HieraSynth: A Parallel Framework for Complete Super-Optimization with Hierarchical Space Decomposition

SIRUI LU, University of Washington, USA
RASTISLAV BODÍK, Google DeepMind, USA

Modern optimizing compilers generate efficient code but rarely achieve theoretical optimality, often necessitating manual fine-tuning. This is especially the case for processors with vector instructions, which can grow the instruction set by an order of magnitude. Super-optimizers can synthesize optimal code, but they face a fundamental scalability constraint: as the size of the instruction set increases, the length of the longest synthesizable program decreases rapidly.

To help super-optimizers deal with large instruction sets, we introduce HieraSynth, a parallel framework for super-optimization that decomposes the problem by hierarchically partitioning the space of candidate programs, effectively decreasing the instruction set size. It also prunes search branches when the solver proves unrealizability, and explores independent subspaces in parallel, achieving near-linear speedup. HieraSynth is sufficiently efficient to run to completeness even on many hard problems, which means that it exhaustively explores the program space. This ensures that the synthesized program is optimal according to a cost model.

We implement HieraSynth as a library and demonstrate its effectiveness with a RISC-V Vector super-optimizer capable of handling instruction sets with up to 700 instructions while synthesizing 7–8-instruction programs. This is a significant advancement over previous approaches that were limited to 1–3 instructions with similar instruction set sizes. Specifically, HieraSynth can handle instruction sets up to 10.66× larger for a given program size, or synthesize up to 4.75× larger programs for a fixed instruction set. Evaluations show that HieraSynth can synthesize code surpassing human-expert optimizations and significantly reduce synthesis time, making super-optimization more practical for modern vector architectures.

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

Additional Key Words and Phrases: Program Synthesis, Super-Optimization, Component-Based Synthesis, SMT, RISC-V, RVV, RISC-V Vector

ACM Reference Format:
Sirui Lu and Rastislav Bodík. 2025. HieraSynth: A Parallel Framework for Complete Super-Optimization with Hierarchical Space Decomposition. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 384 (October 2025), 28 pages. https://doi.org/10.1145/3763162

1 Introduction

Modern compilers generate performant code but are not designed to generate the fastest possible code. For performance-critical applications, experts still hand-craft hardware-specific assembly. These kernels are especially challenging for complex vector instruction set architectures (ISAs), whose numerous semantic variants create non-obvious combinatorial optimization opportunities. Our analysis of expert-written RISC-V Vector libraries reveals a 2–4× performance gap between hand-optimized code and the theoretical optimum (see our case study with the Highway library [16] in Section 7.6).

Authors' Contact Information: Sirui Lu, University of Washington, Seattle, USA, siruilu@cs.washington.edu; Rastislav Bodík, Google DeepMind, Mountain View, USA, rastislavb@google.com.

ACM 2475-1421/2025/10-ART384
https://doi.org/10.1145/3763162

Super-optimizers [24, 29, 32, 37, 40] help close this gap by searching for instruction sequences with improved performance. Unlike traditional compilers, which heuristically rewrite programs, super-optimizers search a space of candidate programs to find one that is correct and faster than a goal specified by a cost model (e.g., an approximate cycle count). A super-optimizer is *complete* if it searches this space exhaustively. Combined with an accurate cost model, this completeness proves that the generated program is optimal, assuring experts that further manual tuning would be futile.

A super-optimizer is useful only if it finishes its search in a reasonable time. Two parameters determine the program space size—and hence the computational challenge:

(1) *(Output) program size k*: Maximum instructions per candidate program.
(2) *Instruction set size n*: Number of distinct ISA instructions available.

These parameters define a program space with $\Omega(n^k)$ candidates, which makes exhaustive enumeration practical only if k, n, or both are restricted. Two complete super-optimizers illustrate the opposite ends of the spectrum: Brahma [15, 24] synthesizes programs up to 16 instructions long, but only by restricting the instruction set to a small, expert-chosen subset. In contrast, Minotaur [29] handles an x86 SIMD ISA, which includes hundreds of instructions, but is limited to synthesizing single-instruction programs in practice. This strong inverse relationship between instruction set size n and feasible program size k is evident across Lens [37], Souper [40], and other complete super-optimizers. This trade-off severely limits super-optimization for modern vector ISAs, which require both large instruction sets and multi-instruction sequences.

An common approach to mitigate the limitation is to decompose the problem along the k dimension. For example, Lens [37] includes a peephole-style [33] variant that optimizes smaller program windows independently. Unfortunately, some optimal programs cannot be constructed by concatenating independently optimized windows because all k instructions must cooperate closely in the computation. Another approach is to abandon the exhaustive search. Stochastic search (STOKE [41]), machine learning (DreamCoder [13]), and equality saturation (Diospyros [47]) can handle larger programs, but cannot guarantee optimality.

These limitations raise a key question: Can we decompose the program space without sacrificing completeness? Our answer is to shift decomposition from program size (k) to instruction selection (n) using a *hierarchical* and *adaptive* approach. We partition the program space into subspaces, each containing a progressively smaller set of instructions. This decomposition preserves both long program synthesis (sufficient k) and full ISA coverage (large n), ensuring optimality.

However, a successful decomposition on n requires careful control of subspace size, which directly determines feasibility. The smallest possible subspaces would contain multisets of exactly k instructions—just enough to build a k-instruction program. But this fine-grained decomposition creates $C(n + k - 1, k)$ subspaces (drawing a k-sized multiset from n instructions with replacement); for $k = 8$ and $n = 400$, this exceeds 10^{18} subspaces, making exhaustive exploration impossible. Using larger subspaces reduce this count, but they become harder for the synthesizer to solve. Ideally, subspaces should be as large as possible while remaining solvable. Previous work on instruction set decomposition [17] has developed static partitioning policies, but static approaches cannot adapt to problem difficulty, which varies across problem instances.

Our decomposition is adaptive: it begins with the full instruction set and recursively splits subspaces until each becomes solvable by the base synthesizer. This top-down approach creates only as many subspaces as needed, adapting to the difficulty of each synthesis problem while preserving completeness. It also prunes entire subspaces when a solver proves them unrealizable. We develop these ideas in HieraSynth, a parallel framework that uses decomposition to achieve a qualitative leap in the scalability of super-optimization. HieraSynth substantially improves on the

k-vs-n trade-off, enabling the discovery of low-cost code that surpasses human-expert optimizations. In many cases, HieraSynth is complete, proving that it has found an optimal program.

Our approach builds on three core capabilities:

(1) *Completeness-preserving hierarchical decomposition*: Our framework enables adaptive, hierarchical decomposition on n (rather than k), dividing the space into manageable subspaces without sacrificing completeness.
(2) *Efficient unrealizability detection*: An underlying solver that can quickly prove the unrealizability of a subspace is key to pruning entire branches, which avoids the need to further partition the subspace.
(3) *Independent parallel exploration*: Decomposing based on instruction selection creates independent subproblems, achieving near-linear parallel speedup that is difficult to achieve with peephole-style approaches.

To implement these capabilities, we designed a program space representation that combines component-based synthesis [15, 24] with program sketching [42, 43], embedding multiple instruction choices within each component. This representation enables our systematic hierarchical partitioning. While our system uses a component-based synthesizer, our hierarchical partitioning framework is general and should work with any base synthesizer that can efficiently prove unrealizability.

To evaluate our approach, we implement HieraSynth as a library and develop a prototype super-optimizer for RISC-V Vector (RVV) [49]. We chose RVV because its flexible and complex design offers rich optimization opportunities while posing significant challenges for both human experts and automated tools. Our results demonstrate that HieraSynth substantially improves on the k-vs-n trade-off, handling instruction sets of up to $n \approx 700$ while synthesizing programs of 7–8 instructions. Specifically, we estimate that HieraSynth can handle up to 10.66× larger n for a given k, or up to 4.75× larger k for a fixed n. This represents a significant leap in capability, enabling the synthesis of complex vector kernels that were previously out of reach for complete super-optimizers.

In summary, this paper makes the following contributions:

(1) A novel approach to component-based synthesis that compactly encodes large program spaces by embedding choices within components, enabling hierarchical decomposition.
(2) A systematic adaptive approach to partition the program space on instruction set size n rather than program size k, preserving completeness while enabling large-scale parallelism.
(3) A generic, parallel framework, HieraSynth, for building complete super-optimizers that can be adapted to different ISAs and base synthesizers.
(4) The first complete super-optimizer for the flexible RISC-V vector ISA, demonstrating HieraSynth's scalability, discovering optimizations that surpass human experts, and, in many cases, proving their optimality.

The remainder of the paper is organized as follows: Section 2 provides an overview of HieraSynth, Section 3 discusses our program space representation and the parallel divide-and-conquer approach, Section 4 describes further refinements to our algorithm, Section 5 formulates the extended component-based synthesis algorithm for solving individual program spaces, Section 6 discusses adapting HieraSynth to build an RVV super-optimizer, and Section 7 presents empirical evaluations demonstrating HieraSynth's effectiveness in discovering low-cost and often cost-optimal code for real-world applications.

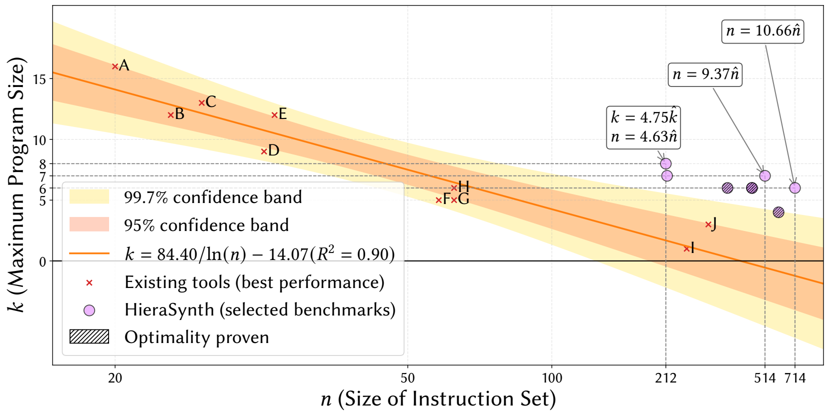


Fig. 1. The trade-off between maximum program size k and instruction set size n shown for a range of super-optimization systems. Points marked × show the best-reported performance for prior tools, revealing an inverse relationship between n and the achievable k, shown as the performance frontier (solid red line). Circles show HieraSynth's results on selected challenging benchmarks. By partitioning along the n dimension, HieraSynth surpasses the existing performance frontier, handling up to 10.66× larger n for a given k, or up to 4.75× larger k for a fixed n. Additionally, HieraSynth's focus on complete search allows it to prove that the synthesized program is optimal; hatched markers denote proven optimality. Prior work: A: Brahma [15, 24], B: Lens [37] on GA144 ISA, C: Massalin's original super-optimizer [32], D: Barthe et al. [7], E: our base synthesizer based on Brahma, F: Aquarium [18, 19] and PASSES [17], G: Lens on ARMv7, H: Souper [40], I: Minotaur [29], J: Bansal and Aiken's peephole super-optimizer [6].[1]

2 Overview

In this section, we first examine the trade-off between output program size (k) and instruction set size (n) that governs existing complete super-optimizations. We then explain why decomposition on k can fail to guarantee optimality. Finally, we introduce our adaptive, hierarchical decomposition on n, which makes finding good solutions (and often also proving them optimal) practical for larger problems.

2.1 Background: The k-vs-n Trade-off

A complete super-optimizer's goal is to find a program with the lowest possible cost for a given specification, according to a cost model. It does this by systematically searching the entire space of possible programs. This completeness preserves the path to finding the optimal solution, distinguishing them from non-exhaustive approaches (e.g., stochastic search) that may find good solutions but cannot guarantee or prove optimality.

The program space of a super-optimization problem is primarily defined by program size (k) and instruction set size (n). The resulting program space grows at least as $\Omega(n^k)$, making exhaustive exploration intractable as either parameter increases. Note that this is a rough estimate and does not account for immediate constants or inter-instruction dependencies through variables or registers.

[1] Brahma restricts effective n with expert-selected components, making the program space smaller than $\Omega(n^k)$. For Souper, which adopts Brahma's approach without expert-selected components, we observe $k = 6$ with $n = 61$. For Lens, we counted the supported instruction set size from their artifact. Aquarium reported larger k for OS kernel code using domain-specific heuristics; we report Aquarium's performance as a general program synthesizer. For other works, we estimated values from their papers. We may overestimate, and the reported k and n may not be simultaneously achievable for some tools.

