Minotaur: A SIMD-Oriented Synthesizing Superoptimizer

ZHENGYANG LIU, University of Utah, USA
STEFAN MADA, University of Utah, USA
JOHN REGEHR, University of Utah, USA

A superoptimizing compiler—one that performs a meaningful search of the program space as part of the optimization process—can find optimization opportunities that are missed by even the best existing optimizing compilers. We created Minotaur: a superoptimizer for LLVM that uses program synthesis to improve its code generation, focusing on integer and floating-point SIMD code. On an Intel Cascade Lake processor, Minotaur achieves an average speedup of 7.3% on the GNU Multiple Precision library (GMP)'s benchmark suite, with a maximum speedup of 13%. On SPEC CPU 2017, our superoptimizer produces an average speedup of 1.5%, with a maximum speedup of 4.5% for 638.imagick. Every optimization produced by Minotaur has been formally verified, and several optimizations that it has discovered have been implemented in LLVM as a result of our work.

CCS Concepts: • **Software and its engineering → Translator writing systems and compiler generators**.

Additional Key Words and Phrases: superoptimization, SIMD, program synthesis, peephole optimization

ACM Reference Format:
Zhengyang Liu, Stefan Mada, and John Regehr. 2024. Minotaur: A SIMD-Oriented Synthesizing Superoptimizer. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 326 (October 2024), 25 pages. https://doi.org/10.1145/3689766

1 Introduction

Optimizing compilers emit better code than non-optimizing compilers do, but even so their output is usually far from optimal. Our work started when we noticed substantial opportunities for improvement in the output of LLVM's autovectorizer. As a step towards fixing these, we created Minotaur: a synthesis-based superoptimizer for the LLVM intermediate representation [17] that focuses on LLVM's portable vector operations as well as its x86-64-specific SIMD intrinsics. Our goal is to automatically discover useful optimizations that are missed by LLVM.

Minotaur works on code fragments that do not span multiple loop iterations; it is based on the assumption that existing compiler optimization passes such as loop unrolling, software pipelining, and automatic vectorization will create the necessary opportunities for its optimizations to work effectively. For example, consider this loop, in C, from the compression/decompression utility gzip, where `name` is the base address of a string and `p` is a pointer into the string:

```
do {
  if (*--p == '.') *p = '_';
} while (p != name);
```

When it is compiled by LLVM 18 for a target supporting AVX2 vector extensions, this code is found inside the loop:

Authors' Contact Information: Zhengyang Liu, University of Utah, Salt Lake City, USA, liuz@cs.utah.edu; Stefan Mada, University of Utah, Salt Lake City, USA, stefan.mada@utah.edu; John Regehr, University of Utah, Salt Lake City, USA, regehr@cs.utah.edu.

ACM 2475-1421/2024/10-ART326
https://doi.org/10.1145/3689766

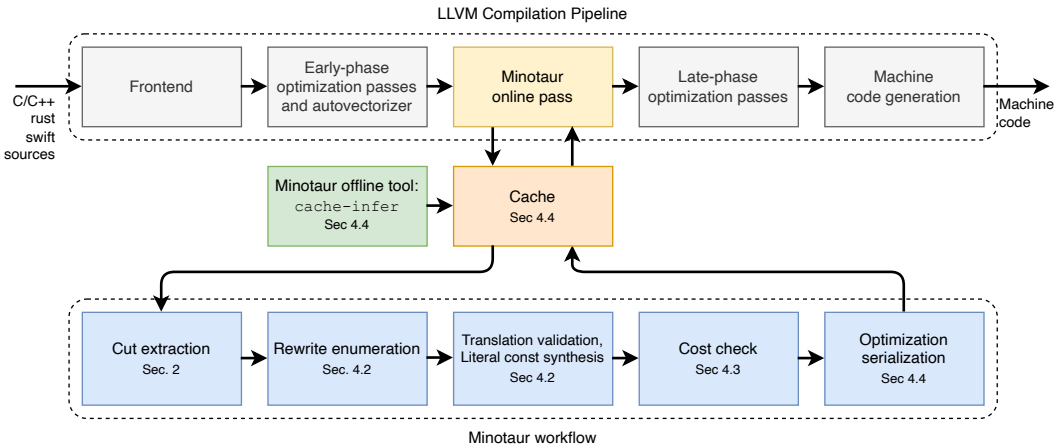


Fig. 1. Overview of how Minotaur works, and how it fits into the LLVM optimization pipeline

```
%1 = shufflevector <32 x i8> %0, poison, <31, 30, 29, 28, 27, ... 4, 3, 2, 1, 0>
%2 = icmp eq <32 x i8> %1, <46, 46, 46, 46, 46, ... 46, 46, 46, 46, 46>
%3 = shufflevector <32 x i1> %2, poison, <31, 30, 29, 28, 27, ... 4, 3, 2, 1, 0>
```

The first shufflevector reverses a 32-byte chunk of the string, the `icmp` instruction checks which elements of the chunk are equal to 46 (ASCII for the period character), and then the second shufflevector reverses the vector containing the results of the computation. This code cannot be optimized further by LLVM 18; when it is lowered to object code and executed on an Intel Cascade Lake processor, it requires 13 uOps, or "micro-operations," processor-internal RISC-like instructions that modern x86 implementations actually execute. Minotaur, on the other hand, automatically determines that the vector reversals are unnecessary, and rewrites the code in this equivalent, but significantly cheaper (three uOps), form:

```
%3 = icmp eq <32 x i8> %0, <46, 46, 46, 46, 46, ... 46, 46, 46, 46, 46>
```

Although SIMD operations are Minotaur's main focus, it also discovers optimizations for scalar code. For example, this code, from the SPEC CPU 2017 benchmark 619.lbm, computes the difference between two floating-point values, and then checks if the result is greater than zero:

```
%0 = fsub float %x, %y
%1 = fcmp ogt float %0, 0.000000e+00
```

Minotaur found that this code is equivalent to checking if the second value is less than the first:

```
%1 = fcmp ogt float %x, %y
```

It is perhaps surprising that LLVM, in 2024, could not perform this simple rewrite, which reduces the computation cost from seven uOps to five. However, it has now been implemented in upstream LLVM as a result of our work.

Figure 1 illustrates Minotaur's high-level structure, and how it fits into LLVM. It works by extracting many different *cuts* from an LLVM function. Each cut serves as the specification for a program synthesis problem, where the objective is to synthesize a new cut that refines the old one and is cheaper. When such a cut is found, Minotaur uses it to rewrite the original LLVM function, and also caches the rewrite.

Reasoning about the correctness of optimizations at the level of LLVM IR can be very difficult; we have repurposed Alive2 [22] to serve as a verification backend. To Alive2, we added formal semantics for Intel-architecture-specific SIMD intrinsics. Reasoning about the relative costs of code

sequences is another difficult problem; the solution adopted by Minotaur is to reuse the LLVM Machine Code Analyzer [10], which has adequately accurate pipeline models for various modern processors. These tools, along with the LLVM compiler itself, form the foundation upon which Minotaur is built.

Research contributions: First, we designed and implemented a domain-specific program transformer that extracts an SSA value from an LLVM function, along with context about how that value was computed. Extracting enough context to permit interesting optimizations, without extracting so much context that the underlying SMT solver was overwhelmed, was an interesting empirical problem. Second, we created a synthesis engine that searches for cheaper code sequences; it enumerates partially symbolic candidates where the instructions are concrete, but constants are symbolic. For this part of our work, we created formal semantics for 165 LLVM intrinsic functions that correspond to SIMD operations supported by x86-64 processors, and added these to Alive2. We also modified Alive2 to support synthesis of literal constants. Third, to mitigate the large performance overhead of running program synthesis at compile time, we developed infrastructure for caching optimizations. Thus, while Minotaur can be hundreds of times slower than `clang -O3` when its cache is cold, with a warm cache it is just 3% slower, when building the SPEC CPU 2017 benchmarks.

We performed a detailed evaluation of Minotaur's ability to speed up code, showing that it can find numerous optimizations that LLVM fails to perform, and also that it can achieve speedups on a variety of real-world libraries and benchmarks. Minotaur is also useful for compiler developers, and in fact several optimizations it has discovered have now been implemented in upstream LLVM.

