to the subspace creation and the parallel overhead, they sometimes slow down synthesis up to 6.59× and 5.23× for the programs that take *Aquarium* less than 1 second to synthesize, respectively. For the $10s+$ programs, they produce a slowdown of 1.93× and 2.38×, respectively.

Inc-SC and *PriorInc-SC* are also effective, and they both perform better than *TypeSimpl-SC*. They produces a geometric mean speedup of 1.45× and 1.51× with all benchmarks, and 10.71×and 11.30× for the $10s+$ programs, respectively. Counting those timeouts as 1800 seconds, they make the synthesis for those programs not accessible to *Aquarium* up to 282.84× and 284.99×, respectively. In the worst case, they make our synthesis 6.88× and 6.44× slower for small programs, respectively. For the $10s+$ programs, they produces a slowdown of 1.64× and 1.56×, respectively.

Comparison with Parallel SMT Solver and Different Random Seeds. Compared to *PBoolector* and *RandomSeed*, all five heuristics produce better synthesis

performance for the 140 OS-related benchmarks, especially in larger cases as shown in Figure 2b. In the best case, *TypeSimpl-SC*, *Inc-SC* and *PriorInc-SC* all outperform *PBoolector* and *RandomSeed* with a maximum speedup of up to about 200× and 280×, respectively, for the programs that are not accessible to *Aquarium* (counting timeouts as 1800 seconds for comparison).

6 Discussion

6.1 Parallelism Trade-off

PASSES generates multiple sub-problems and synthesizes them simultaneously, leveraging the fact that smaller search spaces make it easier, and therefore faster, to find a solution or determine that one does not exist. However, Amdahl's law [41] also clearly delineates the scenarios in which this analysis holds. That is, the overall performance speedup gained is limited by the fraction of time that the improved part is actually used [39]. Each parallel algorithm comes with its own overhead, particularly in terms of setup in apportioning the work to a set of sub-problems and tear-down in collecting the aggregated results from the sub-problems. Furthermore, multiple CPU and I/O resources are required for parallel execution, and synthesis itself is a memory-consuming work; coordinating multiple synthesis tasks, in general, leads to drastically high memory usage, which in turn slows down the entire procedure.

The evaluation clearly showed that there is a trade-off between creating sub-problems with small search spaces and reducing the parallel overhead for synthesis. Sub-problems with smaller search spaces, in general, can be solved more quickly, but to prevent missing some potential solutions, the number of sub-problems increases. In the evaluation, we initialize with eight processors for parallelism on the AWS instance to counteract memory consumption. Balancing between the number of parallel tasks and memory consumption for each task remains an important avenue of future investigation, to explore and identify the threshold where the benefit of parallelism for synthesis is maximized.

6.2 Performance Trade-off

Another critical trade-off is between finding a solution quickly and synthesizing optimal solutions. The nature of synthesis indicates that there can exist multiple solutions that satisfy a given specification [13]. With multiple subspaces created, those possible solutions are distributed over potentially many sub-problems. The iterative procedure in CEGIS guarantees that for the original synthesis problem Q, the synthesized program always has the minimum program length. However, given a set of n sub-problems $q_1 \ldots q_n$, derived from Q, for example, q_i could produce an a-instruction program in t_i seconds, while q_j generates a b-instruction program in t_j seconds, where $a > b$ and $t_i < t_j$. Since PASSES always takes the first returned result from any sub-problem as the solution, it returns with an assembly program of a instructions instead of one with b instructions. It is possible that the length of the program produced by PASSES is not minimized.

Furthermore, assembly programs are frequently optimized for many metrics: code size, code density, execution speed, program latency, data throughput, or even energy consumption [10]. To address this concern, we assume that techniques such as superoptimization [34] can be performed after synthesis. Meanwhile, PASSES also has the ability to collect multiple solutions from sub-problems and evaluate them against different optimization or desirability metrics.