Decades of research show a consistent pattern: as n increases, the maximum feasible k decreases substantially. Fig. 1 illustrates this trade-off. A regression on prior work, based on the n^k estimation, reveals $k = 84.40/\ln(n) - 14.07$ with $R^2 = 0.90$. This strong relationship represents a practical capability frontier for existing techniques.

HieraSynth substantially pushes this frontier. As shown in Fig. 1, for a fixed program size like $k = 6$, HieraSynth can handle an ISA up to 10.66× larger than the empirical capability frontier. For a fixed ISA size like $n \approx 200$, HieraSynth can synthesize a program up to 4.75× larger than the estimated baseline capability. This significant improvement enables synthesizing complex vector kernels, as demonstrated in Section 7.6.

2.2 Why Decomposition on k Loses Optimality

To address scalability, most previous approaches use peephole-style optimizations [33], decomposing based on output program size k. These methods examine small instruction windows to find local improvements, apply peephole rewrites, and iterate until a fixed point.

For example, Bansal and Aiken extract windows of 6 instructions and replace them with optimal sequences of up to 3 instructions ($k = 3$ for the output program size), and Lens improves on this approach through *context-aware window decomposition*, which leverages surrounding code as preconditions and postconditions to relax correctness constraints and synthesize more efficient code. While effective for many scenarios, these approaches sacrifice global completeness and cannot synthesize programs requiring holistic, global changes. In Section 7.6, we demonstrate a vector kernel where finding the optimal implementation requires synthesizing a program of $k \geq 6$ as a single unit. The new algorithm invents a different representation of the datatypes, and such global representational changes are beyond the reach of local peephole rewrites.

2.3 Our Idea: Decompose on n to Preserve Optimality

The incompleteness of peephole-style optimization leads to a fundamental question: Is there an alternative decomposition strategy that preserves completeness while enabling scalability? A promising direction, noted by previous work, is to decompose on the instruction set size n, which involves partitioning the set of available instructions. Crucially, this approach preserves the target program size k, avoiding the need to compose solutions from smaller synthesized programs.

For example, the PASSES system [17] explores decomposition on n by statically splitting the instruction set into predefined subsets and attempting to synthesize a program using only the instructions from one subset. However, partitioning the instruction set this way loses candidate programs that require instructions from different subsets simultaneously. To maintain completeness, PASSES uses a fallback task that synthesizes with the entire instruction set, which could not scale on harder problems. Although this approach improves scalability in many cases, it does not fully solve the challenge of scaling to larger instruction sets of programs.

A more principled approach would partition the ISA in a way that creates a *complete* partition of the original program space, ensuring no candidate programs are lost. To understand how to achieve this, we analyze the program spaces of different tools in Fig. 1. For a program with large k and n, decomposing on n creates subproblems with a smaller n but the same large k. This suggests that we need a base synthesizer that excels at this type of problem. We find that Brahma is a competitive choice, as it achieves the largest k values among various existing tools. As shown in Fig. 2b, Brahma takes a multiset of generic instructions (the standard library) augmented with problem-specific instructions picked by experts. It uses these instructions as components and determines the optimal way to order and wire them. This approach is powerful, but does not scale to large instruction sets without expert guidance.

```
fn f_sketch(a, b) {
  // ?op chooses an operator
  // from add/sub/popcnt/...
  o1 = ?op(??, ??)
  o2 = ?op(??, ??)
  o3 = ?op(??, ??)
  return o3
}
```

(a) Sketch

```
fn f_brahma(a, b) {
  reorder_and_wire {
    // stdlib
    add, sub, and, ...
    // user-extended
    or, ...
  }
}
```

(b) Brahma

```
fn f_hierasynth(a, b) {
  // here k=3 and n=2
  reorder_and_wire {
    ?op from {add/sub}
    ?op from {and/or}
    ?op from {and/or}
  }
}
```

(c) HieraSynth

Fig. 2. The program space definition of HieraSynth combines the advantages of Sketch and Brahma. Sketch offers fine-grained control over the programs by choosing the operators available to each instruction slot. In contrast, Brahma uses a single instruction mix; its synthesizer reorders the set of instructions in all possible ways and wires them to pass values. HieraSynth adopts Brahma's ability to reorder the instructions "for free" with Sketch's per-slot choices. The per-slot choices let us gradually partition the program space into smaller spaces as shown in Fig. 3.

To use Brahma as the base synthesizer, the core challenge is to automatically generate the "expert-picked" instruction sets. A straightforward approach to creating a complete partition of the program space is to enumerate every possible multiset of k instructions from the full set of size n. This is computationally infeasible, as it yields $C(n+k-1,k)$ distinct multisets. For a benchmark like Min128 ($k = 8$, $n > 200$), this would generate approximately 10^{14} subspaces, which is intractable even with massive parallelization.

To address this combinatorial explosion, we augment Brahma with ideas from Sketch (Fig. 2a) and introduce a compact program space representation (Fig. 2c). Instead of working with individual instructions, our synthesizer uses groups (subsets) of instructions as components. The synthesizer's task is to select one instruction from each component and then find the optimal wiring. By enumerating combinations of these instruction groups rather than individual instructions, we can reduce the number of subspaces to roughly $C(\frac{n}{g} + k - 1, k)$, where g is the average group size.

However, this introduces a dilemma: larger components create fewer subspaces, but each synthesis task becomes harder. Conversely, using small components makes each task easier, but their sheer number becomes overwhelming. The Min128 benchmark illustrates this trade-off: components of size 40 prove too hard for the solver, while components of size 13 give an impractical 7.4×10^5 subspaces. Furthermore, synthesis difficulty correlates only loosely with the size of the program space: some small spaces are very hard, while some large ones can be proven unrealizable in seconds. Consequently, no fixed-size grouping can be optimal, and this necessitates a more dynamic and adaptive strategy.

2.4 Hierarchical Parallel Divide-and-Conquer on n

The key idea of our approach is to gradually partition the program space, trying to find subspaces that are of the right size and complexity for the base synthesizer to handle efficiently. Instead of a fixed, one-level partition, we create a hierarchy and adaptively break down only the most difficult parts of the program space.

We define the program space in terms of k components, which are placeholders for instructions. The synthesizer's job is to select an instruction for each component and then determine their correct and optimal ordering and wiring. Our method works by recursively partitioning the instruction choice set for each component. The process progresses as follows and is illustrated in Fig. 3:

(1) *Start and Partition.* In the initial program space, each of the k components can use any instruction from the full set of size n. This initial program space is partitioned to create

subproblems solvable in parallel, to exploit available parallel hardware. For example, the initial problem with a 4-instruction set is partitioned into four smaller subspaces (labeled 1-4) in Fig. 3.

(2) *Synthesize.* Each subspace is assigned to a base synthesizer on a parallel worker, which returns one of three outcomes:
   (a) *Unrealizable:* The solver proves no valid program exists within the subspace. This allows us to prune the entire corresponding search branch, eliminating potentially millions of subproblems without further exploration (e.g., subspaces 1 and 4).
   (b) *Success:* We get a valid solution and can continue searching within this promising subspace for a lower-cost program. (e.g., continue exploring subspace 3 for lower cost). The search is often accelerated by reusing internal lemmas in the solver.
   (c) *Timeout:* The synthesis task is still too difficult and exceeds the time limit. The subspace is added to a queue for further partitioning (e.g., subspace 2).

(3) *Recurse Adaptively.* Timed-out subspaces are recursively partitioned into even smaller problems. As shown in Fig. 3, the timed-out subspace 2 is split into 2.1 and 2.2. This process continues until all branches of the search tree have been either solved or proven unrealizable.

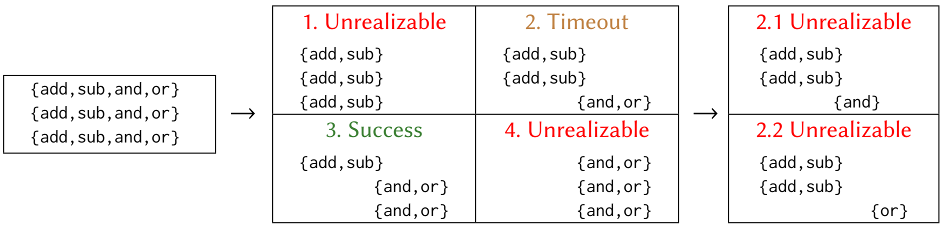


Fig. 3. Two steps of the hierarchical program space decomposition. We show three components, each with four initial instruction choices, i.e., $k = 3$, $n = 4$. We start with the unpartitioned program space on the left. We first divide it into four partitions, shown in the middle. Next, we run the base synthesizer on each subspace, which can return three outcomes: Unrealizable means the solver has proved that no valid program exists in that subspace, allowing pruning of the corresponding subspaces; Success means that a valid program was found (and will be further optimized, not shown here); Timeout means that synthesis exceeded the time limit, triggering further decomposition. In the second step, we partition the `{and,or}` subset within the timed-out subspace into two new subsets.

By adapting to problem difficulty, this hierarchical approach resolves the dilemma of subproblem size versus quantity. It starts with large, challenging problems and refines only those found to be difficult. By aggressively pruning unrealizable branches (Section 3.3), we dramatically reduce the number of subproblems that must be explored. For the MIN128 benchmark, this method reduces the number of explored subspaces from potentially millions to approximately 500, making the problem tractable on a single server (see Section 7 for details).

To make our hierarchical approach effective, we had to address several practical considerations, such as determining which subspaces to explore next and how to efficiently coordinate parallel workers. We discuss our approach to task scheduling, efficient pruning, and managing the exploration-exploitation trade-off in Sections 3 and 4.

It is worth noting that decomposition on n is a general technique that could enhance other synthesizers beyond our BRAHMA-inspired base synthesizer. The core principle, that is, a hierarchical decomposition of a large program space into manageable subspaces tackled by a base synthesizer to find optimal solutions or prove unrealizability, is a general methodology. The base synthesizer, program space representation, and unrealizability prover are, in principle, pluggable. This allows

HieraSynth to be adapted to other domains and benefit from orthogonal optimizations in other works. For example, one could use a SyGuS [5] base synthesizer, hierarchically decompose the production rule choices, and prove unrealizability using dedicated approaches [20, 27]. The rest of the paper realizes our ideas: a partitionable program space representation (Section 3.1), a parallel scheduler with pruning (Section 3.3 and 3.4), and two optimizations to the parallel solving technique (Section 4).

3 Parallel Divide-and-Conquer

In this section, we present our approach to breaking the k-vs-n trade-off. We begin with our program space representation that enables hierarchical decomposition on instruction set size (n). We then present our parallel divide-and-conquer algorithm and extend it for optimal synthesis.

3.1 Program Space Representation

As discussed in Section 2, our key innovation is to decompose the search based on instruction set size (n) rather than program size (k). This requires a program space representation that enables systematic partitioning while preserving completeness.

In our representation, a program space consists of k components, each able to choose from a set of instructions. These instructions are organized into a tree structure, where the leaf nodes are individual instructions like `+` and `-`. Related instructions are grouped into sets, annotated with sequence numbers (e.g., `{+,-}!2`) for partitioning priority. The `reorder_and_wire` notation indicates that components may be reordered and wired into a program by the synthesizer.

```
fn space(a, b) {
  reorder_and_wire {
    3 * {{+,-}!2,{*,/}!3}!1
  }
}
```

(a) Program space with $k = 3$ and $n = 4$ (operators)

```
fn f(a, b) {
  r1 = a + b
  r2 = r1 * b
  return r2 - a
}
```

(b) One concrete program from (a)

```
fn subspace_n(a, b) {
  reorder_and_wire {
    n * {+,-}!2 // n is 0~3
    (3-n) * {*,/}!3
  }
}
```

(c) Lossless subspace family parameterized by n

```
fn subspace_2_1(a, b) {
  reorder_and_wire {
    +, -, {*,/}!3
  }
}
```

(d) Refinement of the $n = 2$ subspace

Fig. 4. Lossless hierarchical partitioning of a program space with $k = 3$ and $n = 4$. (a) Three components; each slot chooses from `{+,-,*,/}`. (b) One concrete program in (a). (c) *Family* `subspace_n` ($n \in \{0, 1, 2, 3\}$): exactly n slots are restricted to `{+,-}`, and the remaining $3 - n$ to `{*,/}`; the children partition the parent (lossless). (d) Refinement of the $n = 2$ case by further partitioning the `{+,-}` choices; one child subspace is shown.