2 Cutting LLVM Functions

Using a typical function in LLVM IR as the specification, it is not practical to directly synthesize an optimized version of that function. The state of the art in program synthesis simply does not scale up to the size of LLVM functions found in the wild. Instead, Minotaur takes a divide-and-conquer approach: we individually attempt to optimize each instruction in an LLVM function by extracting a *cut*—a subset of that instruction's dependencies. If this cut of LLVM instructions can be optimized by program synthesis then, by the compositionality of refinement, so can the original LLVM function. The rest of this section describes this process in more detail.

2.1 Problem Statement

Given a function F, an instruction I within F, and a depth bound B, our goal is to create a new LLVM function C that:

(1) is loop-free,
(2) returns the value computed by I, and
(3) contains every instruction in F that can be reached by following up to B backwards data, control, and memory dependency edges.

Informally, we can think of C as summarizing a subcomputation in F, that is (hopefully) tractable for an SMT solver to reason about.

When an instruction is part of C but its inputs are not, they become free inputs—these are implemented by adding them as parameters to C. We can think of every instruction that is not part of the cut as being part of a residual function R. However, note that Minotaur does not explicitly construct R—it computes and optimizes C, and then applies the discovered optimization, if any, directly to F using a rewrite mechanism described in Section 4.4.

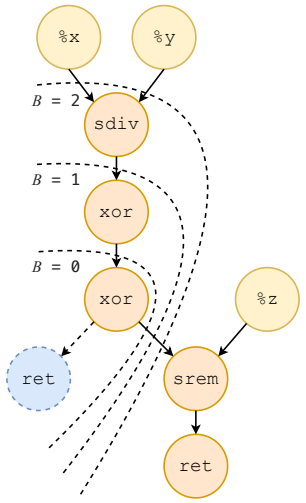


Fig. 2. Example of cutting an LLVM function

2.2 Example

Consider this LLVM function that takes three 64-bit arguments and returns a 64-bit value, where `sdiv` is signed integer division and `srem` is the signed integer modulus operator:

```
define i64 @f(i64 %x, i64 %y, i64 %z) {
  %a = sdiv i64 %x, %y
  %b = xor i64 %a, -1
  %c = xor i64 %b, -1
  %d = srem i64 %c, %z
  ret i64 %d
}
```

Figure 2 illustrates various cuts of this function. If we cut this function with respect to `%c` with $B = 0$ then we get the following decomposition (however, again, please bear in mind that Minotaur does not actually construct R—we show it here to make the explanation concrete):

```
define i64 @r0(i64 %x, i64 %y, i64 %z) {
  %m = sdiv i64 %x, %y
  %n = xor i64 %m, -1
  %o = call i64 %c0(i64 %n)
  %p = srem i64 %o, %z
  ret i64 %p
}
```

```
define i64 @c0(i64 %t1) {
  %t2 = xor i64 %t1, -1
  ret i64 %t2
}
```

This is not useful, the cut `c0` contains too little context to support any optimizations. If we cut `f` with respect to `%c` with $B = 1$ then we get:

```
define i64 @r1(i64 %x, i64 %y, i64 %z) {
  %m = sdiv i64 %x, %y
  %o = call i64 @c1(i64 %m)
  %p = srem i64 %o, %z
  ret i64 %p
}
```

```
define i64 @c1(i64 %t1) {
  %t2 = xor i64 %t1, -1
  %t3 = xor i64 %t2, -1
  ret i64 %t3
}
```

This decomposition is useful: `c1` can be optimized to simply return its argument. If we increase the depth bound to two, then we would get:

```
define i64 @r2(i64 %x, i64 %y, i64 %z) {
  %o = call i64 @c2(i64 %x, i64 %y)
  %p = srem i64 %o, %z
  ret i64 %p
}
```

```
define i64 @c2(i64 %t1, i64 %t2) {
  %t3 = sdiv i64 %t1, %t2
  %t4 = xor i64 %t3, -1
  %t5 = xor i64 %t4, -1
  ret i64 %t5
}
```

Here the cut `c2` can again be optimized (it can just return `%t3`), but now the solver must reason about a 64-bit signed division—operations like this are difficult and frequently lead to timeouts. Choosing an appropriate depth bound is an empirical problem that we address in Section 5.2.

2.3 Correctness Argument

The composition of R and C is equivalent to the original function: $F = R \circ C$. In other words, the decomposition of F into R and C preserves the behavior of the original function—the difference is simply that some dependency edges that were previously internal to F now cross the boundary between R and C.

Next, if Minotaur can synthesize C', an optimized function that refines C, then we can compose that with the residual function to get a new function $F' = R \circ C'$. Since refinement is compositional, it follows that F' refines F, which is the property that we need for Minotaur to be a correct optimizer. The details of establishing a refinement relation between functions in LLVM IR were presented by Lopes et al. [22].

Alive2 is intended to be a sound refinement checker for LLVM IR for LLVM functions that do not contain loops. We avoid this potential unsoundness by ensuring that C is loop-free, in which case C' is also loop-free since Minotaur never synthesizes a loop.

2.4 Detailed Solution

Algorithm 1 shows the procedure that Minotaur uses to extract a cut. It works in two phases. In the first, Minotaur determines which instructions will be part of the cut, using a depth-bounded depth-first search. During the search, two sets, *Harvest* and *Unknown*, are propagated which will be used in the second phase for constructing the cut. Minotaur uses LLVM's LoopInfo pass [18] to identify loops in the source function. If instruction I is in a loop, Minotaur will only extract instructions that are defined inside the loop. If the loop is nested, Minotaur will only extract instructions that are defined inside the innermost loop. Minotaur gives up if the loop is irregular. If I is not in a loop, Minotaur will skip the instructions that are in a loop. Minotaur marks non-intrinsic function calls, operations on global variables, and operations that are not recognized by Alive2 as unsupported. All unsupported operations, operations that are beyond the depth limit, and operations that are outside the loop are discarded and replaced with free inputs.

For each conditional branch instruction, Minotaur extracts the branch condition, since these carry control flow information that is useful during synthesis. Similarly, when it extracts a load from memory, Minotaur consults LLVM's MemorySSA pass [19] to get a list of stores that potentially influence the loaded value. MemorySSA marks memory operations with one of the three memory access tags: `MemoryDef`, `MemoryUse`, and `MemoryPhi`. Each memory operation is associated with a version of the memory state. A `MemoryDef` can be a store, a memory fence, or any operation that creates a new version of the memory state. A `MemoryPhi` combines multiple `MemoryDefs` when control flow edges merge. A `MemoryUse` is a memory instruction that does not modify memory, it

Algorithm 1 Extract a cut from an LLVM function

```
1:  function ExtractCut(F: Function, I: Instruction, B: ℕ)
2:      if I is in a loop then
3:          AllowedBasicBlocks ← all basic blocks in I's loop (innermost loop if nested)
4:      else
5:          AllowedBasicBlocks ← all basic blocks in F that is not in a loop
6:      Harvested ← ∅
7:      Unknown ← ∅
8:      Visited ← ∅
9:      WorkList ← { (I, 0) }
10:
11:                                                  ▷ Phase 1: Instruction extraction
12:     while WorkList is not empty do
13:         (WI, Depth) ← WorkList.pop()
14:         if WI ∈ Visited then
15:             continue
16:         Insert WI into Visited
17:         if Depth > B then
18:             Insert WI into Unknown
19:             continue
20:         if WI is not supported then
21:             Insert WI into Unknown
22:             continue
23:         BB ← WI's basic block
24:         if BB ∉ AllowedBasicBlocks then
25:             Insert WI into Unknown
26:             continue
27:         Insert WI into Harvested
28:         if WI is a Load instruction then
29:             M ← WI's linked MemoryPhi or MemoryDef
30:             if M is a MemoryDef ∧ M is a store then
31:                 MI ← M's stored value
32:                 Push (MI, Depth + 1) into WorkList
33:         else
34:             for all operand Op in WI do
35:                 Push (Op, Depth + 1) into WorkList
36:         T ← terminator of WI's basic block
37:         if T is a conditional branch instruction then
38:             TI ← T's condition value
39:             Push (TI, Depth + 1) into WorkList
40:     Insert every terminator instruction in F to Harvested
41:
42:                                          ▷ Phase 2: Construct a loop-free LLVM function
43:     Clone F into C
44:     Delete instructions in C except those in Harvested
45:     Delete all back-edges in C
46:     Add values in Unknown to C as function arguments
47:     Create return instruction for I in C
48:     return C
```