6.3 Generalizability of PASSES

Unlike prior work that used the divide-and-conquer methodology to expedite synthesis, in PASSES, none of the presented heuristics manipulate the specification: they directly decompose the search space by imposing instruction-level constraints or simplifying the machine model; the specification remains intact. Overall, assembly synthesis, as a special category in general program synthesis problems, has features that facilitate parallel processing. The assembly synthesizer, in general, uses symbolic execution to explore every execution path based on the program semantics, while with instruction-level *constraints* and machine model *simplification*, we instantiate symbolic values to prune impossible execution paths, achieving smaller search space for each sub-problems. Those symbolic values are highly related to the machine model and the target program; instantiating them based on assembly features does not require low-level operations, and it also benefits the following parallel computing. Morover, The incremental heuristics in PASSES share the idea with Lazy Task Creation [35] to explore parallel tasks dynamically at run-time. To generalize PASSES to other synthesis problems, it is important to identify those highly problem-related symbolic variables and concretize them with values. Though previous work on adaptive concretization demonstrates that randomly concretizing influential unknowns helps synthesis performance [27], PASSES still shows the significance of problem-related instantiation during symbolic execution in program synthesis.

7 Conclusion

To improve the scalability of assembly synthesis with parallelization, we present a novel parallel assembly synthesis system, PASSES. PASSES uses domain knowledge of assembly language to parallelize synthesis problems. It identifies multiple approaches to subspace creation, using this domain knowledge. We describe the PASSES subspace creation in terms of the three properties, *collective exhaustivity*, *mutual exclusivity*, and *subspace size equality*, and introduce five complementary and reusable heuristics to improve assembly synthesis performance using parallelism. We evaluate the performance of PASSES with general bit manipulation problems and machine-dependent code from pre-existing operating systems, showing that, compared to the state-of-the-art automated assembly synthesizer and SMT parallelization approaches, the heuristics in PASSES significantly improve assembly synthesis scalability for various realistic assembly programming problems.

References

1. Alur, R., Bodik, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Proceedings of the 2013 Formal Methods in Computer-Aided Design. pp. 1–8. FMCAD '13, IEEE, Portland, OR, USA (10 2013). https://doi.org/10.1109/FMCAD.2013.6679385
2. Alur, R., Černý, P., Radhakrishna, A.: Synthesis through unification. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. pp. 163–179. Springer International Publishing, Cham (2015)
3. Alur, R., Radhakrishna, A., Udupa, A.: Scaling enumerative program synthesis via divide and conquer. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 319–336. Springer Berlin Heidelberg, Berlin, Heidelberg (2017)
4. Alur, R., Radhakrishna, A., Udupa, A.: Scaling enumerative program synthesis via divide and conquer. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 319–336. Springer Berlin Heidelberg, Berlin, Heidelberg (2017)
5. Barman, S., Bodik, R., Chandra, S., Torlak, E., Bhattacharya, A., Culler, D.: Toward tool support for interactive synthesis. In: 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!). p. 121–136. Onward! 2015, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2814228.2814235
6. Basin, D., Deville, Y., Flener, P., Hamfelt, A., Nilsson, J.: Synthesis of programs in computational logic. Program Development in Computational Logic **3049**, 30–65 (01 2004). https://doi.org/10.1007/978-3-540-25951-0_2
7. Baumann, A., Barham, P., Dagand, P.E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., Singhania, A.: The multikernel: A new os architecture for scalable multicore systems. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. p. 29–44. SOSP '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1629575.1629579
8. Brummayer, R., Biere, A.: Boolector: An efficient smt solver for bit-vectors and arrays. In: Kowalewski, S., Philippou, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 174–177. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
9. Buchi, J.R., Landweber, L.H.: Solving sequential conditions by finite-state strategies. Transactions of the American Mathematical Society **138**, 295–311 (1969)
10. Cempron, J.P., Salinas, C.S., Uy, R.L.: Assembly program performance analysis metrics: Instructions performed and program latency exemplified on loop unroll. Philippine Journal of Science **147**(3), 441–452 (2018)
11. Chennupati, G., Azad, R.M.A., Ryan, C., Eidenbenz, S., Santhi, N.: Synthesis of Parallel Programs on Multi-Cores, pp. 289–315. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-78717-6_12
12. Cypher, A.: Eager: Programming repetitive tasks by example. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. p. 33–39. CHI '91, Association for Computing Machinery, New York, NY, USA (1991). https://doi.org/10.1145/108844.108850
13. David, C., Kroening, D.: Program synthesis: challenges and opportunities. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences **375**(2104), 20150403 (2017)