Our representation enables systematic partitioning of the program space while preserving completeness. To illustrate, consider the example in Fig. 4, where we have a program space with $k = 3$ components, each able to select from $n = 4$ operators `{+,-,*,/}`. We partition this space by first splitting on the choice node with the smallest sequence number (here, `!1`). This creates subspaces where components choose from different subsets of instructions. In our example, `subspace_n` represents a parameterized family of subspaces where n components choose from `{+,-}` and the remaining $3 - n$ components choose from `{*,/}`. We can further partition these subspaces for finer-grained exploration. For instance, `subspace_2_1` shows a subspace where `{+,-}!2` in `subspace_2` is further split into one `+` and one `-`. As each program in the original space appears in some subspaces, our partitioning scheme is *lossless*, preserving completeness. This hierarchical approach addresses the combinatorial explosion that would result from naively enumerating all possible combinations of instructions. Instead of generating $C(n + k - 1, k)$ subspaces immediately, we progressively refine the partitioning, solve subspaces in parallel, prune unrealizable spaces, and focus computational resources on promising areas of the search space.

3.2 Counterexample-Guided Inductive Synthesis

Before presenting our parallel algorithm, we briefly describe the Counterexample-Guided Inductive Synthesis (CEGIS) framework [42, 43] that serves as the base synthesizer for subspaces. CEGIS operates through an iterative loop with two main components:

(1) *Proposer*: Generates candidate programs that satisfy the specification on a known set of inputs. We implement this by encoding the program space and specification into SMT constraints. Our encoding, inspired by Brahma, will be discussed in Section 5.
(2) *Challenger*: Checks whether the proposed program satisfies the specification. If not, it generates counterexamples (inputs where the program violates the specification) and adds them to the known input set for the next iteration.

This iterative process continues until either:

(1) *Success*: The challenger confirms that the candidate satisfies the specification for all inputs.
(2) *Unrealizable*: The proposer proves that no program in the space satisfies the specification on the known inputs.
(3) *Timeout*: A predefined timeout is reached.

3.3 Parallel Divide-and-Conquer Algorithm

With our hierarchically partitionable program space representation and the base synthesizer established, we now present our parallel divide-and-conquer algorithm. This algorithm coordinates multiple base synthesizers (workers) to explore different subspaces simultaneously while avoiding combinatorial explosion through effective pruning strategies.

Our hierarchical partitioning forms a tree of subspaces with two fundamental properties:

(1) A child space is a subset of its parent.
(2) The union of all immediate child spaces equals the parent space.

These properties enable the following pruning strategies:

(1) *Downward pruning*: When a parent space is proven unrealizable, all its child spaces are also unrealizable and can therefore be pruned without exploration.
(2) *Upward pruning*: When all child spaces of a parent are proven unrealizable, the parent space is also unrealizable.

Algorithm 1: Global scheduler. Poll workers, prune unrealizable branches, and start new tasks; return the first solution or conclude that the space is unrealizable.

```
results ← CheckRunningNodes();
foreach node,result ← results do
    match result do
        case unrealizable → Prune(node);
        case success (prog) → return prog;
        case timeout → DoNothing();
    end
end
TryStartNodes();
if #running tasks = 0 then
    if no leaf has timed out then return unrealizable;
    return failed;
end
```

These pruning rules allow us to eliminate large portions of the search space efficiently. Importantly, we leverage an asymmetry in the synthesis process: proving unrealizability is often easier than finding a program that satisfies the specification. The intuition is that proving unrealizability requires only identifying a small set of counterexamples where no program in the program space can meet the specification on them simultaneously, and we can control the system to generate smaller counterexamples first (see the two-track synthesis in Section 4.1). In contrast, finding a program involves finding a large set of inputs to narrow down the program space into the specific program that matches the specification, where the synthesis gets harder as the solver context grows with large counterexamples. This difference enables us to quickly prune large branches of the search tree by ruling out unrealizable branches without exhaustive enumeration.

Our parallel algorithm uses a global scheduler to orchestrate the synthesis process, as shown in Algorithm 1. It runs in a loop, spawns workers for subspaces, periodically checks their results, and manages partition and exploration strategy. It prunes a subspace when a worker confirms its unrealizability. Selecting the next program space to explore is handled by Algorithm 2, which maintains two priority queues:

(1) q: Contains program spaces ready for exploration, prioritized first by depth in the tree and then by a random priority.
(2) q_s: Contains program spaces that need to be split into smaller subspaces.

When no nodes are ready for exploration (q is empty), the algorithm takes nodes from q_s and splits them, adding their children to q. Then it selects nodes from q to start new workers.

During synthesis, if any solution is found, this algorithm terminates and reports the solution. If every program space is proven unrealizable, the algorithm concludes the original space is unrealizable. If any leaf nodes have timed out, we report a synthesis failure but cannot guarantee unrealizability. For applications requiring a definitive unrealizability proof, we can disable timeouts for leaf nodes, though this may significantly increase runtime.

Algorithm 2: `TryStartNodes` algorithm for task prioritization, splitting, and spawning.

```
Data: q: priority search queue of nodes, prioritized
      by ⟨Depth, Priority⟩ lexicographically
Data: q_s: priority search queue of nodes to split
while q is empty and q_s is not empty do
    node ⟵ PopTop(q_s);
    if NotPruned(node) then
        SplitAndEnqueueChildren(node, q, q_s);
end
while q is not empty and #running tasks < #cores do
    node ⟵PopTop(q);
    if NotPruned(node) then Start(node);
end
```

Algorithm 3: Global scheduler loop body for optimal program search.

```
Data: bestProg: initialized to empty.
Data: bestCost: initialized to ∞ (or the reference
      cost).
responses ⟵ CheckRunningNodes();
foreach node,response ⟵ responses do
    match response do
        case unrealizable ⟶ Prune(node);
        case success (prog) ⟶ UpdateBestProg(prog);
        case timeout | gotNewCex ⟶ DoNothing();
    end
    if node is running then
        SyncCurrentCost(node);
end
TryStartNodes();
if #running tasks = 0 then
    match (#timed out leaves, bestProg) do
        case (0, empty) ⟶ return unrealizable;
        case (_, empty) ⟶ return failed;
        case (0, prog) ⟶ return optimal (prog);
        case (_, prog) ⟶ return maybeOptimal (prog);
    end
end
```

3.4 Search for Optimal Programs

In practice, we typically seek not just any valid program but the optimal one according to a cost model, which assigns numerical costs to programs, with lower values indicating better programs. Common costs include instruction count or estimated cycles.

We extend our scheduler in Algorithm 3 to maintain the best-known cost, initially set to infinity or the reference cost. `UpdateBestProg` compares the cost of the program with the current best cost and updates the best program and cost if the new program has a lower cost. This cost serves as a dynamic upper bound that tightens throughout the search process. To effectively propagate cost information to running workers, we modify workers to also report to the scheduler by emitting a gotNewCex whenever a new counterexample is generated during CEGIS. These reporting events create natural synchronization points to transfer the current best cost to the worker. At these points, the proposer is in a standby state, which allows us to inject additional cost constraints without disrupting the running solver.

Another change not shown in Algorithm 3 is to the worker behavior. Instead of terminating when a solution is found, the worker continues to refine the solution with lower costs. It awaits the current cost from the scheduler, injects cost constraints, and solves incrementally. In this way, it leverages the lemmas learned in synthesizing the earlier programs to get improved results faster.

Our approach provides clear optimality guarantees: if all program spaces have been explored with no running tasks remaining and no leaf nodes timed out, we can guarantee that the best-known solution is optimal. In cases where some leaves time out, we report the best solution but cannot guarantee optimality. If a definitive optimality guarantee is desired, we can disable leaf timeouts.

4 Optimizations for Parallel Solving

While our basic parallel divide-and-conquer approach already offers significant advantages and is the key to breaking the trade-off between instruction set size (n) and program size (k) by decomposing on n, we can further enhance its efficiency through additional optimizations. In this section, we introduce two key optimizations that improve the performance of our approach on complex synthesis tasks: two-track synthesis and biased search.

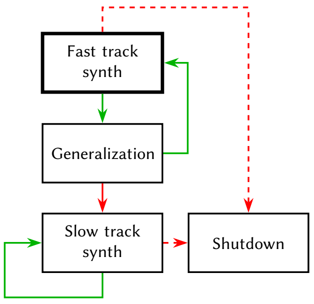


Fig. 5. Worker state machine. Green, solid arrows are transitions on success. Red arrows are transitions on failure. Fast track

4.1 Two-Track Synthesis

As described earlier, we time out workers to avoid wasting resources on program spaces unlikely to yield results in a reasonable amount of time. However, this approach risks premature termination of promising tasks. To mitigate this, we use a two-track approach to identify promising spaces and adjust timeouts accordingly. The worker's state machine under this approach is illustrated in Fig. 5.

The key insight behind our two-track approach is that varying the verification strictness allows quick identification of promising program spaces before dedicating additional resources. Instead of using a single challenger that performs rigid verification to ensure the equivalence between the specification and the proposed program, we enhance our CEGIS implementation with two classes of challengers:

(1) *Easy challengers*: These operate with restricted data ranges or smaller bit-widths. The synthesis on these restricted inputs will be faster, but the result is not guaranteed to be correct on all possible inputs. However, failing to synthesize a program against easy challengers indicates that the program space is unrealizable and can be pruned.
(2) *Hard challengers*: These utilize full data ranges and bit-widths. A program that passes all hard challengers is guaranteed to satisfy the complete specification.

Our synthesis proceeds through three phases using these challengers:

(1) *Fast-track synthesis*: The worker initially synthesizes using only easy challengers. This phase quickly identifies promising program spaces while filtering out obviously unrealizable ones.
(2) *Generalization*: If fast-track synthesis succeeds, instead of directly starting to use hard challengers to generate more counterexamples, we attempt to generalize the fast-track result to a smaller program space. For example, we may use a sketch (not component-based) program with the same structure as the fast-track synthesis result but with symbolic immediates. We then synthesize using this smaller program space with the hard challengers, which is typically faster than synthesis with the full program space.

(3) *Slow-track synthesis*: If generalization fails, we fall back to synthesis with the original program space and hard challengers. This guarantees the completeness of our approach.

To guarantee soundness, we never accept a program until it passes *all hard* challengers. After the fast track successfully finds a solution, we deem this program space as promising and increase its timeout to allocate more resources for the generalization and slow-track phases.

4.2 Biased Search

While the two-track synthesis approach helps identify promising program spaces, we also use this information to guide parallel exploration. This insight leads to our biased search strategy, which prioritizes the children of promising program spaces.

Algorithm 4 presents our enhanced task selection strategy with biased search. Here, the algorithm maintains an additional queue q_p prioritizing nodes first by the distance to the nearest ancestor that successfully passed the fast-track. If no successful ancestor is available, we treat the distance as infinity. When selecting the next task to start, we generate a random value t, and with probability p, we select from q_p (biased search); otherwise, we select from the standard queue q. This probabilistic approach effectively balances exploitation (exploring more promising program spaces) with exploration (exploring diverse program spaces). By prioritizing children of fast-track successful nodes, we focus computational resources on the most promising search spaces and accelerate the discovery of solutions.

Algorithm 4: `TryStartNodes` algorithm with biased search, splitting omitted for brevity

```
Data: $p$: probability for biased search
Data: $q$: priority search queue of nodes, prioritized by
        ⟨Depth, Random priority⟩ lexicographically
Data: $q_p$: priority search queue of nodes, prioritized by
        ⟨Distance to the nearest fast-track-successful
        ancestor, Depth, Random priority⟩ lexicographically
while $q$ is not empty and #running tasks < #cores do
    $t \leftarrow$ UniformRandVar(0, 1);
    if $t < p$ then node ← PopTop($q_p$); Remove($q$, node);
    else node ← PopTop($q$); Remove($q_p$, node);
    if NotPruned(node) then Start(node);
end
```

5 Component-Based Synthesis with Choices in Components

In this section, we describe the base synthesizer constraint encoding for individual program spaces. Our approach extends Brahma's formalism [24], reusing its concepts of I/O variables, locations, and dataflow relations, while introducing instruction choices within each component. This introduces several new constraints and modifies some existing constraints within the Brahma framework.