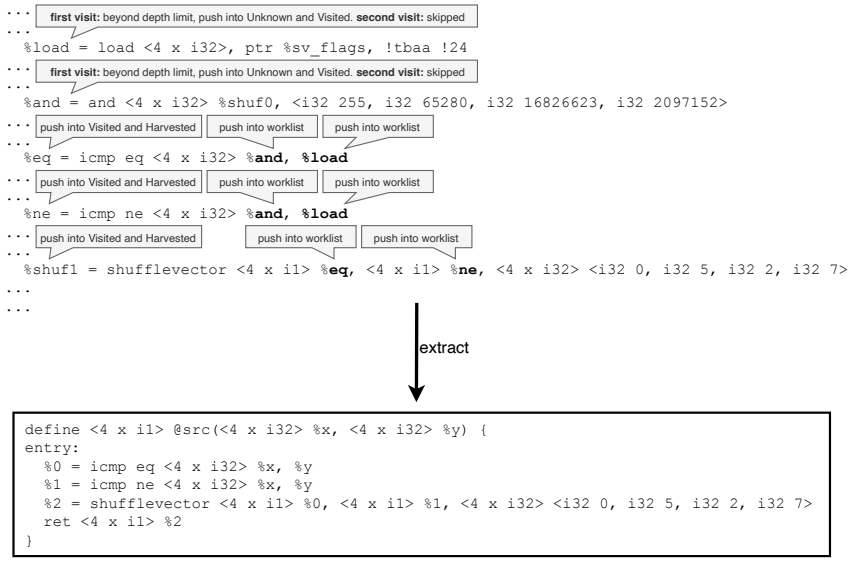


Fig. 3. Example of cut extraction

only reads the memory state created by `MemoryDef` or `MemoryPhi`; a load instruction is always a `MemoryUse`. Because it must overapproximate, Minotaur is conservative when finding load-affecting stores: it starts from the loads in `MemoryUse`'s memory version and walks along the MemorySSA's def-use chain. When the associated memory operation is a `MemoryDef`, it checks if the operation is a store and pushes the stored value into the worklist, to provide Minotaur a more specific context to optimize the load instruction.

In the second phase, Minotaur builds the extracted function; it does this by cloning the original function and then deleting all instructions that are not in the cut. Minotaur then deletes all loop backedges, so that the extracted function is loop-free. Finally, a return instruction is added to return the value computed by the instruction that is the basis for the cut.

Figure 3 shows an example of cut extraction for value `%shuf1`. The cutting algorithm starts on `%shuf1` and walks along the def-use chain to extract the instructions that are involved in the computation of `%shuf1`. A new function is created to hold the extracted instructions shown in the bottom of the figure.

2.5 Relation to Previous Cut-Based Superoptimizers

Minotaur's cut extraction algorithm is fundamentally more aggressive than Bansal and Aiken's approach [4], which extracted a small window of sequential instructions. It is also considerably more aggressive than Souper [27], which had a very limited view of control flow and refused to consider memory operations, vector operations, and floating-point operations.

3 Formalizing Vector Intrinsics in Alive2

In order for Minotaur to use Alive2 as a verification backend, we had to modify Alive2 to support a number of x86-64-specific vector intrinsics.

Algorithm 2 Semantics of `@llvm.x86.{avx|avx2|avx512}.pavg.{b|w}`

```
1: PAVG<LANES, BITS, MASKED>(S_a, S_b, PassThrough, Mask)
2: for each i in range [0 to LANES - 1]
3:     if MASKED ∧¬ Mask[i] then
4:         S_ret[i].val ← PassThrough[i].val
5:         S_ret[i].poison ← PassThrough[i].poison
6:     else
7:         S_ret[i].val ← (S_a[i].val +_BITS S_b[i].val +_BITS 1) /_BITS 2
8:         S_ret[i].poison ← S_a[i].poison ∨ S_b[i].poison
```

Algorithm 3 Semantics of `@llvm.x86.{sse2|avx2|avx512}.pmadd.wd`

```
1: PMADD.WD<LANES, MASKED>(S_a, S_b, PassThrough, Mask)
2: for each i in range [0 to LANES - 1]
3:     if MASKED ∧¬ Mask[i] then
4:         S_ret[i].val ← PassThrough[i].val
5:         S_ret[i].poison ← PassThrough[i].poison
6:     else
7:         S_ret[i].val←sext(S_a[2·i].val ×_16 S_b[2·i].val) +_32 sext(S_a[2·i + 1].val ×_16 S_b[2·i + 1].val)
8:         S_ret[i].poison←S_a[2·i].poison ∨ S_b[2·i].poison ∨ S_a[2·i + 1].poison ∨ S_b[2·i + 1].poison
```

3.1 Background: Vectors in LLVM

LLVM uses a typed, SSA-based intermediate representation (IR). It supports a derived *vector type*; for example, a vector with eight lanes, where each element is a 64-bit integer, would have type `<8 x i64>`. Many LLVM instructions, such as arithmetic operations, logical operations, and pointer arithmetic, can operate on vectors as well as scalars. IR-level vectors are target-independent; backends attempt to lower vector operations to native SIMD instructions, if available.

Beyond the vertical ALU instructions that are element-wise vector versions of scalar instructions, LLVM supports a variety of horizontal vector reduction intrinsics and an assortment of memory intrinsics such as vector load and store, strided load and store, and scatter/gather. Additionally, there are three vector-specific data movement instructions: *extractelement* retrieves the element at a specified index from a vector; *insertelement* non-destructively creates a new vector where one element of an old vector has been replaced with a specified value; and, *shufflevector* returns a new vector that is a permutation of two input vectors using elements whose indices are specified by a constant mask vector. Finally, to provide direct access to platform-specific vector instructions, LLVM provides numerous intrinsic functions such as `@llvm.x86.avx512.mask.cvttps2dq.512`, aka "convert with truncation packed single-precision floating-point values to packed signed doubleword integer values."

3.2 Assigning a Formal Semantics to Vector Intrinsics

The version of Alive2 that we started with supports most of the core LLVM intermediate representation, including its target-independent vector operations. However, Alive2 did not have a semantics for any of the numerous LLVM-level intrinsic functions that provide predictable, low-level access to target-specific vector instructions.

We added semantics for 165 x86-64 vector intrinsics to Alive2; these come from the SSE, AVX, AVX2, and AVX-512 ISA extensions. The resulting version of Alive2 supports the x86 vector intrinsics that are widely used and that an SMT solver can reason about fairly efficiently. This includes special vertical operations that do not overlap with LLVM's platform-independent vector instructions

(such as `@llvm.x86.avx2.psign.b`), special data movement intrinsics that operate differently than LLVM's `shufflevector` (such as `@llvm.x86.avx512.packsswb.512`), and special horizontal operations that are only available in x86 processors (such as `@llvm.x86.avx512.vpdpbusd.512`). We have not supported dedicated cryptographic operations (that an SMT solver is unlikely to be able to make use of within a reasonable amount of CPU time), nor have we supported some unpopular SIMD intrinsics that we have not observed being used in programs that we have compiled with Minotaur.

There is significant overlapping functionality between vector instructions; for example, there are eight different variants of the `pavg` instruction that computes a vertical (element-wise) average of two input vectors. To exploit this overlap, our implementation is parameterized by vector width, vector element size, and by the presence of a masking feature that, when present, uses a bitvector to suppress the output of vector results in some lanes. Algorithms 2 and 3 show our implementations of the `pavg` (packed average) and `pmadd.wd` (packed multiply and add) families of instructions. This parameterized implementation enabled a high level of code reuse, and our implementation of these semantics is only 660 lines of C++. Our semantics differ slightly from the semantics of the corresponding processor instructions because, at the LLVM level, we must account for poison values—a form of deferred undefined behavior. Our strategy for dealing with poison follows the one used by existing LLVM vector instructions: poison propagates lane-wise, but does not contaminate non-dependent vector elements.