14. Farzan, A., Nicolet, V.: Phased Synthesis of Divide and Conquer Programs, p. 974–986. PLDI '21, Association for Computing Machinery, New York, NY, USA (2021)
15. Flener, P., Partridge, D.: Inductive programming. Automated Software Engineering **8**(2), 131–137 (Apr 2001). https://doi.org/10.1023/A:1008797606116
16. Gulwani, S.: Programming by examples. Dependable Software Systems Engineering **45**(137), 3–15 (2016)
17. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 62–73. PLDI '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1993498.1993506
18. Gulwani, S., Polozov, A., Singh, R.: Program Synthesis, vol. 4. NOW, Hanover, MA, USA (August 2017)
19. Hamadi, Y., Jabbour, S., Sais, L.: Manysat: solver description. learning **4**, 5 (2008)
20. Hamadi, Y., Jabbour, S., Sais, L.: Manysat: a parallel sat solver. Journal on Satisfiability, Boolean Modeling and Computation **6**(4), 245–262 (2010)
21. Holland, D.A., Hu, J., Kawaguchi, M., Lu, E., Chong, S., Seltzer, M.I.: Aquarium: Cassiopea and alewife languages (2019). https://doi.org/10.48550/ARXIV.1908.00093
22. Holland, D.A., Lim, A.T., Seltzer, M.I.: A new instructional operating system. In: Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education. p. 111–115. SIGCSE '02, Association for Computing Machinery, New York, NY, USA (2002). https://doi.org/10.1145/563340.563383
23. Hu, J., Lu, E., Holland, D.A., Kawaguchi, M., Chong, S., Seltzer, M.I.: Trials and tribulations in synthesizing operating systems. In: Proceedings of the 10th Workshop on Programming Languages and Operating Systems. p. 67–73. PLOS '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3365137.3365401
24. Hu, J., Lu, E., Holland, D.A., Kawaguchi, M., Chong, S., Seltzer, M.I.: Towards porting operating systems with program synthesis. ACM Trans. Program. Lang. Syst. (sep 2022). https://doi.org/10.1145/3563943, just Accepted
25. Hu, J., Vaithilingam, P., Chong, S., Seltzer, M., Glassman, E.L.: Assuage: Assembly Synthesis Using A Guided Exploration, p. 134–148. Association for Computing Machinery, New York, NY, USA (2021)
26. Jeon, J., Qiu, X., Solar-Lezama, A., Foster, J.S.: Adaptive concretization for parallel program synthesis. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. pp. 377–394. Springer International Publishing, Cham (2015)
27. Jeon, J., Qiu, X., Solar-Lezama, A., Foster, J.S.: Adaptive concretization for parallel program synthesis. In: Proceedings of the 2015 Computer Aided Verification. pp. 377–394. Springer International Publishing, Cham (2015)
28. Jeon, J., Qiu, X., Solar-Lezama, A., Foster, J.S.: An empirical study of adaptive concretization for parallel program synthesis. Form. Methods Syst. Des. **50**(1), 75–95 (mar 2017). https://doi.org/10.1007/s10703-017-0269-8
29. Jha, S., Seshia, S.A.: A theory of formal synthesis via inductive learning. Acta Inf. **54**(7), 693–726 (nov 2017). https://doi.org/10.1007/s00236-017-0294-5
30. Le Frioux, L., Baarir, S., Sopena, J., Kordon, F.: PaInleSS: a Framework for Parallel SAT Solving. In: The 20th International Conference on Theory and Applications of Satisfiability Testing. Lecture Notes in Computer Science, vol. 10491, pp. 233–250. Springer, Melbourne, Australia (Aug 2017). https://doi.org/10.1007/978-3-319-66263-3_15