5.1 Notations for Program Space Representation

We first present the formal representation of our component-based synthesis with choices. A synthesis problem consists of a specification and a program space. Following Brahma, a program's specification is a tuple $\langle \vec{I}, \vec{O}, \phi_{\text{spec}}(\vec{I}, \vec{O}) \rangle$, where:

(1) $\vec{I}$ and $\vec{O}$ are tuples of input and output variables of the program,
(2) $\phi_{\text{spec}}(\vec{I}, \vec{O})$ is an expression that specifies the desired input-output relationship.

We extend the program space to support choices in components. A program space is specified as:

$$\begin{aligned} &\langle \{ \langle C_i, S_i \rangle \mid i = 1, \cdots, N \}, S \rangle \\ \text{where } C_i &= \{ \langle \vec{I_{i,j}}, \vec{O_{i,j}}, \phi_{i,j}(\vec{I_{i,j}}, \vec{O_{i,j}}) \rangle \mid j = 0, \cdots, M_i - 1 \} \\ S_i &= \langle E_i, P_i, N_{I,i}, N_{O,i}, \vec{L_{I,i}}, \vec{L_{O,i}} \rangle \\ S &= \langle \vec{L_I}, \vec{L_O} \rangle \end{aligned}$$

This representation has two main parts:

(1) *Instruction choices*: C_i is the set of possible instructions for the i-th component. Each component can have M_i different instruction choices, where:
   - $\vec{I_{i,j}}$ and $\vec{O_{i,j}}$ are the I/O variables for instruction choice j (intermediate values),
   - $\phi_{i,j}$ specifies the relationship between inputs and outputs for this instruction.
   - Different instructions within a component may use different numbers of I/O variables.

(2) *Structural variables*: S_i and S represent what is being synthesized:
   - E_i is a Boolean that indicates whether component i is enabled
   - P_i is an integer that selects which instruction to use for component i
   - $N_{I,i}$ and $N_{O,i}$ are the number of inputs/outputs for the selected instruction
   - $\vec{L_{I,i}}$ and $\vec{L_{O,i}}$ are "location" values that encode how components connect to form a directed acyclic graph (DAG)

Intuitively, the structural variables define how we reorder and connect the instructions to form a program, and what choices we make for each component. The location values L constrain the connections between components, and two intermediate values at the same location must be the same. P and N denote the specific instruction choice for each component. The enabled bit E accommodates cases where some components cannot be connected to others due to ISA constraints.

For notational clarity, we use $\vec{V}^k$ for the element of $\vec{V}$ at index k (starting from 0). When element order is irrelevant, we use tuples as sets. We use $\theta(\vec{V})$ for the number of elements in a tuple or set.

We also introduce these intermediate notations:

- $\mathcal{C}$: all the choice sets in all the components (all C_i).
- $\mathcal{S}$: all the structural variables in the set of all S_i along with S,
- $\mathcal{I}$: all the inputs to the choices (union of all $\vec{I_{i,j}}$),
- $\mathcal{O}$: all the outputs of the choices (union of all $\vec{O_{i,j}}$),
- $\mathcal{U}$: all the uses in the program ($\mathcal{I}$ along with $\vec{O}$),
- $\mathcal{D}$: all the definitions in the program ($\mathcal{O}$ along with $\vec{I}$)

We also define a location function L that maps from $\mathcal{U} \cup \mathcal{D}$ to integers:

$$L(\vec{I}^k) = \vec{L_I}^k, \quad L(\vec{O}^k) = \vec{L_O}^k, \quad L(\vec{I_{i,j}}^k) = \vec{L_{I,i}}^k, \quad L(\vec{O_{i,j}}^k) = \vec{L_{O,i}}^k$$

This function is well defined when $\theta(\vec{L_I}) = \theta(\vec{I})$, $\theta(\vec{L_O}) = \theta(\vec{O})$, $\theta(\vec{L_{I,i}}) = \max_j \theta(\vec{I_{i,j}})$, and $\theta(\vec{L_{O,i}}) = \max_j \theta(\vec{O_{i,j}})$. We will assume that L is well-defined throughout our discussion.

5.2 Synthesis Constraints

The synthesis constraints with our program space representation can be described as:

$$\exists \mathcal{S}.\phi_{\text{wfp}}(\mathcal{S}) \wedge (\forall \vec{I}.\forall \vec{O}.\exists \mathcal{I}, \exists \mathcal{O}.\phi_{\text{conn}}(\mathcal{S}, \mathcal{D}, \mathcal{U}) \wedge \phi_{\text{lib}}(\mathcal{I}, \mathcal{O}) \wedge \phi_{\text{spec}}(\vec{I}, \vec{O}))$$

This formula states that our synthesis problem is to find the structural variables for the program space such that for all I/O pairs, a set of intermediate values exists to fulfill the constraints.

5.2.1 Encoding Well-Formed Programs. We now establish the constraints ϕ_{wfp} for the structural variables $\mathcal{S}$ to be valid. A valid assignment to these variables leads to a well-formed program in static single-assignment (SSA) form within our program space.

First, we need to assign integers to locations in a principled way. Only equality relationships matter for location values, so we can choose a canonical assignment for them. The canonicity constraint ensures the following properties and simplifies subsequent constraints:

- Locations are assigned non-negative integers smaller than the total number of definitions.

- Program inputs receive the smallest possible numbers.
- Outputs of a component receive consecutive location values.

$$\phi_{\text{cano}} = \bigwedge_{v \in \mathcal{D} \cup \mathcal{U}} 0 \le L(v) < \theta(\mathcal{D}) \wedge \bigwedge_{0 \le k < \theta(\vec{I})} L(\vec{I}^k) = k \wedge \bigwedge_{\substack{0 \le i < \theta(C), \\ 0 \le j < \theta(C_i), \\ 0 \le k < \theta(\vec{O_{i,j}}) - 1}} L(\vec{O_{i,j}}^k) = L(\vec{O_{i,j}}^{k+1}) - 1$$

The canonicity constraint does not prevent multiple definitions from sharing the same location, which would violate SSA form. Therefore, we need a consistency constraint:

$$\phi_{\text{cons}} = \bigwedge_{x, y \in \mathcal{D}, x \ne y} L(x) \ne L(y)$$

In a well-formed program, variables must be defined before they are used, and the data dependency graph must be a DAG. Following our canonicity constraint, we ensure acyclicity by requiring that the location values for definitions are always greater than those for the uses in each component:

$$\phi_{\text{acyc}} = \bigwedge_{0 \le i < \theta(C), 0 \le j < \theta(C_i), x \in \vec{I_{i,j}}, y \in \vec{O_{i,j}}} L(x) < L(y)$$

In each component, we ensure choosing a valid choice, encoded by the choice constraint:

$$\phi_{\text{choi}} = \bigwedge_{0 \le i < \theta(C)} 0 \le P_i < \theta(C_i) \wedge N_{I,i} = \theta(\vec{I_{i,P_i}}) \wedge N_{O,i} = \theta(\vec{O_{i,P_i}})$$

Another thing to note is that each component can choose from operators with different numbers of inputs and outputs and can be disabled. This means we must ensure that all uses only reference valid, enabled outputs. Here, the antecedent states that the use x from an enabled component uses an output of the component C_d, and the consequent states that it must only use valid outputs of C_d.

$$\phi_{\text{valid}} = \bigwedge_{0 \le u, d < \theta(C), x \in \vec{I_{u,P_u}}} (E_u \wedge \vec{L_{O,d}}^0 \le L(x) < \vec{L_{O,d}}^0 + \theta(L_{O,d})) \Rightarrow (E_d \wedge L(x) - \vec{L_{O,d}}^0 < N_{O,d})$$

The well-formed constraint can then be defined as the conjunction of the five conditions:

$$\phi_{\text{wfp}} = \phi_{\text{cano}} \wedge \phi_{\text{cons}} \wedge \phi_{\text{acyc}} \wedge \phi_{\text{choi}} \wedge \phi_{\text{valid}}$$

5.2.2 *Encoding Dataflow Semantics.*

For dataflow semantics with choices, we only constrain values involved in execution based on structural variables. The input-output relation for the instructions should only apply when the instruction is chosen and the component is enabled:

$$\phi_{\text{lib}} = \bigwedge_{0 \le i < \theta(C), 0 \le j < \theta(C_i)} (E_i \wedge j = P_i) \Rightarrow \phi_{i,j}(\vec{I_{i,j}}, \vec{O_{i,j}})$$

We also need a constraint modeling connections between components such that the I/O variables should be equal when the locations are equal, the components are enabled, and the I/O variables are associated with chosen instructions.

$$\phi_{\text{conn}} = \bigwedge_{u \in \mathcal{U}, d \in \mathcal{D}} (\text{chosen}(u) \wedge \text{chosen}(d) \wedge L(u) = L(d)) \Rightarrow u = d \quad \text{where}$$

$$\text{chosen}(x) = x \in \vec{I} \vee x \in \vec{O} \vee \bigvee_{0 \le i < \theta(C)} (E_i \wedge (x \in \vec{I_{i,P_i}} \vee x \in \vec{O_{i,P_i}}))$$

5.3 Solving the Constraint

Putting the constraints together, we get our synthesis constraint:

$$\exists \mathcal{S}.\phi_{\text{wfp}}(\mathcal{S}) \land (\forall \vec{I}.\forall \vec{O}.\exists \mathcal{I}.\exists \mathcal{O}.\phi_{\text{conn}}(\mathcal{S}, \mathcal{D}, \mathcal{U}) \land \phi_{\text{lib}}(\mathcal{I}, \mathcal{O}) \land \phi_{\text{spec}}(\vec{I}, \vec{O}))$$

This constraint has nested quantifiers and cannot be solved efficiently with modern solvers. While Brahma effectively applies the CEGIS algorithm [43] with angelic variables to address the problem, it formulates the problem as a single, monolithic constraint system. The monolithic nature of this formulation makes it difficult to support composable features like sub-procedures, which can be used to implement techniques like context-aware window decomposition [37] and user-introduced pseudo-instructions. We address this limitation by reformulating the construction of the synthesis constraint as an angelic programming [9] problem, which is inherently composable.

5.3.1 Finite Synthesizer with Angelic Programming. To handle the nested quantifiers, we first skolemize the constraint. This standard logical transformation effectively replaces the existentially-quantified intermediates $\mathcal{I}$ and $\mathcal{O}$ with functions over the universally-quantified program inputs/outputs $\vec{I}$ and $\vec{O}$. For a finite set of I/O pairs (concrete instantiations of $\vec{I}$ and $\vec{O}$), we can further replace the universal quantifier with a conjunction:

$$\exists \mathcal{S}.\phi_{\text{wfp}}(\mathcal{S}) \land (\exists \mathcal{I}, \exists \mathcal{O}. \bigwedge_{\vec{I},\vec{O}} (\phi_{\text{conn}}(\mathcal{S}, \mathcal{D}(\vec{I},\vec{O}), \mathcal{U}(\vec{I},\vec{O})) \land \phi_{\text{lib}}(\mathcal{I}(\vec{I},\vec{O}), \mathcal{O}(\vec{I},\vec{O})) \land \phi_{\text{spec}}(\vec{I},\vec{O})))$$

For each I/O pair, we can now generate constraints as a relation ϕ over $\vec{I}$, $\vec{O}$, $\mathcal{S}$, and intermediate variables ($\mathcal{I}(\vec{I},\vec{O})$ and $\mathcal{O}(\vec{I},\vec{O})$). We view this as an angelic programming problem [9]. As shown in Fig. 6, ϕ can be transformed into an angelic interpreter that produces symbolic outputs for given inputs, generates *fresh* variables, and asserts the constraints. Crucially, each call to the interpreter generates a distinct set of fresh variables, ensuring that they are not interfering with each other. Here, the angelic programming and constraints can be managed as monadic effects in HieraSynth's host library Grisette [30], and the interpreter can be written as a conventional monadic interpreter and used anywhere an interpreter is expected.

```
interpret(input):
  output,intermediate = fresh_symbolic_variables()
  assert(phi(input,output,structural,intermediate))
  return output
```

Fig. 6. Transforming a constraint into an angelic interpreter with fresh variables and assertions

5.3.2 Composable Synthesis with Angelic Programming. The primary motivation for using angelic programming is to provide a clean and composable interface for super-optimization. This allows us to treat both the concrete program semantics (which is naturally defined as interpreters) and symbolic program space semantics (which become angelic interpreters) under a unified interpreter abstraction. The uniformity is crucial for seamlessly supporting features like sub-procedures. We can now offer a composable interface where both program specifications and instruction semantics are expressed as interpreters. The monad M encapsulates the effects, such as generating fresh angelic variables and asserting constraints:

$$\text{interpret}(\mathit{space}) \coloneqq \vec{I} \mapsto M(\vec{O}) \quad \text{interpret}(\mathit{inst}_{i,j}) \coloneqq \vec{I_{i,j}} \mapsto M(\vec{O_{i,j}})$$

This allows concrete programs and component-based program spaces to be mixed freely, as all are implemented as composable monadic interpreters. Fig. 7 illustrates two applications: context-aware window decomposition and user-defined pseudo-instructions.