3.3 Validating our Changes to Alive2

We made heavy use of randomized differential testing to ensure that our new intrinsics correctly implement the intended semantics. Each iteration of our tester randomly chooses constant inputs to a single vector intrinsic and then:

(1) Creates a small LLVM function passing the chosen inputs to the intrinsic.
(2) Evaluates the effect of the function using LLVM's JIT compilation infrastructure [20]. The effect is always to produce a concrete value, since the inputs are concrete.
(3) Converts the LLVM function into Alive2 IR and then asks Alive2 whether this is refined by the output of the JITted code.

Any failure of refinement indicates a bug. When we fielded this tester, it rapidly found 11 cases where our semantics produced an incorrect result, usually for some edge case. For example, the semantics for `pavg` were incorrect when the sum overflowed. It also found three cases where Minotaur generated SMT queries that failed to typecheck. For example, we set the wrong lane size when parameterizing the semantics for `psra.w` and `psra.d`, causing the solver to reject our malformed queries. After we fixed these 14 bugs, extensive testing failed to find additional defects.

4 Synthesizing Optimizations

For every cut extracted from an LLVM function, Minotaur's goal is to synthesize a cheaper way to compute the value returned by that cut. It does this by enumerating *candidates*—code fragments that potentially refine the current cut. When a candidate is found that refines the original cut, Minotaur consults a cost model. If the new code is cheaper than the original cut, Minotaur applies the rewrite to the function that is being optimized.

4.1 Designing an Appropriate Synthesis Procedure

A delicate part of designing a practical program synthesis algorithm is determining how much of the search is pushed to the solver, and how much searching gets done by code outside the solver. At one extreme, as the Denali paper [15] points out, we could simply give the SMT solver a conjecture

Operation Type	Instructions
Unary integer	ctpop, ctlz, cttz, bitreverse, bswap
Unary floating point	fneg, fabs, fceil, ffloor, frint, fround, fnearbyint, froundeven
Binary integer	add, sub, mul, udiv, sdiv, umax, umin, smax, smin
Binary floating point	fadd, fsub, fmul, fdiv, frem, fmaximum, fminimum, fmaxnum, fminnum
Bitwise	and, or, xor, shl, lshr, ashr
Comparison	icmp, fcmp, select
Conversion	zext, sext, trunc, fptrunc, fpext, fptosi, sitofp, fptoui, uitofp
Data movement	extractelement, insertelement, shufflevector
SIMD intrinsics	165 vector intrinsics mapping to SSE, AVX, AVX2, and AVX-512 instructions

Table 1. Operations that Minotaur can synthesize. The data movement and SIMD intrinsic instructions require vector operands. The rest of the operations apply to both scalar and vector values.

of the form "No program of the target architecture computes P in at most eight cycles." If the solver can disprove this conjecture, then its counterexample will tell us how to compute P in eight or fewer cycles. This kind of query is asking the solver to do all of the work of finding a program that disproves the conjecture, including reasoning about the costs of various alternatives, in a single, complicated query—this is very heavy lifting. At the other extreme, we could enumerate completely concrete candidates, and use the SMT solver only to perform the necessary refinement checks. The problem with this approach is literal constants: even a single 64-bit constant in the synthesized code will require us to enumerate and check 2^{64} alternatives; this is clearly infeasible.

We spent a considerable amount of time investigating different points between these extremes, and finally settled on a design that makes things as easy as possible for the solver, but without exploring all possible choices of values for literal constants. Minotaur creates *partially symbolic* candidates where instructions are represented concretely, but constants are symbolic. This gives a reasonably tractable enumeration space without giving up synthesis power. Our rationale for this design is that, based on extensive experience with LLVM and Alive2, a lot of individual refinement checks that we want to perform—especially those that contain multiplications, divisions, floating point operations, and pointer indirections—are already very difficult.

4.2 Synthesis in Minotaur

Algorithm 4 describes Minotaur's synthesis procedure. In Phase 1, it creates a pool of instructions whose operands are selected from the available SSA values in the current cut (a dominance check is not required since cuts are constructed in such a way that every existing SSA definition dominates the synthesized portion of the function), from symbolic constants, and from *holes* that represent instructions that have not yet been enumerated. The list of instructions that Minotaur can synthesize is shown in Table 1. The description in Algorithm 4 only shows the case for instructions taking two operands, and it also omits a number of simple pruning strategies that are useful in practice, such as avoiding enumeration of redundant versions of commutative operations. In Phase 2 of the synthesis procedure, instructions from the pool are used to recursively fill holes; this procedure terminates when all holes are filled (in which case a complete candidate has been generated) or when at least one hole remains, but there is no remaining instruction budget to fill it (in which case the incomplete candidate is discarded). A subtlety here is that LLVM's `bitcast` instruction, which changes the type of an SSA value without changing its representation, does not count towards the instruction limit. This is because Minotaur takes a low-level, untyped view of values. For example, it internally treats a 16-way vector of 8-bit values the same as an 8-way vector of 16-bit values: both of these are simply 128-bit quantities. This lack of type enforcement allows Minotaur to find

Algorithm 4 Minotaur's Synthesis Procedure

```
1:  function SynthesizeRefinements(Cut: Function, InstLimit: ℕ, TimeLimit: ℕ)
2:                                                  ▷ Phase 1: Populate the instruction pool
3:      Inputs ← all the SSA definitions in Cut
4:      InstPool ← ∅
5:      for all op in binary operations listed in Table 1
6:          InstPool ← InstPool ∪ { (op hole, hole), (op sym-const, hole), (op hole, sym-const) }
7:          for all input1 in Inputs
8:              InstPool ← InstPool ∪ { (op input1, sym-const), (op sym-const, input1) }
9:              InstPool ← InstPool ∪ { (op input1, hole), (op hole, input1) }
10:             for all input2 in Inputs
11:                 InstPool ← InstPool ∪ { (op input1, input2) }
12:     for all other operations in Table 1                    ▷ omitted for brevity
13:         ...
14:
15:                                                 ▷ Phase 2: Generate partially-symbolic candidates
16:     WorkList ← { (ret hole), (ret sym-const) }
17:     Candidates ← Inputs
18:     while WorkList ≠ ∅
19:         I ← WorkList.pop()
20:         if I does not contain holes then
21:             Candidates ← Candidates ∪ { I }
22:             continue
23:         for all Hole in I
24:             if CountNewInsts(I) ≥ InstLimit then
25:                 continue
26:             for all Inst in InstPool
27:                 J ← I with Hole substituted by Inst
28:                 if TargetTransformInfoCost(J) ≥ TargetTransformInfoCost(Cut) then
29:                     continue
30:                 WorkList ← WorkList ∪ { J }
31:
32:                                         ▷ Phase 3: Refinement checking and constant synthesis
33:     Sort Candidates by TargetTransformInfoCost
34:     StartTime ← time()
35:     Refinements ← ∅
36:     for all C in Candidates
37:         if C does not contain symbolic constants then
38:             if Alive2 claims that C refines Cut then
39:                 Refinements ← Refinements ∪ { C }
40:         else
41:             Build exists-forall query to get a model for symbolic constants
42:             if satisfiable then
43:                 C' ← C with symbolic constants substituted by the constants in the model
44:                 Refinements ← Refinements ∪ { C' }
45:         if time() - StartTime ≥ TimeLimit then
46:             break
47:     return Refinements
```