31. Le Frioux, L., Baarir, S., Sopena, J., Kordon, F.: Modular and efficient divide-and-conquer sat solver on top of the painless framework. In: Vojnar, T., Zhang, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 135–151. Springer International Publishing, Cham (2019)
32. Manna, Z., Waldinger, R.: A deductive approach to program synthesis. ACM Trans. Program. Lang. Syst. **2**(1), 90–121 (jan 1980). https://doi.org/10.1145/357084.357090
33. Massalin, H.: Superoptimizer: A look at the smallest program. SIGARCH Comput. Archit. News **15**(5), 122–126 (oct 1987). https://doi.org/10.1145/36177.36194
34. Massalin, H.: Superoptimizer: A look at the smallest program. In: Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems. p. 122–126. ASPLOS II, IEEE Computer Society Press, Washington, DC, USA (1987). https://doi.org/10.1145/36206.36194
35. Mohr, E., Kranz, D., Halstead, R.: Lazy task creation: a technique for increasing the granularity of parallel programs. vol. 2, pp. 264–280 (1991). https://doi.org/10.1109/71.86103
36. Myers, B.A.: Creating user interfaces using programming by example, visual programming, and constraints. ACM Trans. Program. Lang. Syst. **12**(2), 143–177 (apr 1990). https://doi.org/10.1145/78942.78943
37. Partridge, D.: The case for inductive programming. Computer **30**(1), 36–41 (jan 1997). https://doi.org/10.1109/2.562924
38. Polozov, O., Gulwani, S.: Flashmeta: A framework for inductive program synthesis. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. p. 107–126. OOPSLA '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2814270.2814310
39. Reddy, M.: Chapter 7 - performance. In: Reddy, M. (ed.) API Design for C++, pp. 209–240. Morgan Kaufmann, Boston (2011). https://doi.org/https://doi.org/10.1016/B978-0-12-385003-4.00007-5
40. Reisenberger, C.: PBoolector: a parallel SMT solver for QF_BV by combining bit-blasting with look-ahead. Ph.D. thesis, Master's thesis, Johannes Kepler Univesität Linz, Linz, Austria (2014)
41. Rodgers, D.P.: Improvements in multiprocessor system design. In: Proceedings of the 12th Annual International Symposium on Computer Architecture. p. 225–231. ISCA '85, IEEE Computer Society Press, Washington, DC, USA (1985)
42. Schkufza, E., Sharma, R., Aiken, A.: Stochastic superoptimization. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. p. 305–316. ASPLOS '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2451116.2451150
43. Smith, D.R.: The design of divide and conquer algorithms. Science of Computer Programming **5**, 37–58 (1985). https://doi.org/https://doi.org/10.1016/0167-6423(85)90003-6
44. Smith, D.R.: Top-down synthesis of divide-and-conquer algorithms. Artif. Intell. **27**(1), 43–96 (sep 1985). https://doi.org/10.1016/0004-3702(85)90083-9
45. Solar-Lezama, A.: The sketching approach to program synthesis. In: Hu, Z. (ed.) Programming Languages and Systems. pp. 4–13. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
46. Solar-Lezama, A., Jones, C.G., Bodik, R.: Sketching concurrent data structures. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language

Design and Implementation. p. 136–148. PLDI '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1375581.1375599
47. Solar-Lezama, A., Jones, C.G., Bodik, R.: Sketching concurrent data structures. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 136–148. PLDI '08, ACM, New York, NY, USA (2008). https://doi.org/10.1145/1375581.1375599
48. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems. p. 404–415. ASPLOS '06, Association for Computing Machinery, New York, NY, USA (2006). https://doi.org/10.1145/1168857.1168907
49. Srinivasan, V., Reps, T.: Synthesis of machine code from semantics. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 596–607. PLDI '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2737924.2737960
50. Srinivasan, V., Sharma, T., Reps, T.: Speeding up machine-code synthesis. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. p. 165–180. OOPSLA '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2983990.2984006
51. Warren, H.S.: Hacker's Delight. Addison-Wesley Professional, Boston, MA, USA, 2nd edn. (2012)
52. Wintersteiger, C.M., Hamadi, Y., Moura, L.: A concurrent portfolio approach to smt solving. In: Proceedings of the 21st International Conference on Computer Aided Verification. p. 715–720. CAV '09, Springer-Verlag, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_60
53. Zhang, J., Cambronero, J.P., Gulwani, S., Le, V., Piskac, R., Soares, G., Verbruggen, G.: Pydex: Repairing bugs in introductory python assignments using llms. Proc. ACM Program. Lang. **8**(OOPSLA1) (apr 2024). https://doi.org/10.1145/3649850
54. Zhang, J., Li, D., Kolesar, J.C., Shi, H., Piskac, R.: Automated feedback generation for competition-level code. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. ASE '22, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3551349.3560425
55. Zhang, J., Piskac, R., Zhai, E., Xu, T.: Static detection of silent misconfigurations with deep interaction analysis. Proc. ACM Program. Lang. **5**(OOPSLA) (oct 2021). https://doi.org/10.1145/3485517