In Fig. 7a, a program fragment under super-optimization can be part of a larger program. The code can leverage the assumptions about its calling context and be simplified accordingly—this is the context-aware window decomposition idea from Lens [37]. We will show an example of this

```
int32 programSpace(int32 a, int32 b) {
  reorder_and_wire { ... }
}
int32 program(int32 a, int32 b) {
  int32 add = a + b;
  int32 mul = a * b;
  return programSpace(add, mul);
}
```

(a) Program space as subroutine

```
int32 pseudoMulAdd(int32 a, int32 b) {
  return a + a * b;
}
int32 programSpace(int32 a, int32 b) {
  reorder_and_wire {
    2 * {..., pseudoMulAdd, ...}
  }
}
```

(b) Concrete pseudo-instruction as subroutine

Fig. 7. Subroutines enable composable super-optimization. (a) A prologue computes program space inputs; synthesis fills the callee. (b) A concrete pseudo-instruction (e.g., pseudoMulAdd) is called from the space.

in Section 7.6. Another application (Fig. 7b) is using concrete subroutines as pseudo-instructions. When users identify useful code patterns, they can provide them as subroutines, potentially reducing the k required to synthesize an optimization. We will discuss this further in Section 6.3.

6 Adapting HieraSynth to RISC-V Vector Code Super-Optimization

In this section, we describe how we adapt HieraSynth to build a RISC-V Vector (RVV) code super-optimizer. We implement HieraSynth as a reusable library with the Grisette [30] symbolic evaluation framework, enabling super-optimizer developers to obtain a program synthesizer simply by defining the instruction semantics with standard monadic Haskell code. The main challenge is adapting the RVV type system to our framework and making it work with our two-track approach to synthesis. As types in conventional SIMD ISAs like AVX or NEON are typically simpler, we believe that our approach could extend to these ISAs with simplifications. We will also describe how we use sub-procedures to further scale our synthesizer, and briefly outline the approach we use to over-approximate a program space from a reference implementation.

6.1 RISC-V Vectors

The RISC-V "V" extension introduces a flexible vector programming model with distinctive features:

- *Scalable vector length (VLEN)*: Hardware vendors can implement different vector register lengths (e.g., SiFive X280 [3] uses 512-bit registers, while P670 [4] is 128 bits). RVV code is usually (and is encouraged to) written in a portable, architecture-agnostic way.
- *Vector grouping*: Instructions can operate across multiple registers through the "vector group multiplier" (LMUL) parameter. For example, `vuint8m2_t` represents an 8-bit unsigned vector spanning two registers. RVV also allows vectors to occupy register fractions using fractional LMULs. Choosing different LMULs affects performance: on SiFive X280 with a 256-bit vector ALU but 512-bit VLEN, using LMUL=1/2 can halve execution time versus LMUL=1.
- *Selective operations*: Most RVV instructions support masking and configuring to process only the first `vl` elements. Elements outside the mask or exceeding `vl` can be configured to "agnostic" (undefined values) or "undisturbed" (preserved values) modes.

These features provide fine-grained control and facilitate optimizations. However, they also increase programming complexity for both developers and super-optimizers as there are numerous instruction and configuration choices to make.

6.2 RVV Types and Two-Track Synthesis

For two-track synthesis (described in Section 4.1), the apparent approach is using smaller VLEN for easy challengers. However, this is not always efficient since the minimum 128-bit VLEN specified by RVV remains challenging for solvers, especially with complex non-linear operations.

To make fast-track more effective, we further reduce bit-width to hypothetical machines with a modified RVV type system. Instead of using element width for vectors (the 8 in `vint8m1_t`) or bit-width for scalars, we represent bit-width as ratios between bit-width and XLEN configuration (scalar register width). For example, assume 64-bit machines, `vint8m1_t` will become `vint{1/8}m1_t`, while `uint64_t` will become `uint{1}`. This enables systematic machine configuration scaling with reasonable semantics, using the ratio as an invariant. Programs with these types become polymorphic across bit-width reductions: a program running with 32-bit data on a 64-bit machine with 128-bit vectors can also process 8-bit data on a hypothetical 16-bit machine with 32-bit vectors.

As our super-optimizer uses reference code as specification, scaling does not always work when magic numbers assume specific bit-widths. We mitigate this by providing multiple scaling models:

- Scale the reference program with reduced bit semantics.
- Only scale the inputs/outputs with zero/sign extension and truncation.
- Use only real 64-bit configurations (as a last resort).

The choice of scaling mode is currently left to users, and we use reduced bit semantics whenever possible. Automatic mode detection or parallel searching with multiple modes remains future work.

6.3 Sub-Procedures

As shown in Section 5.3.2, reformulating synthesis constraints as angelic programming enables sub-procedures. Our RVV synthesizer leverages this to help synthesize more complex programs.

As a general super-optimizer, we refrain from adding many pseudo-instructions, as the benefit is unclear. We include some scalar pseudo-instructions recommended by RISC-V specification and recognized by assemblers. We additionally add one type of vector pseudo-intrinsic: mask manipulation via regular vector instructions, which we found very common and general. One example of such pseudo-instruction is shown in Fig. 8. We believe that it is useful for users to develop their own domain-specific pseudo-instructions to further aid synthesis.

6.4 Program Space Inference

Though our system can handle large sets of choices, it is not beneficial to add all of them to the program space when doing the synthesis. We adopt simple heuristics to infer a program space from the reference programs: we collect all types and instructions present in the reference program and add all potentially relevant instructions to the choices.

```
vbool64_t vadd_mm(vbool64_t a, vbool64_t b) {
  size_t vl = __riscv_vsetvlmax_e8mf8();
  vuint8mf8_t av =
    __riscv_vlmul_trunc_u8mf8(__riscv_vreinterpret_u8m1(a));
  vuint8mf8_t bv =
    __riscv_vlmul_trunc_u8mf8(__riscv_vreinterpret_u8m1(b));
  vuint8mf8_t rv = __riscv_vadd(av, bv, vl);
  return __riscv_vreinterpret_b64(__riscv_vlmul_ext_u8m1(rv));
}
```

Fig. 8. Pseudo-intrinsic using reduced LMUL and vadd to shift masks. This is the only vector pseudo-intrinsic we include in our experiments.

For example, if multiple vector types are used, we introduce all possible conversion/narrowing/widening instructions. If we see multiplication, we add multiplication-related instructions. In contrast, we will not add multiplication to the choices when only linear arithmetic is present. Despite not using all instructions for an arbitrary program, we ensure all possibly useful instructions are added to the best of our knowledge. The resulting program space is still comprehensive and allows us to synthesize interesting optimizations, as shown in Section 7.

For the inferred instruction set, we use simple heuristics to group similar instructions together as this may give a better SMT encoding by enhancing term sharing. Finding a better methodology for grouping is left for future work.

7 Evaluation

We evaluate HieraSynth to understand the following research questions.

RQ1 How well does HieraSynth help scaling a base synthesizer?
RQ2 How effectively can HieraSynth synthesize and prove optimality for vector programs?
RQ3 Can HieraSynth overcome the fundamental trade-off between output program size k and instruction set size n?
RQ4 How effective is HieraSynth's two-track synthesis approach?
RQ5 Can HieraSynth discover optimizations beyond those identified by human experts?
RQ6 How efficiently does HieraSynth scale with increasing parallelism?

7.1 Benchmarks

We evaluate HieraSynth on 25 scalar and 26 vector benchmarks. Statistics are in Table 1.

The scalar benchmarks come from Brahma [24], with 25 bit-manipulation programs from Hacker's delight [48] rewritten using RISC-V instructions. For these benchmarks, HieraSynth synthesizes using a subset of RISC-V basic ISA with Zicond (integer conditional) and Zbb (basic bit-manipulation) extensions. We include four register-immediate instructions (`andi`, `slli`, `srai`, and `srli`), and one component returning an immediate (equivalent to the `li` pseudo-instruction). Immediates are treated as angelic values [9] to be synthesized by the solvers. We extend the program space for p20 with division and p25 with multiplication instructions from the M extension. We use a simple cost model where most instructions have a cost of 1, with higher costs assigned to multiplication and division instructions. The initial cost is set to the reference cost plus one to ensure the reference program is included in our synthesis space.

Our first set of vector benchmarks include operators from Highway [16], a C++ library providing portable SIMD/vector intrinsics optimized for various platforms, including RVV. While Highway operators aim to achieve performance within 10–20% of hand-coded assembly, we identified 12 operators for further optimization with HieraSynth. Section 7.6 explains the `Lt128` operator, and Highway's documentation [1] covers other operator semantics. Highway operators are polymorphic to different RISC-V vector register groupings (LMUL) and vector lengths (VL). We instantiate operators with LMUL=1 and full vectors and compile them to LLVM IR with `clang -O3`. The `ZeroExtendResizeBitCast` operator is benchmarked with both LMUL=1 to LMUL=2 extension and LMUL=1 to LMUL=1/2 truncation. For most benchmarks, the program space is fully inferred from the source program (Section 6.4). For `Lt128` and `Min128`, we include two benchmarks each, one using the inferred program space for the entire operator and the other incorporating a prologue. Our cost model for vector synthesis is derived from the LLVM cost model for SiFive X280. We verify correctness up to 512-bit vectors for all benchmarks. In addition, we gathered 11 kernels from the vqsort (vectorized Quicksort) algorithm [8]. Some kernels do not cover all 32/64/128-bit values either because they are not supported or are proven optimal with the inferred program space.

All experiments were run on three Rocky Linux 9.4 servers, each with two Intel Xeon Platinum 8160M CPUs (48 cores/96 threads total) and 768 GB RAM. We use Bitwuzla 0.7.0 SMT solver [35] with a 3600 s timeout for benchmarks, 500 s for fast-track tasks, and 1500 s for slow-track tasks. We report mean runtime across three runs per benchmark.

7.2 RQ1: Scaling a Base Synthesizer

We evaluate whether hierarchical program-space decomposition and parallelism help a base synthesizer scale. We instantiate the base synthesizer as Brahma in all our experiments.

Vanilla Brahma does not automatically explore subspaces; instead, it relies on human insights to augment a *standard library* to reduce the effective ISA size. Following Brahma [24], we built a

Table 1. Benchmark statistics. Program space metrics: number of components (k), choices per component (n), theoretical number of leaf subspaces ($C(n+k-1,k)$) if fully partitioned (Leaf), and number of well-typed programs in the program space (Prog). Performance metrics: instruction count (Inst) and cost (Cost) for reference (R) and best synthesized (B) programs. N/A indicates unknown metrics as HieraSynth timed out.