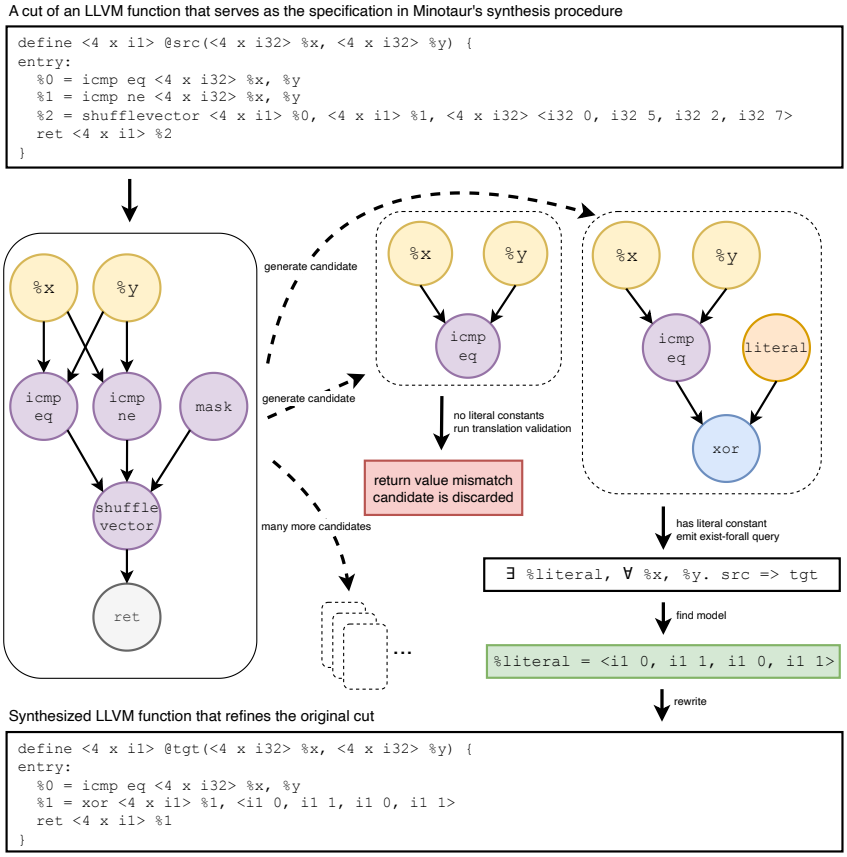


Fig. 4. Example of synthesizing a rewrite that contains literal constants. Purple nodes are instructions reused from the original cut; blue and orange nodes are synthesized instructions and literal constants.

interesting, low-level optimizations such as those that use bitwise operations to rapidly perform certain floating point operations.

In Phase 3 of Algorithm 4, Minotaur uses Alive2 to eliminate every candidate that does not refine the specification. First, we sort the candidates in order of increasing cost using LLVM's TargetTransformInfo [21]: a cost model that roughly captures execution cost on the target, and is cheap to compute. We do this to ensure that likely-beneficial rewrites are tested first, before the synthesis time limit is reached. For candidates that do not contain symbolic constants, we can use Alive2 as-is. To support symbolic constants, we modified Alive2 to wrap its refinement check in an exists-forall query. In other words, Minotaur asks the question: "Does there exist a valuation of the symbolic constants such that the synthesis candidate refines the specification for all possible values of the inputs?" When such a query is satisfiable, the model returned by the solver can be inspected to find satisfying values of the symbolic constants in the candidate, which now become literal constants, giving a complete, sound optimization. To avoid potentially-expensive exists-forall queries, we experimented with various techniques such as generalization by substitution [11]. However, these failed to outperform exists-forall queries, in the version of Z3 that we used (4.12.4). Figure 4 illustrates Minotaur's synthesis procedure.

4.3 Identifying Profitable Rewrites

The output of Algorithm 4 is a list of candidates that all refine the cut. Of these, we want to choose the best one—but predicting throughput of code running on modern microprocessors is not straightforward. We leverage the LLVM Machine Code Analyzer (LLVM-MCA) [10], which was created to help developers improve performance-critical code. It is an interactive tool that emits a graphical depiction of pipeline behavior, but its functionality can also be accessed programmatically, and this is what Minotaur does, after lowering each candidate to x86-64 object code. Then, Minotaur only applies a rewrite if its estimated cost, using LLVM-MCA, is lower than that of the original cut, and lower than that of any other synthesized refinement of the original cut.

Although LLVM-MCA can estimate the cycle cost of LLVM functions, we instead use the number of uOps ("micro-operations," a modern x86-64 processor's internal instruction set) as the estimated cost. This choice was driven by empirical data: after extensive experimentation, we determined that, for our purposes, uOps are a better performance predictor than cycles.

4.4 Representing and Caching Rewrites

Minotaur stores each potential rewrite as a pair: (C, S) where C is a cut, represented by a function in LLVM Intermediate Representation (IR), and S is a rewrite description—an expression in Minotaur's own intermediate representation that describes a different way to compute the return value of C. Rewrite descriptions are directed acyclic graphs containing nodes that represent operations, and edges representing data flow. Although the elements found in Minotaur IR are similar to those found in LLVM IR, we could not reuse LLVM IR to represent rewrites since LLVM IR does not support incomplete code fragments, and also rewrites must contain enough information to support connecting the new code in the rewrite to code in the unoptimized function.

To support caching, rewrites must be serializable. The cut C can be serialized using existing LLVM functionality, and we created a simple S-expression syntax for serializing the S part. Figure 5 shows the syntax of the IR. For example, if the returning value of C, a 32-bit instruction is replaced by left shift by one bit position, the textual format for the expression is `(shl (val i32 %0), (const i32 1), i32)`.

Rewrites are cached in a Redis instance: this implementation choice allows the cache to be persistent across multiple Minotaur runs and also makes the cache network-accessible. Synthesis can be done online—during compilation—but also offline, in a mode where Minotaur extracts cuts into the Redis cache but does not perform synthesis. In this mode, compilation is only slowed down by a few percent. Minotaur's offline mode is designed for batch processing. In this mode, a separate program called `cache-infer` retrieves cuts from the cache, runs synthesis on them, and stores any optimizations that it discovers back into the cache. Unlike the online mode, which runs synthesis tasks one after the other, offline mode can run all synthesis jobs in parallel.

4.5 Integration with LLVM

Minotaur is loaded into LLVM as a shared library where it runs as an optimization pass. We arranged for it to run at the end of LLVM's auto-vectorization pipeline. We invoke LLVM's Dead Code Elimination pass after Minotaur to clean up the resulting code.

5 Evaluation

Our primary evaluation metric for Minotaur is its ability to speed up legacy application code, compared to an optimized build using LLVM 18. Secondarily, we look at Minotaur's impact on compile time, optimizations that have been integrated into upstream LLVM based on our work, and other issues.

Op	::=	*Inst* | *Constant* | *Value*
Inst	::=	(*UnaryOp Op*, *Type*) | (*BinaryOp Op*, *Op*, *Type*) | (*Conversion Op*, *Type*) | (insertelement *Op*, *Op*, *Op*) | (extractelement *Op*, *Op*) | (shufflevector *Op*, *Op*, *Constant*) | (*Comparison Op Op*) | (select *Op*, *Op*, *Op*) | (*Intrinsic Op*, *Op*)
Constant	::=	(const *Type* `number-literal`)
Value	::=	(val *Type* `llvm-identifier`)
Type	::=	*ScalarType* | <elements × *ScalarType*>
ScalarType	::=	i1 | i8 | i16 | i32 | i64 | half | float | double | fp128
BinaryOp	::=	xor | and | or | add | sub | mul | udiv | sdiv | ashr | lshr | shl | umax | umin | smax | smin fadd | fsub | fmul | fdiv | copysign | fmaximum | fminimum | fmaxnum | fminnum
UnaryOp	::=	ctpop | ctlz | cttz | bswap | bitreverse | ret fneg | fabs | fceil | ffloor | frint | fround | ftrunc | fnearbyint | froundeven
Conversion	::=	zext | sext | trunc | fptrunc | fpext | fptosi | sitofp | fptoui | uitofp
Comparison	::=	eq | ne | ult | ule | slt | sle | oeq | ogt | oge | olt | ole | one | ord | ueq | ugt | uge | ult | ule | une | uno
Intrinsic	::=	ssse3.phadd.d.128 | avx2.pavg.b | avx512.pmaddubs.w.512 | . . . (165 intrinsics in total)

Fig. 5. Syntax for Minotaur rewrites

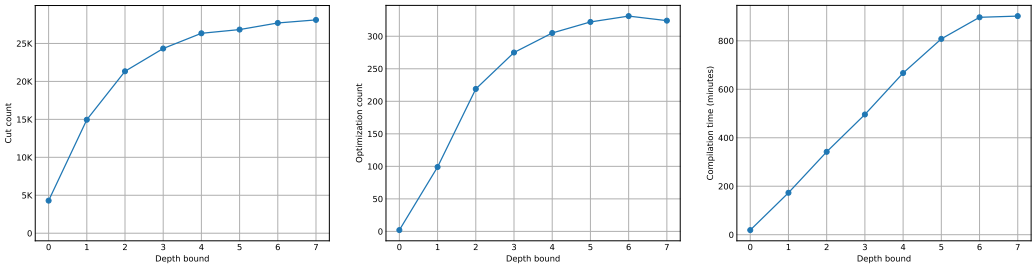
5.1 Correctness

Every optimization discovered by Minotaur has been formally verified by Alive2. Even so, bugs might remain in the instruction semantics that we have added to Alive2, in our cut extractor, in our rewrite mechanism, in Alive2, or in Z3. To defend against implementation errors, we have compiled numerous open source applications using Minotaur, and then run those applications' test suites, to ensure that they were not miscompiled. Furthermore, we have compiled SPEC CPU 2017 using Minotaur and used the SPEC drivers to ensure that all of its benchmarks behave as expected.