A PASSES Heuristic Algorithms

A.1 PASSES Algorithm

Algorithm 1 shows pseudo code for PASSES. It first initializes the list of instances with different heuristics in line 4 and then parallelizes the synthesis for each instance in line 5, by invoking multiple individual synthesizers. Given the specification, a machine model, and a list of constraints, in function *synth_instance* (lines 9 – 15), `Synthesizer` either successfully generates a target program *prog* in line 12 or gets a synthesis failure in line 13. Depending on the heuristic, PASSES handles the failure differently as shown in line 15. PASSES executes all running

instances in parallel, and if one of them successfully produces a program, it returns the first result in line 7; otherwise, it waits until all instances finish with a synthesis failure in line 8.

Algorithm 1: The system, PASSES

```
   Input  : A specification spec, and a machine model model.
   Output : The synthesis result { SUCCESS(Program), FAILURE }.
 1 begin
 2   result ← UNKNOWN
 3   failed_counts ← 0
 4   initialize the list of instances to be synthesized
     // instance: sub-problem with a machine model and a set of
        constraints
 5   foreach instance i ∈ instances do in parallel synth_instance(i)
 6   wait until
 7     if result = SUCCESS (prog) then return SUCCESS(prog)
 8     if failed_counts = the total amount of synthesized instances then
         return FAILURE
 9 Function synth_instance (i: instance) is
10   remove i from the list of instances
     // synthesize with specification, machine model and possible
        constraints
11   switch Synthesizer(spec, i) do
12     case SUCCESS(prog) do result ← SUCCESS(prog)
13     case FAILURE do
14       failed_counts ← failed_counts + 1
15       (add new instance(s) to the list of instances with incremental
          subspace creation)
```

A.2 Constraint-SC Algorithm

Algorithm 2 shows the pseudo code for the instance initialization in *Constraint-SC*. In this code example, we initialize synthesis instances by forcing the first instruction in the target program to be different types (lines 3 – 4); one could imagine creating more detailed subspaces with constraints on other instruction locations, which further prunes the search space, expediting synthesis. In Section 5, we first apply instruction type constraints on the first instruction location. Since different instruction types can contain different numbers of instructions, and there are six categories of instruction types, to enable parallelism with more synthesis sub-problems and encourage *subspace size equality*, we take the cross product of the types of the first two instructions to produce $6^2 = 36$ groups, and merge parts of them to create groups with roughly equal sizes.

Algorithm 2: Instance Initialization in *Constraint-SC*

Input : A machine model *model*.
Output : A list of *instances*.
1 *instances* ← [ ]
2 **foreach** *instruction type* $t \in$ *model* **do**
3   *constraints* ← "the 1st instruction of the target program must belong to t"
4   append (*model*, *constraints*) to *instances*
5 **return** *instances*

A.3 Simpl-SC Basic Algorithm

Algorithm 3 shows the pseudo code for the instance initialization in *Simpl-SC*. We first divide the given complete machine model, *complete-model*, into a list of models with smaller instruction sets, *sub-models*, in line 2; each sub-model represents a subspace of the search space of the original synthesis problem;. We discuss model division in more detail later in this section. We refer to the synthesis problems for these sub-models as *sub-model instances* in line 4.