Name	Space				Inst		Cost		Name	Space				Inst		Cost	
	k	n	Leaf	Prog	R	B	R	B		k	n	Leaf	Prog	R	B	R	B
ZeroExtendResizeBitCast	1	726	7.3e02	2.0e00	12	1	5	0	p01	2	31	5.0e02	9.3e03	3	2	2	2
ConcatLowerLower	2	65	2.1e03	3.7e02	5	2	4	2	p02	2	31	5.0e02	9.3e03	3	2	2	2
ConcatUpperLower	2	65	2.1e03	3.7e02	6	2	8	2	p03	2	31	5.0e02	9.3e03	2	2	2	2
ConcatUpperUpper	2	65	2.1e03	3.7e02	7	2	12	2	p04	2	31	5.0e02	9.3e03	3	2	2	2
ZeroExtendResizeBitCast.Ext	3	473	1.8e07	2.2e03	11	3	22	6	p05	2	31	5.0e02	9.3e03	3	2	2	2
ConcatLowerUpper	4	65	8.1e05	4.2e06	6	4	8	4	p06	2	31	5.0e02	9.3e03	3	2	2	2
AddSub	4	594	5.2e09	6.2e08	11	4	10	5	p07	2	31	5.0e02	9.3e03	4	2	3	2
InsertLane	5	283	1.6e10	5.0e10	10	5	13	6	p08	2	31	5.0e02	9.3e03	4	2	3	2
OddEvenBlocks	5	365	5.5e10	3.8e12	9	5	16	8	p09	2	31	5.0e02	9.3e03	4	2	3	2
Lt128.WithPrologue	6	118	4.2e09	2.1e15	18	6	28	13	p10	4	31	4.6e04	2.3e10	5	4	4	4
MulEven	6	450	1.2e13	2.3e12	13	6	24	19	p11	2	31	5.0e02	8.1e04	3	2	3	2
MulOdd	6	450	1.2e13	2.3e12	13	6	24	19	p12	3	31	5.5e03	3.5e07	5	3	4	3
Min128.WithPrologue	7	116	6.7e10	7.3e23	14	7	22	15	p13	2	31	5.0e02	9.3e03	5	2	4	1
Lt128	7	214	4.5e12	6.3e17	18	7	28	11	p14	4	31	4.6e04	2.3e10	5	4	4	4
Min128	8	212	1.2e14	1.6e22	14	8	22	13	p15	4	31	4.6e04	2.3e10	5	4	4	4
PrevValue_32	2	40	8.2e02	4.4e01	4	2	4	2	p16	1	31	3.1e01	3.0e02	7	1	6	1
PrevValue_64	2	40	8.2e02	4.4e01	4	2	4	2	p17	4	31	4.6e04	1.1e09	5	4	4	4
PrevValue_128	5	286	1.7e10	4.4e10	16	5	26	6	p18	3	31	5.5e03	2.5e06	7	3	6	3
SortPairsDistance4_64	6	77	3.5e08	8.7e10	10	6	12	8	p19	6	31	1.9e06	3.4e17	6	6	6	6
SortPairsDistance1_64	6	212	1.4e11	8.1e12	12	6	18	9	p20	6	32	2.3e06	7.9e14	7	6	39	39
SwapAdjacentPairs_32	6	231	2.3e11	7.9e12	10	6	12	10	p21	7	31	1.0e07	6.6e21	13	7	12	7
SwapAdjacentPairs_64	6	356	2.9e12	2.9e14	11	6	20	9	p22	3	31	5.5e03	2.5e06	12	3	10	2
SwapAdjacentQuads_32	6	714	1.9e14	2.9e14	13	6	20	10	p23	1	31	3.1e01	6.3e01	24	1	15	1
SortPairsDistance1_32	7	514	2.0e15	2.2e17	18	7	28	20	p24	5	31	3.2e05	7.0e11	17	5	12	4
SortPairsDistance1_128	N/A	284	N/A	N/A	26	N/A	50	N/A	p25	1	35	3.5e01	3.5e02	18	1	24	3
SortPairsDistance4_32	N/A	542	N/A	N/A	21	N/A	26	N/A									

standard library of 12 commonly used RISC-V instructions. This fixed set is insufficient for optimal results in 14 of the 25 scalar benchmarks, requiring additional components identified via *human insight*. Minimizing such human input can severely limit synthesizable program size, as observed with Souper [40]. To provide a fair upper bound for the base synthesizer, we simulate Brahma's *best-case* performance by selectively adding exactly the components needed to synthesize the optimal programs identified by HieraSynth. With this oracle-like advantage, Brahma's effective ISA is much smaller than HieraSynth's full 31–35 instruction choice space.

Unlike the original Brahma evaluation (which reported time to the first valid solution), Fig. 9 reports: (1) time to the best (optimal) solution and (2) time to prove optimality within the declared program space. With 72 cores, HieraSynth consistently outperforms the base synthesizer, achieving 1.20–106.73× speedups in finding the best solution and 3.16–17.03× speedups in proving optimality.

HieraSynth also outperforms Brahma sequentially (derivable from Fig. 9 with the speedup baselines in Fig. 14 and 15). This is not due to parallelism but due to changes in the search methodology by decomposing on n: (1) early proofs of unrealizability prune entire partitions, and (2) restricting available operators reduces the size of each SMT query.

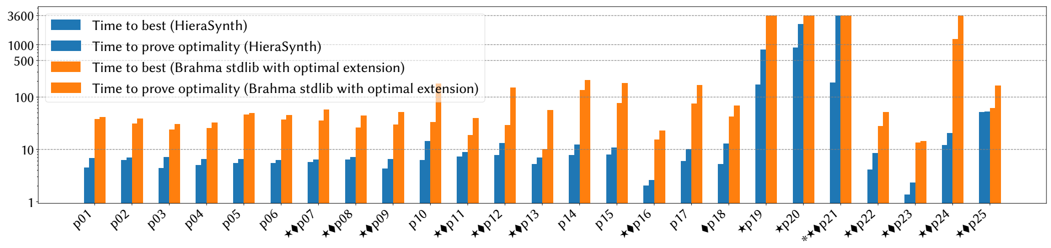


Fig. 9. Comparison of HieraSynth with the base synthesizer. Bars show synthesis time (seconds) across benchmarks (log scale; 3600 s timeout). ♦: HieraSynth found cheaper solutions than the reference. ∗: solution found but not proven optimal within 3600 s on 72 cores. ★: Hacker's delight benchmarks that require extra components beyond Brahma's standard library to achieve the best solution.

7.3 RQ2: Scaling to Vector Synthesis

The scalar benchmarks use a relatively small instruction set and cannot fully demonstrate HieraSynth's scalability to large vector ISAs. In Fig. 10, we evaluate HieraSynth on synthesizing vector programs. We do not compare with Brahma for these vector benchmarks, as constructing a standard library covering the diverse range of vector instructions needed is impractical. The sheer number of vector instructions and their complex semantics make manual library construction or instruction selection extremely challenging for human experts.

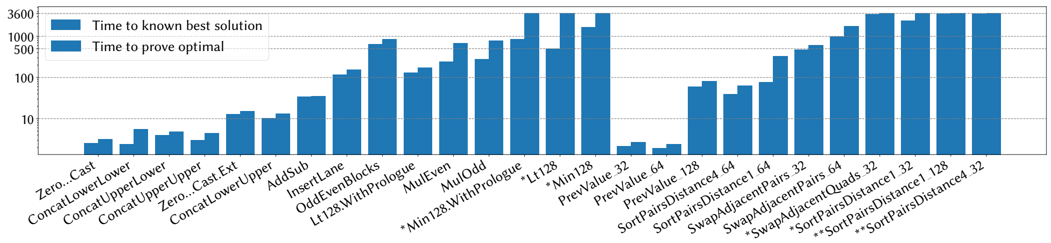


Fig. 10. Synthesis time (seconds, log scale) with HieraSynth for vector benchmarks. Each run uses 72 cores with a 3600 s timeout. The Highway and vqsort benchmarks are sorted by k then n. *: HieraSynth cannot prove optimality within the timeout. **: HieraSynth cannot synthesize a solution.

These benchmarks show the practical limits. Among the 26 benchmarks from Highway and vqsort, our approach successfully synthesizes optimized programs for 25 benchmarks and proves optimality for 19 of them. The synthesis time generally correlates with program complexity. These results demonstrate that our tool can handle complex vector kernels with up to 8 output instructions, demonstrating a significant advance in super-optimization for modern vector ISAs.

For the cases where synthesis failed or optimality could not be proven within the time limit, we analyzed the exploration patterns by tracking the number of nodes explored at each depth. Fig. 11 shows these patterns for four representative benchmarks: two successful cases (`AddSub` and `MulOdd`) where optimality was proven, one partially successful case (`Min128`) where a solution was found but optimality was not proven, and one failure case (`SortPairsDistance1_128`). The presence of pending nodes at deeper depths (shown by dotted lines in Fig. 11) for complex cases indicates that the search space continues to grow beyond our computational resources.

Successful cases show node exploration peaking at moderate depths, indicating effective pruning. However, benchmarks like Min128 and SortPairsDistance1_128, with $n \approx 200$–300 and $k \geq 8$, experience explosive growth, exceeding search capacity. Future heuristics to detect unrealizable subspaces more efficiently could help address this. Although more parallelism might also help, we leave further scaling to future work. Despite these challenges, HieraSynth successfully handles many complex benchmarks, outperforming prior super-optimizers by a huge margin.

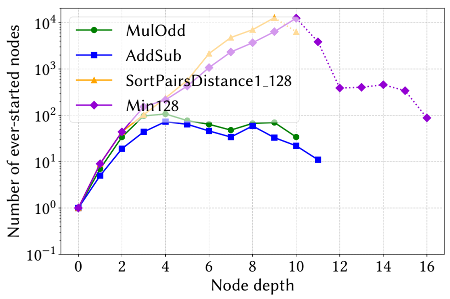


Fig. 11. Nodes ever started at each depth. AddSub and MulOdd finished their optimality proofs. Min128 found a solution but was not proven optimal. SortPairsDistance1_128 failed to synthesize a solution. Min128 and SortPairsDistance1_128 ran for two days on 72 cores to collect these statistics. Dotted segments mark depths with pending nodes; counts there are incomplete.

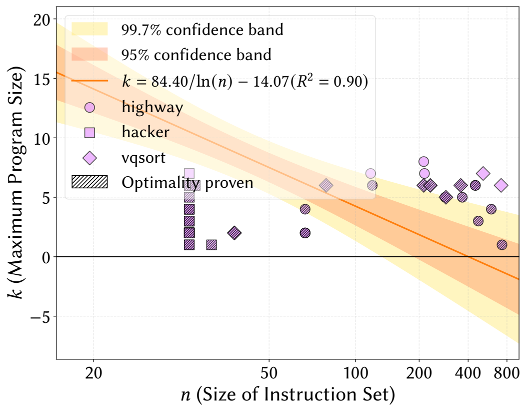


Fig. 12. Trade-off between k and n. The line fits the *maximum* reported k at each n for existing tools (an empirical capability frontier, might not be jointly achievable). For HieraSynth, we plot the *actual* k and n per benchmark. Points below the line represent easier benchmarks.

7.4 RQ3: The k-vs-n Trade-off

Fig. 12 quantifies HieraSynth's impact on the k-vs-n trade-off. Prior complete techniques are constrained by a practical capability frontier ($k = 84.40/\ln(n) - 14.07$, $R^2 = 0.90$), while HieraSynth successfully synthesizes solutions across diverse benchmarks, significantly extending beyond this capability frontier, with up to 10.66× larger n for a given k, or up to 4.75× larger k for a fixed n. This demonstrates our approach can handle programs with larger k and n than previously feasible. Crucially, for many of these challenging benchmarks, we formally prove the solution's optimality in our program space (hatched markers). This enables HieraSynth to address complex, real-world synthesis problems like vector programming that were previously intractable for complete super-optimizers.

7.5 RQ4: Effectiveness of Two-Track Synthesis

To answer RQ4, we evaluate the effectiveness of our two-track synthesis approach, with results detailed in Table 2. The data show the effectiveness of this strategy. For most benchmarks, the final solution is found without resorting to the more resource-intensive slow-track synthesis, as evidenced by the "Slow (%)" column being 0.0. This indicates that the fast-track and generalization phases successfully solved the problem with less computation. Note that a generalization success rate below 100% does not imply that the fast-track success was spurious. Instead, it often implies that the task was pruned during generalization after a solution is found in a different subspace and the new cost bound has been synchronized.

Table 2. Two-Track Synthesis Success Rate. #Easy shows the number of subspaces that passed the easy challengers. Slow (%) and Generalization (%) show the percentage of those subspaces that led to a solution via the slow-track and the generalization step, respectively. Most tasks do not rely on the slow-track. p23 and p25 use a single track on real 64-bit machine configuration. SortPairsDistance1_128 and SortPairsDistance4_32 had no subspaces pass the easy challengers. These benchmarks are excluded.