5.2 Effect of Depth Bounds in the Cut Extractor

It is important for Minotaur to extract cuts that are of an appropriate size. If they are too large, compile times suffer and also the SMT solver can be overwhelmed, leading to timeouts; if cuts are too small, then they form an insufficient basis for driving an optimization. To determine a good value for B, the depth parameter to the cut extraction procedure shown in Algorithm 1, we performed an empirical study. We started with FlexC's benchmark suite [36], a collection of 2,386 compilable, non-trivial C functions containing loops from FFMPEG, FreeImage, DarkNet, xz, bzip2, and the LivermoreC benchmark. When compiled to LLVM IR, these functions contain a total of 123,062 instructions; thus, our cut extractor was invoked 123,062 times for each depth bound. We chose this code as the basis for our experiment because it is derived from real applications while also being small enough to keep compile times manageable (compared to, e.g., SPEC CPU 2017, which is much larger).

We then ran Minotaur on these functions with all depth bounds from 0–7, measuring the number of unique cuts that were extracted, the number of optimizations found, and the compilation time. We used a one-minute timeout for individual Z3 queries, and we also gave Minotaur a total of up to five minutes to synthesize an optimized version of each cut. Figure 6 summarizes the results of this experiment. The number of unique cuts that are extracted grows quickly with B, but eventually begins to saturate simply because the functions being compiled do not always have very long dependency chains. The number of synthesized optimizations also grows quickly, but it peaks when $B = 6$ and then it decreases because the size of the cuts causes many solver timeouts. Finally, the



(a) Unique cuts extracted (b) Unique opts. synthesized (c) Compilation time

Fig. 6. Evaluating the effect of varying B, the depth bound for cut extraction

	memory	FP vector	FP scalar	integer vector	integer scalar	overall
Number of rewrites	3	17	4	191	109	324
Geomean speedup	1.0605x	1.0600x	1.0572x	1.0142x	1.0506x	1.0610x
Contribution to speedup	0.65%	1.64%	5.90%	75.57%	16.23%	100%

Table 2. Results of an ablation study based on optimization categories

total compile time increases smoothly with the depth bound, eventually leveling off as most solver queries time out.

For the experiments in the rest of the evaluation section, we chose $B = 4$ because this gets close to the maximum observed number of optimizations without requiring exorbitant compile times. It seems likely that there is room for improvement in this aspect of Minotaur: perhaps the depth bound should be determined adaptively. In this scenario, we would extract more and more components into the cut, until either an optimization is found or else the solver begins to time out. We leave explorations of this nature for future work.

5.3 What Kind of Optimizations Matter Most?

To determine which of Minotaur's optimizations matter most, we performed an ablation study, again using the FlexC benchmark suite that we described in Section 5.2. We split the optimizations that Minotaur found into five categories: memory, floating-point vector, floating-point scalar, integer vector, and integer scalar. Then, we ran Minotaur in a way that omitted each of these categories of optimizations. As shown in Table 2, integer vector optimizations produce the most rewrites, and also produce the majority of the observed speedup.

5.4 Speedups for Benchmarks and Applications

In this section, we show how Minotaur speeds up real-world benchmarks and applications.

Experimental setup. We used two machines for our evaluation. The first has an Intel Xeon Gold 6210U processor running at 2.5 GHz, and has 20 cores; this implements the Cascade Lake microarchitecture [13] and supports the AVX-512 instruction set. The second has an AMD Ryzen 5950X processor running at 3.4 GHz, and has 16 cores; this processor implements the Zen 3 microarchitecture [2]. Both machines run Linux and were idle except for a single core running our benchmarks (however, when measuring compile times, as reported in Table 3, we used all cores). To reduce the performance variation caused by frequency scaling, we disabled turbo boost on the

	Intel Cascade Lake					AMD Zen 3				
	Compilation time (minutes)			Stats		Compilation time (minutes)			Stats	
Benchmarks	Cold cache	Warm	Clang	# Cuts	# Opts.	Cold cache	Warm	Clang	# Cuts	# Opts.
SPEC CPU 2017	2,337	3	3	109,177	2,683	2,580	3	3	114,612	2,820
gmp-6.2.1	440	< 1	< 1	9,170	336	445	< 1	< 1	9,265	387
libYUV	2,196	< 1	< 1	6,849	334	2,193	< 1	< 1	6,809	357

Table 3. Minotaur's effect on compilation time

Intel machine and core performance boost on the AMD machine. We also disabled simultaneous multithreading on both machines.

We invoked LLVM with the `-march=native` compilation flag to ask it to take maximum advantage of processor features; we left other compilation flags unchanged, except where noted. All benchmarks are compiled at the `-O3` optimization level. We set the timeout for Z3 [9] queries to one minute. Finally, for each instruction that it tries to optimize, Minotaur gives up if no solution is found within five minutes.

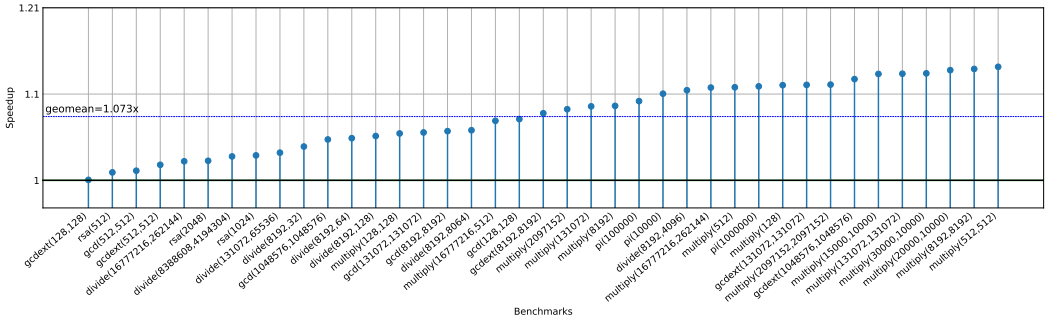
Benchmark selection. We evaluate on SPEC CPU 2017[1] because it is a widely accepted standard benchmark. We only evaluate on the *speed* subset of the SPEC suite, and we omit 648.exchange, 607.cactuBSSN, 621.wrf, 627.cam4, 628.pop2, 649.fotonik3d, and 654.roms as they contain Fortran code. We additionally use GMP, the GNU Multiple Precision Library,[2] and libYUV,[3] which is used by Google Chrome/Chromium for manipulating images in the YUV format. We chose these libraries because they have been heavily tuned for performance, they are loop-intensive, and they come with performance benchmark suites that we could simply reuse.

Compile times. Table 3 shows how long it takes Minotaur to build our benchmarks, along with the number of potentially optimizable values and the number of optimizations found. The compile times are for parallel builds; we set the MAKE's `-j` flag and SPEC CPU 2017's `build_ncpus` configuration to the number of cores on the machine. Minotaur is very slow when it runs with a cold cache because it performs many solver queries. However, with a warm cache, it is only 3% slower than baseline `clang`.