To preserve *collective exhaustivity*, *Simpl-SC* also adds a single synthesis instance with the complete-model, referred to as the *complete-model instance* (line 6), which allows the synthesizer to explore other available search spaces in the original problem. In other words, this guarantees that all subspaces explored by the complete-model and sub-model instances, exhaustively cover the entire search space. To avoid duplicate searches, we generate a set of mutually exclusive constraints (line 5) and apply them to the complete-model instance; those constraints force the synthesizer to produce a program with one or more instructions outside all possible sub-models. With those constraints, the complete-model instance precludes all subspaces searched by the sub-model instances and minimizes overlaps in the search spaces. Note that the complete-model instance preserves *mutual exclusivity* from all other sub-model instances, while those sub-model instances may not be mutually exclusive from each other.

A.4 Inc-SC Algorithm

Algorithm 4 shows the pseudo code for the function *synth_ instance* in *Inc-SC* with synthesis failure handling shown in lines 5 – 10. *Inc-SC* first enlarges the machine model used in the failed instance into a list of new models (line 7) and then creates new instances with enlarged models and mutually exclusive constraints (lines 8 – 10). The model enlargement extends the given model with at least one more different instruction type from the complete machine model; in other words, whenever there is an instance with a x-type ($x \geq 2$) model failure, *Inc-SC* enlarges this x-type model into several $(x + 1)$-type models and launches the corresponding sub-model instances in parallel. The mutually exclusive constraints (line 9) guarantee that the target program cannot be synthesized with only x instruction types, i.e., the target program must include all $(x + 1)$ types in its enlarged model; they utilize *mutual exclusivity* to assure that the search space

Algorithm 3: Instance Initialization in *Simpl-SC*

```
  Input  : A machine model complete-model.
  Output : A list of instances.
1 instances ← [ ];  constraints ← [ ]
2 divide complete-model into a list of sub-models     // with RandSimpl-SC or
    TypeSimpl-SC
3 foreach sub-model ∈ sub-models do
4 |  append (sub-model, [ ]) to the list of instances  // sub-model instances
5 |  append "∃ instruction i in the target program, s.t. i ∈ complete-model and
  |    i ∉ sub-model" to the list of constraints  // avoid duplicate searches
6 append (complete-model, constraints) to the list of instances
    // complete-model instance
7 return instances
```

Algorithm 4: *synth_ instance(·)* in *Inc-SC*

```
1  Function synth_instance (i: instance) is
2  |  remove i from the list of instances
3  |  switch Synthesizer(spec, i) do
4  |  |  case SUCCESS(prog) do result ← SUCCESS(prog)
5  |  |  case FAILURE do
6  |  |  |  failed_counts ← failed_counts + 1
7  |  |  |  generate a list of enlarged-models, where each element
   |  |  |    enlarged-models = model (the machine model in instance i) + { at
   |  |  |    least one different instruction }
8  |  |  |  foreach enlarged_model ∈ enlarged_models do
9  |  |  |  |  constraints ← "∃ instruction x in the target program, s.t.
   |  |  |  |    x ∈ enlarged-model and x ∉ model"        // avoid duplicate
   |  |  |  |    searches
10 |  |  |  |  append (enlarged-model, constraints) to the list of instances
```

explored by previously failed instances will not be reconsidered. Compared with the original complete-model, these enlarged models are still relatively smaller and their sub-model instances are easier to be evaluated by the synthesizer. Note that though *Inc-SC* does not preserve *collective exhaustivity* at the beginning, given that there may exist multiple programs that satisfy the specification, incremental model enlargement ensures that the solution will be eventually found.

B Bit Manipulation Benchmark Examples

Figure 4 describes all the examples we used. The examples, numbered P1a – P14b, are grouped by similarity. Some examples include signed or unsigned comparisons, such as P11a and P14a, while others involve shifting with 0-fill (logical shift) or sign-fill (arithmetic shift), such as P8a and P12a. We mark the type over each comparison and shift symbol.

Table 5 reports the number of instructions in each synthesized implementation by Aquarium, indicating the minimum length requirement for each benchmark.

Benchmark	P1a	P1b	P2a	P2b	P3a	P3b	P4a	P4b	P5a	P5b	P6	P7a	P7b	P8a
Length (loc)	2	2	2	2	3	3	2	3	2	3	4	2	2	3

Benchmark	P8b	P9	P10	P11a	P11b	P11c	P12a	P12b	P13a	P13b	P14a	P14b
Length (loc)	3	3	3	3	3	3	4	4	3	4	5	3

Table 5: Size of the bit manipulation benchmarks with MIPS implementations (lines of code).