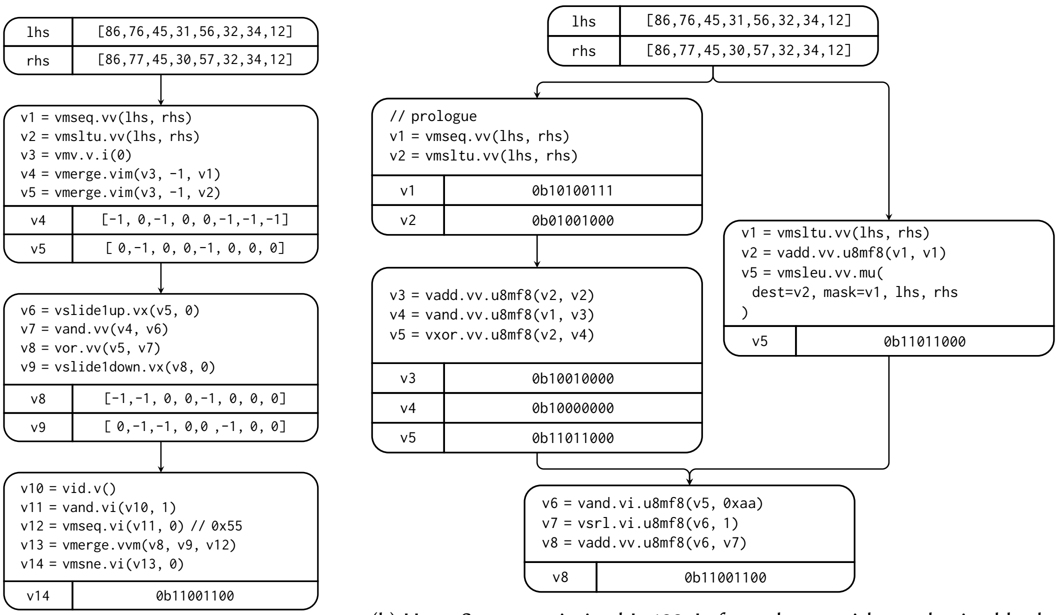
Name	Stats			Name	Stats		
	#Easy	Slow (%)	Generalization (%)		#Easy	Slow (%)	Generalization (%)
ZeroExtendResizeBitCast	5	0.0	64.7	p01	4	0.0	25.0
ConcatLowerLower	1	0.0	100.0	p02	5	0.0	62.5
ConcatUpperLower	4	0.0	83.3	p03	3	0.0	30.0
ConcatUpperUpper	2	0.0	37.5	p04	5	0.0	40.0
ZeroExtendResizeBitCast.Ext	3	0.0	36.4	p05	3	0.0	30.0
ConcatLowerUpper	2	0.0	66.7	p06	5	0.0	18.8
AddSub	1	0.0	60.0	p07	5	0.0	20.0
InsertLane	3	0.0	60.0	p08	5	0.0	20.0
OddEvenBlocks	111	0.9	0.0	p09	4	0.0	25.0
Lt128.WithPrologue	114	0.9	0.0	p10	4	0.0	41.7
MulEven	1	0.0	100.0	p11	2	0.0	50.0
MulOdd	1	0.0	100.0	p12	1	0.0	100.0
Min128.WithPrologue	43	0.8	3.9	p13	5	0.0	94.1
Lt128	21	4.8	1.6	p14	1	0.0	75.0
Min128	49	3.4	1.3	p15	1	0.0	100.0
PrevValue_32	4	0.0	58.3	p16	6	0.0	33.3
PrevValue_64	4	0.0	66.7	p17	1	0.0	100.0
PrevValue_128	14	9.1	0.0	p18	3	0.0	33.3
SortPairsDistance4_64	1	0.0	80.0	p19	1	0.0	60.0
SortPairsDistance1_64	2	0.0	85.7	p20	1	0.0	100.0
SwapAdjacentPairs_32	1	0.0	100.0	p21	1	0.0	100.0
SwapAdjacentPairs_64	2	0.0	83.3	p22	3	0.0	40.0
SwapAdjacentQuads_32	1	0.0	100.0	p24	2	0.0	57.1
SortPairsDistance1_32	1	0.0	100.0				

7.6 RQ5 & Case Study: Synthesis of Lt128 Operator in Highway

To assess HIERASYNTH's capability to discover non-trivial optimizations, we examine its synthesis of the `Lt128` operator from Highway. This operator compares 128-bit integers formed by concatenating adjacent 64-bit elements from two input vectors, returning a mask indicating whether each 128-bit integer in the first vector is less than the corresponding one in the second. Each comparison contributes two (duplicated) bits in the resulting mask. See Fig. 13a for an illustration.

The original `Lt128` implementation follows a step-by-step approach: (1) 64-bit element-wise less than and equal to comparisons, (2) assembly of intermediate results, (3) duplication of results for adjacent bits. This approach requires mask manipulation operations not directly provided by RVV (like shifting and ternary operations). The conventional trick involves converting masks to vectors (with each mask bit corresponding to one element), applying vector instructions, then converting back to masks. This is a recognized technique that is known to be inefficient [2]. In the original implementation (Fig. 13a), `v4` and `v5` are such converted masks.

This implementation is suboptimal, particularly on X280. As masks occupy only a fraction of vector registers, using LMUL=1 instructions on converted masks takes twice the time of mask operations or vector operations with fractional LMUL. We attempted to optimize this program with an early version of our tool, which was not scalable enough to synthesize the operator as a whole. However, it seemed that the first two comparisons must be present in any optimized solution. Under this assumption, we successfully synthesized the left path in Fig. 13b using the comparisons



```
lhs  [86,76,45,31,56,32,34,12]
rhs  [86,77,45,30,57,32,34,12]

v1 = vmseq.vv(lhs, rhs)
v2 = vmsltu.vv(lhs, rhs)
v3 = vmv.v.i(0)
v4 = vmerge.vim(v3, -1, v1)
v5 = vmerge.vim(v3, -1, v2)
v4   [-1, 0,-1, 0, 0,-1,-1,-1]
v5   [ 0,-1, 0, 0,-1, 0, 0, 0]

v6 = vslide1up.vx(v5, 0)
v7 = vand.vv(v4, v6)
v8 = vor.vv(v5, v7)
v9 = vslide1down.vx(v8, 0)
v8   [-1,-1, 0, 0,-1, 0, 0, 0]
v9   [ 0,-1,-1, 0,0 ,-1, 0, 0]

v10 = vid.v()
v11 = vand.vi(v10, 1)
v12 = vmseq.vi(v11, 0) // 0x55
v13 = vmerge.vvm(v8, v9, v12)
v14 = vmsne.vi(v13, 0)
v14  0b11001100
```

(a) Highway baseline for Lt128

```
lhs  [86,76,45,31,56,32,34,12]
rhs  [86,77,45,30,57,32,34,12]

// prologue
v1 = vmseq.vv(lhs, rhs)
v2 = vmsltu.vv(lhs, rhs)
v1   0b10100111
v2   0b01001000

v3 = vadd.vv.u8mf8(v2, v2)
v4 = vand.vv.u8mf8(v1, v3)
v5 = vxor.vv.u8mf8(v2, v4)
v3   0b10010000
v4   0b10000000
v5   0b11011000

v1 = vmsltu.vv(lhs, rhs)
v2 = vadd.vv.u8mf8(v1, v1)
v5 = vmsleu.vv.mu(
  dest=v2, mask=v1, lhs, rhs
)
v5   0b11011000

v6 = vand.vi.u8mf8(v5, 0xaa)
v7 = vsrl.vi.u8mf8(v6, 1)
v8 = vadd.vv.u8mf8(v6, v7)
v8   0b11001100
```

(b) HieraSynth-optimized Lt128. Left: prologue with synthesized body. Right: end-to-end synthesis. End-to-end version fuses instructions.

Fig. 13. Lt128 case study. We assume VLEN=512 and LMUL=1, thus each vector has 512 bits (8 × 64-bit values). The first lane is shown as the rightmost value in vectors. Bitcasts are omitted for brevity.

as a prologue. The program bitcasts masks to LMUL=1/8 vectors and uses corresponding vector instructions, reducing the estimated cycle count by >50%. For Lt128 instantiated with LMUL=8, similar optimization yields approximately 4× speedup. The significant improvement led us to believe that this version was optimal, and we submitted a patch that was merged by the Highway developers.

With later improvements to HieraSynth, we attempted to synthesize `Lt128` without a prologue. Surprisingly, HieraSynth discovered an even more optimized implementation (right path in Fig. 13b) leveraging masked undisturbed intrinsics to fuse the first two steps. While this approach could be synthesized using a peephole optimization on the left path, no previous system could synthesize that left-path transformation end-to-end; it is beyond peephole optimization's capability.

This case highlights HieraSynth's ability to surpass expert optimizations by exploring innovative paths without the biases that experts' familiarity with established solutions can introduce.

7.7 RQ6: Parallel Speedup

To evaluate HieraSynth's parallel scalability, Fig. 14 and Fig. 15 present the speedup with different numbers of cores for synthesizing the best solution and proving optimality. In each figure, we omit benchmarks that finished in 10 seconds on a single core, as short tasks have limited potential for parallel acceleration. The benchmarks are arranged in ascending order of single-core time to highlight the scaling trend from less to more computationally intensive tasks.

In Fig. 14, several benchmarks, such as `PrevValue_128` and `p21`, achieve nearly linear scaling up to 72 cores. The system typically achieves more than 24× speedup for other complex tasks with 48-72 cores. This indicates that computationally intensive tasks benefit most from parallelism. The limitations to this scaling stem from the fact that, as the number of cores increases, more cores may

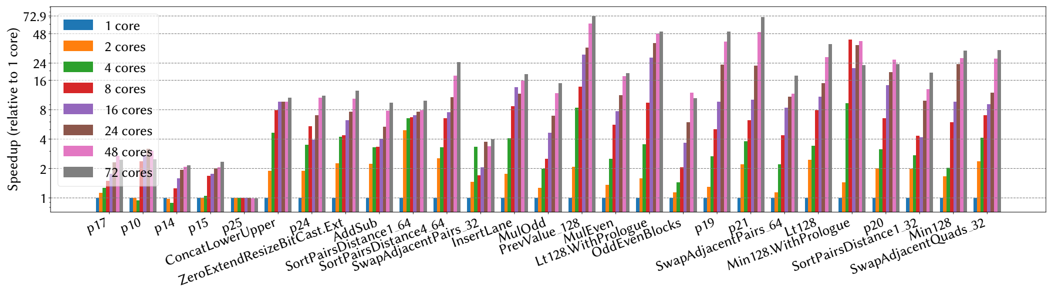


Fig. 14. Parallel speedup for the time to the best solution, relative to a single core.

work on tasks that are ultimately pruned, resulting in wasted work. A more advanced scheduling strategy to better prioritize tasks could mitigate this issue by reducing redundant computation. Developing such strategies remains future work.

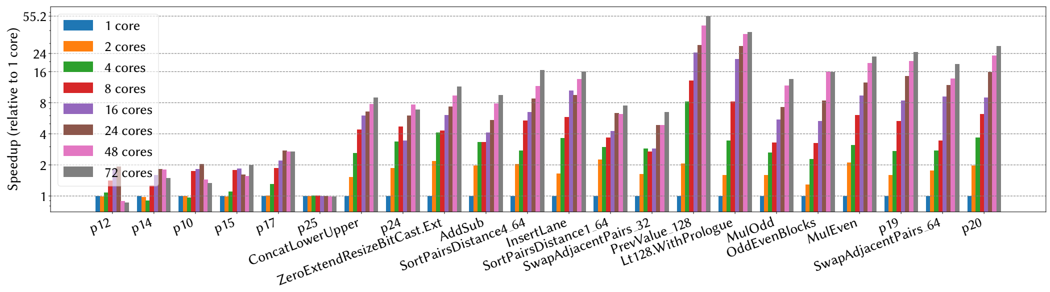


Fig. 15. Parallel speedup for proving optimality, relative to a single core.

In Fig. 15, we observe a similar trend for proving optimality as in Fig. 14, although the overall speedup is typically smaller. This mostly comes from a long-tail in the final stage of parallel synthesis. At this stage, most tasks have been pruned, leaving a small number of remaining tasks. When the number of remaining tasks is fewer than the available cores, some cores remain idle. This reduces effective parallelization and diminishes the speedup.

Overall, although scalability can be limited by the long tail effect in the optimality-proving phase and redundant work on pruned tasks, HieraSynth demonstrates effective parallel scalability, especially for complex tasks requiring extensive search space exploration.

8 Related Work

8.1 Super-Optimizers

Super-optimization, pioneered by Massalin [32], exhaustively searches for shortest instruction sequences but with significant restriction on ISA size. Later work improved scalability through techniques like exploiting symmetries (Bansal's system [6]) and bidirectional search (Lens [37]), often combined with peephole-style optimizations. Another class of techniques uses e-graphs [34, 36] and equality saturation [44, 45, 50] as seen in Denali [25, 26], Tensat [51] and Diospyros [47]. However, these approaches rely on expert-provided rewrite rules, limiting them to known optimization patterns. Solver-based tools like Sketch [42, 43] and Brahma [15, 24], Souper [40], and Minotaur [29] leverage SMT solvers to automate the search. These systems, however, either require users to provide insights or face scalability limits. In contrast, incomplete approaches, such as STOKE [41], use stochastic search, sacrificing the ability to guarantee optimality.