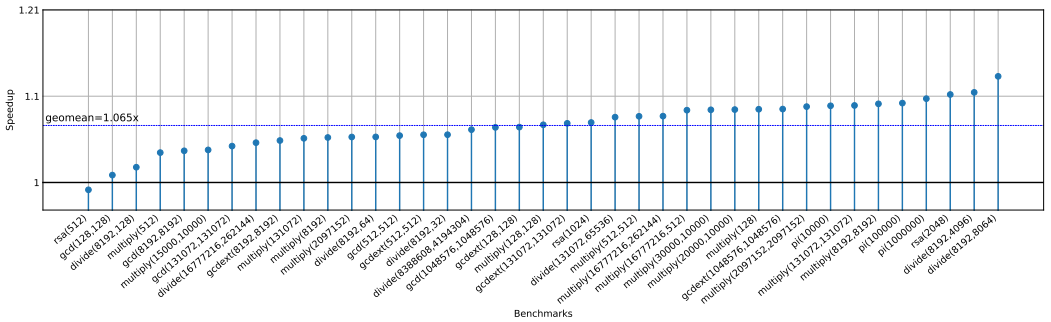
In most cases, Minotaur found more optimizations when targeting the AMD processor. We believe this is because LLVM is more mature targeting AVX2 than AVX-512. Queries with 256-bit vectors are also less likely to timeout in Z3 than are queries with 512-bit vectors.

Optimizing GMP with Minotaur. GMP provides a portable C-language implementation and then, for several platforms, a faster assembly language implementation. For this evaluation, we selected the C implementation, because Minotaur works on LLVM IR and cannot process assembly code at all. The benchmark suite that we used is GMPbench.[4] Figure 7 summarizes the results. When Minotaur targets the Intel Cascade Lake processor, and when the resulting executables are run on that same microarchitecture, all the benchmarks sped up; across all of the benchmarks, the mean speedup was 7.3%. The analogous experiment using the AMD Zen 3 microarchitecture resulted in one benchmark slowing down, and the rest of benchmarks speeding up, for an overall mean speedup of 6.5%.

[1] https://www.spec.org/cpu2017/
[2] https://gmplib.org/
[3] https://chromium.googlesource.com/libyuv/libyuv/
[4] https://gmplib.org/gmpbench



(a) Speedups on Intel Cascade Lake, geomean = 1.073x



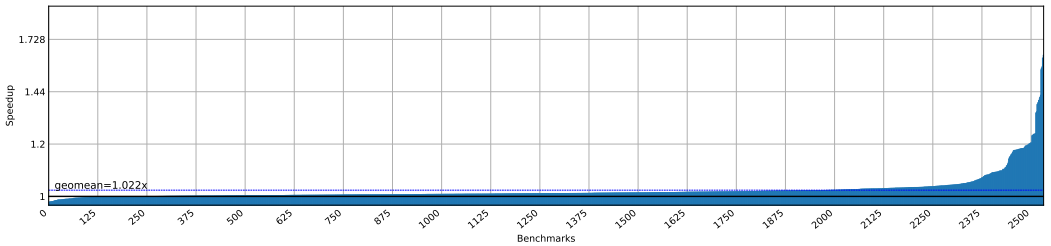
(b) Speedups on AMD Zen 3, geomean = 1.065x

Fig. 7. GNU Multiple Precision Library (GMP) speedups, on a logarithmic scale

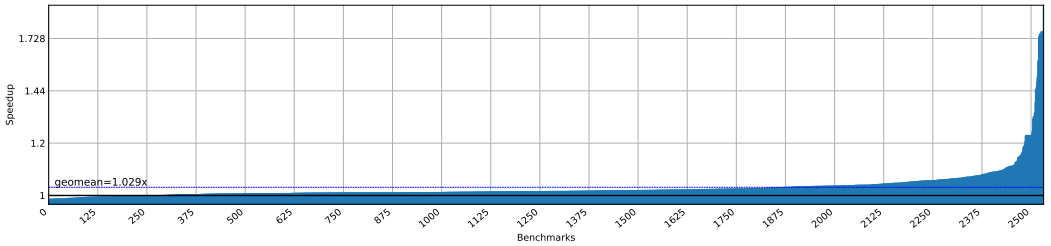
Optimizing libYUV with Minotaur. This library has an extensive test suite, part of which is explicitly intended for performance testing; we used this part as a benchmark. Each test program scales, rotates, or converts a 1280 x 728 pixel image 1,000 times. Figure 8 shows the results of this experiment. When Minotaur targets an Intel processor, 148 programs slowed down, 72 did not change performance, and 2, 312 sped up, for an overall speedup of 2.2%. Targeting an AMD processor, 188 programs slowed down, 85 did not change performance, and 2, 259 sped up, for an overall speedup of 2.9%. Minotaur can make code slower because it looks at optimizations in isolation; it does not attempt to model interactions between optimizations.

libYUV is portable code, but it has already been heavily tuned for performance; most commits to its repository over the last several years have been performance-related. Our hypothesis is that this manual tuning has already eaten up most of the performance gains that we would have hoped to gain from Minotaur. For some time now, Google's released versions of Chrome have been compiled using LLVM; the Chrome engineers have had ample time to ensure that this compiler achieves decent code generation for performance-critical libraries.

Optimizing SPEC CPU 2017 with Minotaur. Figure 9 shows the effect of optimizing the benchmarks from SPEC CPU2017 using Minotaur. When optimizing for, and running on, the Intel processor, we observed a mean speedup of 1.5%. When optimizing for, and running on, the AMD processor, we observed a mean speedup of 1.2%. It is notoriously difficult to speed up the SPEC CPU benchmarks

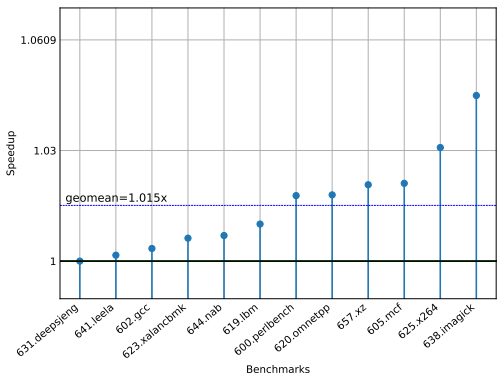


(a) Speedups on Intel Cascade Lake, geomean = 1.022x

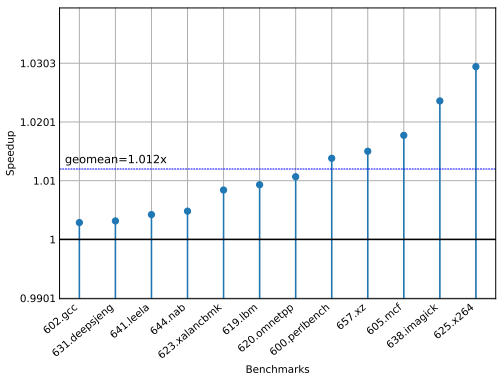


(b) Speedups on AMD Zen 3, geomean = 1.029x

Fig. 8. LibYUV speedups, on a logarithmic scale



(a) Speedups on Cascade Lake



(b) Speedups on Zen 3

Fig. 9. SPEC CPU2017 benchmark performance, on a logarithmic scale

because compiler engineers have already put considerable effort into achieving good code generation for them.

5.5 Impact on Upstream LLVM

In several cases where an optimization discovered by Minotaur seemed to be simple and broadly applicable, we have reported its absence as an LLVM defect, using the project's issue tracker. This section summarizes the results of this informal LLVM-improvement project.

We reported ten missing floating-point optimizations. Five of these (including the one we presented in Section 1) have now been implemented in LLVM. Three of them are in the code review phase: a patch exists and is being discussed by developers. Finally, two of them are being discussed, but a candidate patch does not yet exist.

We also reported five missing vector optimizations. One of these has been fixed, one has a patch that is under review, and three are still being discussed.

5.6 Optimizations Discovered by Minotaur

The purpose of this section is to examine Minotaur's strengths by presenting some optimizations that it found while compiling benchmark programs. None of these optimizations can be performed by the version of LLVM that Minotaur is based on,[5] at its `-O3` optimization level. We present optimizations in an SSA format that is close to LLVM IR, but we have edited it slightly for compactness and legibility.

Example 1. This code is from perlbench in SPEC:

```
%0 = zext <16 x i8> %x to <16 x i16>
%1 = zext <16 x i8> %y to <16 x i16>
%2 = call @llvm.x86.avx2.pavg.w(%0, %1)
%3 = trunc <16 x i16> %2 to <16 x i8>
ret <16 x i8> %3
  =>
%0 = call @llvm.x86.sse2.pavg.b(%x, %y)
ret <16 x i8> %0
```

The unoptimized code zero-extends each 8-bit element of the two input vectors to 16 bits, calls the AVX2 variant of `pavg` to perform element-wise averaging of the extended vectors, and then truncates elements of the resulting vector back to eight bits. The optimized code simply calls an SSE2 version of the `pavg` instruction that operates on 8-bit elements, reducing the uOp cost of the operation from four to one.