P1a(x): Turn-off the right most 1-bit in a word.
$x \ \& \ (x-1)$

P1b(x): Turn-on the rightmost 0-bit in a word.
$x \mid (x+1)$

P2a(x): Turn-off the trailing 1's in a word.
$x \ \& \ (x+1)$

P2b(x): Turn-on the trailing 0's in a word.
$x \mid (x-1)$

P3a(x): Create a word with a single 1-bit at the position of the rightmost 0-bit in x.
$\neg x \ \& \ (x+1)$

P3b(x): Create a word with a single 0-bit at the position of the rightmost 1-bit in x.
$\neg x \mid (x-1)$

P4a(x): Create a word with 1's at the position of the trailing 0's in x.
$\neg x \ \& \ (x-1)$ *or* $\neg(x \mid -x)$

P4b(x): Create a word with 0's at the position of the trailing 1's in x.
$\neg x \mid (x+1)$

P5a(x): Isolate the rightmost 1-bit.
$\neg x \ \& \ x$

P5b(x): Isolate the rightmost 0-bit.
$\neg(-x \ \& \ (x+1))$

P6(x): Turn-off the rightmost contiguous string of 1's.
$((x \mid (x-1))+1) \ \& \ x$

P7a(x): Create a word with 1's at the position of the rightmost 1-bit and the trailing 0's in x.
$x \oplus (x-1)$

P7b(x): Create a word with 0's at the position of the rightmost 0-bit and the trailing 1's in x.
$x \oplus (x+1)$

P8a(x): Absolute Value Function.
$y \leftarrow x \overset{\text{arithmetic}}{>>} 31$
$(x \oplus y)-y$ *or* $(x+y) \oplus y$

P8b(x): Negative Absolute Value Function.
$y \leftarrow x \overset{\text{arithmetic}}{>>} 31$
$y-(x \oplus y)$ *or* $(y-x) \oplus y$

P9(x): Sign Function.
$(x \overset{\text{arithmetic}}{>>} 31) \mid (-x \overset{\text{logical}}{>>} 31)$

P10(x, y): Three-Valued Compare Function.
$(x \overset{\text{signed}}{>} y) - (x \overset{\text{signed}}{<} y)$

P11a(x, y): Max Function.
$((x \oplus y) \ \& \ -(x \overset{\text{signed}}{\geq} y)) \oplus y$

P11b(x, y): Min Function.
$((x \oplus y) \ \& \ -(x \overset{\text{signed}}{\leq} y)) \oplus y$

P11c(x, y): Doz Function (difference or zero).
$(x-y) \ \& \ -(x \overset{\text{signed}}{\geq} y)$

P12a(x, y): Floor of average of two integers without overflowing.
$(x \ \& \ y)+((x \oplus y) \overset{\text{logical}}{>>} 1)$

P12b(x, y): Ceil of average of two integers without overflowing.
$(x \mid y)-((x \oplus y) \overset{\text{logical}}{>>} 1)$

P13a(x, y): Exchange two registers without using a third.
$x \leftarrow x \oplus y$
$y \leftarrow y \oplus x$
$x \leftarrow x \oplus y$

P13b(x, m, k): Exchange two fields A and B of the same register x where m is a mask with 1's in field B and k is the shift distance (the number of bits from end of A to end of B).
$t_1 = (x \ \& \ m) << k$
$t_2 = (x \overset{\text{logical}}{>>} k) \ \& \ m$
$x' = (x \ \& \ m') \mid t_1 \mid t_2$
(m' is a mask that isolates fields other than A and B in register x.)

P14a(x, y): Test if $nlz(x) == nlz(y)$ where nlz is the number of leading zeroes.
$(x \oplus y) \overset{\text{unsigned}}{\leq} (x \ \& \ y)$

P14b(x, y): Test if $nlz(x) < nlz(y)$ where nlz is the number of leading zeroes.
$(\neg y \ \& \ x) \overset{\text{unsigned}}{>} y$

Fig. 4: Bit manipulation benchmark examples (26 in total).