8.2 Parallelism in Program Synthesis

Parallelism is widely used to accelerate program synthesis. Morpheus [14] explores different program sizes in parallel to find a first solution faster. Adaptive concretization [22, 23] uses statistical trials to guide parallel exploration of instantiation of symbolic bits. Synapse [10] explores parameterized sketches in parallel. These systems explore different portions of the program space in parallel but do not systematically partition the space. Some specialized parallelization techniques are also used in some fields. Paresy [46] applies a data-parallel technique on GPUs to regular expression synthesis, and Synthphonia [12] orchestrates asynchronous cooperation between a deducer and enumerators to synthesize string transformations. The most analogous work, PASSES [17], decomposes the synthesis problem based on the instruction selection set. However, its partitioning is static, while its optional adaptive mode sacrifices completeness (and hence optimality). In contrast, HieraSynth's hierarchical decomposition is both adaptive and complete. HieraSynth creates "goldilocks" subspaces and scalable super-optimization without sacrificing optimality.

8.3 Combining Symbolic and Explicit Search

Synthesis approaches often combine symbolic and enumerative search but differ in their decomposition strategies. Adaptive concretization [22, 23] uses random sampling to selectively concretize symbolic bits but ignores high-level program space structure, while Synapse [10] uses a gradient-guided search over sketches that can be bottlenecked by the single hardest case, limiting parallel speedup. Other techniques structurally partition the problem. Synquid [38] synthesizes recursive functional programs by enumerating top parts of the program and synthesizing remaining parts with liquid abduction. DryadSynth [21] also uses divide-and-conquer as HieraSynth does, but partitions the program specification, while HieraSynth partitions the program space.

8.4 Optimal Synthesis

Many program synthesizers consider optimality as a criterion. Chaudhuri et al. [11] propose a smoothed proof search technique for optimizing a program with parameter holes with an optimality measure. The smoothed proof search approach reduces optimal synthesis to optimization problems that are solved numerically. The Symba approach [28] optimizes objective functions in the theory of linear real arithmetic with SMT solvers. It maintains under- and over-approximation of the optimal solution and gradually converges to the solution. Synapse [10] proposes a framework for optimal synthesis with a set of sketches with increasing difficulty and cost bounds. Sketches may be pruned if they cannot contain a more optimized program. There are also works doing implicit optimal synthesis by selecting preferable solutions in the program space, as in FlashMeta [39].

9 Conclusion

We present HieraSynth, a parallel super-optimization framework that uses hierarchical decomposition on instruction selection to overcome the trade-off between output program size and instruction set size that limits existing systems. HieraSynth efficiently synthesizes complex vector programs, outperforms prior methods, surpasses expert-designed optimizations, and achieves near-linear parallel scalability while preserving completeness. This makes super-optimization practical for modern vector architectures like RVV.

10 Data-Availability Statement

The artifact of HieraSynth consists of two parts.

(1) The HieraSynth library: a generic Haskell library that implements the algorithms, built on top of the Grisette [30] symbolic evaluation engine.
(2) An RVV synthesizer implemented using HieraSynth. This synthesizer supports a subset of the RISC-V ISA, including the vector extension and additional extensions, integrated with our type system described in Section 6.2. It also includes a converter that translates LLVM IR into programs represented in our format and types.

All figures and tables in the benchmark section are generated using scripts that process raw data produced by the artifact, with the exception of Fig. 11 and 12. For Fig. 11, we executed the benchmarks once, manually collected the data, and hardcoded it into our script. For Fig. 12, the statistics for related tools are hardcoded in the script, and some estimated values may be subject to inaccuracies. The artifact has been archived [31].

Acknowledgments

This work has been supported in part by the NSF awards ITE−2132318 CCF−2122950 ITE−2029457 ITE−1936731 CCF−1918027 IIS−1924435 the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA NSF CCF−1723352), the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA CMU 1042741−394324 AM01, a grant DARPA FA8750−16−2−0032, as well as gifts from Adobe, Facebook, Google, Intel, and Qualcomm.

References

[1] 2024. *Highway API synopsis/quick reference.* https://web.archive.org/web/20240615094120/https://github.com/google/highway/blob/master/g3doc/quick_reference.md

[2] 2024. *RISC-V vector extension for integer workloads: An informal gap analysis.* https://web.archive.org/web/20241113090818/https://gist.github.com/camel-cdr/99a41367d6529f390d25e36ca3e4b626

[3] 2025. *SiFive Intelligence X200 Series.* https://web.archive.org/web/20250629211746/https://www.sifive.com/cores/intelligence-x200-series

[4] 2025. *SiFive Performance P600-Series.* https://web.archive.org/web/20250629195319/https://www.sifive.com/cores/performance-p600

[5] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design.* 1–8. doi:10.1109/FMCAD.2013.6679385

[6] Sorav Bansal and Alex Aiken. 2006. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII).* Association for Computing Machinery, New York, NY, USA, 394–403. doi:10.1145/1168857.1168906

[7] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, Cesar Kunz, and Mark Marron. 2013. From relational verification to SIMD loop synthesis. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13).* Association for Computing Machinery, New York, NY, USA, 123–134. doi:10.1145/2442516.2442529

[8] Mark Blacher, Joachim Giesen, Peter Sanders, and Jan Wassenberg. 2022. Vectorized and performance-portable Quicksort. arXiv:2205.05982 [cs.IR] https://arxiv.org/abs/2205.05982

[9] Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. 2010. Programming with angelic nondeterminism. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10).* Association for Computing Machinery, New York, NY, USA, 339–352. doi:10.1145/1706299.1706339

[10] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing synthesis with metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16).* Association for Computing Machinery, New York, NY, USA, 775–788. doi:10.1145/2837614.2837666

[11] Swarat Chaudhuri, Martin Clochard, and Armando Solar-Lezama. 2014. Bridging boolean and quantitative synthesis using smoothed proof search. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14).* Association for Computing Machinery, New York, NY, USA, 207–220. doi:10.1145/2535838.2535859

[12] Yuantian Ding and Xiaokang Qiu. 2025. A Concurrent Approach to String Transformation Synthesis. *Proc. ACM Program. Lang.* 9, PLDI, Article 233 (June 2025), 25 pages. doi:10.1145/3729336

[13] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2021. DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 835–850. doi:10.1145/3453483.3454080

[14] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 422–436. doi:10.1145/3062341.3062351

[15] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 62–73. doi:10.1145/1993498.1993506

[16] Highway Developers. 2024. *Highway*. https://github.com/google/highway

[17] Jingmei Hu, Stephen Chong, and Margo Seltzer. 2024. Parallel Assembly Synthesis. In *Logic-Based Program Synthesis and Transformation*, Juliana Bowles and Harald Søndergaard (Eds.). Springer Nature Switzerland, Cham, 3–26.

[18] Jingmei Hu, Eric Lu, David A. Holland, Ming Kawaguchi, Stephen Chong, and Margo Seltzer. 2023. Towards Porting Operating Systems with Program Synthesis. *ACM Trans. Program. Lang. Syst.* 45, 1, Article 2 (March 2023), 70 pages. doi:10.1145/3563943

[19] Jingmei Hu, Priyan Vaithilingam, Stephen Chong, Margo Seltzer, and Elena L. Glassman. 2021. Assuage: Assembly Synthesis Using A Guided Exploration. In *The 34th Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '21)*. Association for Computing Machinery, New York, NY, USA, 134–148. doi:10.1145/3472749.3474740

[20] Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas Reps. 2020. Exact and approximate methods for proving unrealizability of syntax-guided synthesis problems. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1128–1142. doi:10.1145/3385412.3385979

[21] Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. 2020. Reconciling enumerative and deductive program synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1159–1174. doi:10.1145/3385412.3386027

[22] Jinseong Jeon, Xiaokang Qiu, Armando Solar-Lezama, and Jeffrey S. Foster. 2015. Adaptive Concretization for Parallel Program Synthesis. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 377–394.

[23] Jinseong Jeon, Xiaokang Qiu, Armando Solar-Lezama, and Jeffrey S Foster. 2017. An empirical study of adaptive concretization for parallel program synthesis. *Formal Methods in System Design* 50 (2017), 75–95.

[24] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 215–224. doi:10.1145/1806799.1806833

[25] Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: a goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 304–314. doi:10.1145/512529.512566

[26] Rajeev Joshi, Greg Nelson, and Yunhong Zhou. 2006. Denali: A practical algorithm for generating optimal code. *ACM Trans. Program. Lang. Syst.* 28, 6 (Nov. 2006), 967–989. doi:10.1145/1186632.1186633

[27] Jinwoo Kim, Loris D'Antoni, and Thomas Reps. 2023. Unrealizability Logic. *Proc. ACM Program. Lang.* 7, POPL, Article 23 (Jan. 2023), 30 pages. doi:10.1145/3571216

[28] Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. 2014. Symbolic optimization with SMT solvers. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 607–618. doi:10.1145/2535838.2535857

[29] Zhengyang Liu, Stefan Mada, and John Regehr. 2024. Minotaur: A SIMD-Oriented Synthesizing Superoptimizer. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 326 (Oct. 2024), 25 pages. doi:10.1145/3689766

[30] Sirui Lu and Rastislav Bodík. 2023. Grisette: Symbolic Compilation as a Functional Programming Library. *Proc. ACM Program. Lang.* 7, POPL, Article 16 (Jan. 2023), 33 pages. doi:10.1145/3571209

[31] Sirui Lu and Rastislav Bodík. 2025. OOPSLA 2025 Artifact: HieraSynth. Zenodo. doi:10.5281/zenodo.16923496

[32] Henry Massalin. 1987. Superoptimizer: a look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems* (Palo Alto, California, USA) *(ASPLOS II)*. Association for Computing Machinery, New York, NY, USA, 122–126. doi:10.1145/36206.36194

[33] W. M. McKeeman. 1965. Peephole optimization. *Commun. ACM* 8, 7 (July 1965), 443–444. doi:10.1145/364995.365000

[34] Greg Nelson and Derek C. Oppen. 1980. Fast Decision Procedures Based on Congruence Closure. *J. ACM* 27, 2 (April 1980), 356–364. doi:10.1145/322186.322198

[35] Aina Niemetz and Mathias Preiner. 2023. Bitwuzla. In *Computer Aided Verification*, Constantin Enea and Akash Lal (Eds.). Springer Nature Switzerland, Cham, 3–17.

[36] Robert Nieuwenhuis and Albert Oliveras. 2005. Proof-Producing Congruence Closure. In *Term Rewriting and Applications*, Jürgen Giesl (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 453–468.

[37] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling up Superoptimization. *SIGARCH Comput. Archit. News* 44, 2 (March 2016), 297–310. doi:10.1145/2980024.2872387

[38] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 522–538. doi:10.1145/2908080.2908093

[39] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 107–126. doi:10.1145/2814270.2814310

[40] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. 2018. Souper: A Synthesizing Superoptimizer. arXiv:1711.04422 [cs.PL] https://arxiv.org/abs/1711.04422

[41] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 305–316. doi:10.1145/2451116.2451150

[42] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph. D. Dissertation. University of California, Berkeley, USA. Advisor(s) Bodik, Rastislav. AAI3353225.

[43] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 404–415. doi:10.1145/1168857.1168907

[44] Michael Stepp, Ross Tate, and Sorin Lerner. 2011. Equality-Based Translation Validator for LLVM. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 737–742.

[45] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 264–276. doi:10.1145/1480881.1480915

[46] Mojtaba Valizadeh and Martin Berger. 2023. Search-Based Regular Expression Inference on a GPU. *Proc. ACM Program. Lang.* 7, PLDI, Article 160 (June 2023), 23 pages. doi:10.1145/3591274

[47] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2021. Vectorization for digital signal processors via equality saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 874–886. doi:10.1145/3445814.3446707

[48] Henry S Warren. 2013. *Hacker's delight*. Pearson Education.

[49] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. 2014. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*. Technical Report UCB/EECS-2014-54. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html

[50] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (Jan. 2021), 29 pages. doi:10.1145/3434304

[51] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 255–268. https://proceedings.mlsys.org/paper_files/paper/2021/file/cc427d934a7f6c0663e5923f49eba531-Paper.pdf

Received 2025-03-25; accepted 2025-08-12