Example 2. This code is from libYUV:

```
%0 = call @llvm.x86.avx2.pmadd.wd(%x, <0,1,0,1, ...>)
%1 = call @llvm.x86.avx2.pmadd.wd(%x, <1,0,1,0, ...>)
%2 = sub nsw <8 x i32> %1, %0
ret <8 x i32> %2
  =>
%0 = call @llvm.x86.avx2.pmadd.wd(%x,<1,-1,1,-1, ...>)
ret <8 x i32> %0
```

The `pmadd.wd` (multiply and add packed integers) instruction multiplies signed 16-bit integers element-wise from two input vectors, and then computes its output by adding adjacent pairs of elements from the resulting vector. Thus, the input to this instruction is two 16-way vectors containing 16-bit elements, and its output is a single 8-way vector of 32-bit elements.

In this example, the second argument to each `pmadd.wd` instruction in the unoptimized code is a vector of alternating zeroes and ones, which has the effect of selecting odd-indexed elements into `%0` and even-indexed elements into `%1`. Then, after the `sub` instruction, which simply performs element-wise subtraction of `%0` and `%1`, the overall effect of this code is to compute the difference between adjacent pairs of elements of `%x`. Minotaur is able to perform this same computation using a single `pmadd.wd` instruction which negates odd-numbered elements of `%x` before performing the addition. The optimized code requires 5 uOps to execute whereas the original code requires 8.

[5] Minotaur uses LLVM 18.1.0 for all results in this paper.

Example 3. This code is from libYUV:

```
%0 = shufflevector <32 x i8> %x, poison, <3, 7, 11, 15, 19, 23, 27, 31>
%1 = lshr %0, <6, 6, 6, 6, 6, 6, 6, 6>
%2 = zext 8 x i8> %1 to <8 x i32>
ret <8 x i32> %2
  =>
%0 = bitcast <32 x i8> %x to <8 x i32>
%1 = call @llvm.x86.avx2.psrli.d(<8 x i32> %0, 30)
ret <8 x i32> %1
```

The `shufflevector` instruction in the unoptimized code selects every fourth byte-sized element from the input `%x`. The resulting 8-way vector is right-shifted element-wise by six bit positions, and that result is zero-extended to an 8-way vector of 32-bit elements. Minotaur's optimized version (which executes in 4 uOps instead of 11) first reinterprets the input vector's data as 32-bit elements; this bitcast is relevant to LLVM's type system, but it is a nop at the CPU level. Then, the `prsli` instruction shifts each 32-bit element to the right by 30 bit positions. This right-shift-by-30 achieves the same effect as the unoptimized code, where the `shufflevector` can be seen as a right-shift-by-24, followed by an explicit right-shift-by-6.

Example 4. This code, from compiling perlbench from SPEC CPU 2017, illustrates Minotaur's ability to reason about control flow:

```
entry:
  br i1 %c, label %body, label %if.end
body:
  br label %if.end
if.end:
  %p1 = phi [ %a, %body ], [ %b, %entry ]
  %p2 = phi [ %b, %body ], [ %a, %entry ]
  %r = call @llvm.x86.avx2.pavg.b(%p1, %p2)
  ret <32 x i8> %r
    =>
  %r = call @llvm.x86.avx2.pavg.b(%a, %b)
  ret <32 x i8> %r
```

The intent of the code is to compute the element-wise average of input vectors `%a` and `%b`, with a Boolean value `%c` determining the order in which the input vectors are presented to the `pavg` instruction. However, the order of arguments to this instruction does not matter, and Minotaur's version executes in 4 uOps while the original code requires 10. Note that Minotaur was not explicitly taught that `pavg` is commutative; the necessary information was inferred naturally from the formal specification.

Example 5. This is an optimization discovered by Minotaur when it was used to compile GMP:

```
%0 = lshr i64 %x, 1
%1 = and i64 %0, 0x5555555555555555
%2 = sub i64 %x, %1
%3 = lshr i64 %2, 2
%4 = and i64 %2, 0x3333333333333333
%5 = and i64 %3, 0x3333333333333333
%6 = add nuw nsw i64 %4, %3
%7 = lshr i64 %6, 4
%8 = add nuw nsw i64 %7, %6
%9 = and i64 %8, 0xf0f0f0f0f0f0f0f
```

```
ret i64 %9
  =>
%0 = bitcast i64 %x to <8 x i8>
%1 = call @llvm.ctpop(<8 x i8> %0)
%2 = bitcast <8 x i8> %1 to i64
ret i64 %2
```

The original code performs a series of bit-level manipulations on a 64-bit integer value, with the net result of performing an 8-way vectorized 8-bit popcount operation.[6] The optimized code simply calls an intrinsic function to do the popcount; it costs 13 uOps instead of the original code's 19. Although robust recognition of open-coded idioms is not the focus of our work, Minotaur does sometimes manage to achieve this.

Taking a strict view of types in the synthesis process could help prune the search space, but it would also cause us to miss optimizations that require a flexible view of types. This example illustrates the latter case: the original code contains no indication that a good optimization can be found using a vector of type <8 x i8>, and therefore a strictly type-guided synthesis procedure would miss this one.

Example 6. This code comes from 644.nab in SPEC CPU 2017:

```
%0 = fcmp oge float %x, 0.000000e+00
%1 = fneg float %x
%2 = select i1 %0, float %0, float %2
%3 = fcmp oeq float %2, 0.000000e+00
ret i1 %3
  =>
%1 = fcmp oeq float %x, 0.000000e+00
ret i1 %oeq
```

The original code computes the absolute value of a floating-point number `%x` and then checks if the result is zero. Minotaur found that that the original code is equivalent to simply checking if `%x` is zero.

Example 7. This code comes from 619.lbm in SPEC CPU 2017:

```
%0 = fmul float %x, 0x3FF0CCCCC0000000
%1 = fcmp olt float %t1, 0x3FE20418A0000000
ret i1 %1
  =>
%0 = fcmp ole float %x, 0x3FE12878E0000000
ret i1 %0
```

The original code multiplies a floating-point value `%x` by a constant, and then checks if the result is less than another constant. Minotaur found that this code is equivalent to checking if `%x` is less than or equal to a third constant. This example shows that Minotaur can reason about and synthesize floating point literals.

Example 8. This code comes from 638.imagick in SPEC CPU 2017:

```
%0 = fmul float %x, 0.000000e+00
%1 = fmul float %0, 3.000000e+00
ret float %1
  =>
%0 = fmul float %x, 0.000000e+00
ret i1 %0
```

[6]The popcount, or Hamming weight, of a bitvector is the number of "1" bits in it.

The original code multiplies a floating-point value %x by zero, and then multiplies the result by 3.0. Minotaur found that this code is equivalent to multiplying %x by zero directly. Note the original code cannot be optimized to 0.0 directly, because of the NaN and signed zero propagation rules in floating-point arithmetic. This example shows that Minotaur is able to reason about these corner cases and synthesize the correct code.

Example 9. This code comes from FlexC's benchmark suite:

```
%0 = extractelement <4 x ptr> %0, i32 0
%1 = extractelement <4 x ptr> %0, i32 3
%2 = load i32, ptr %0
%3 = load i32, ptr %1
%4 = insertelement <4 x i32> zeroinitializer, i32 %2, i32 0
%5 = insertelement <4 x i32> %4, i32 %3, i32 3
ret <4 x i32> %5
  =>
%0 = call @llvm.masked.gather(%0, 4, <true, false, false, true>, zeroinitializer)
ret <4 x i32> %0
```

The original code extracts two pointers from a vector of pointers, loads the values from these pointers, and then inserts these values into a vector of integers. Minotaur found that this code is equivalent to performing a masked gather operation, which loads values from memory using a vector of pointers and a mask.

6 Related Work

A *superoptimizer* is a program optimizer that meaningfully relies on search to generate better code, in contrast with traditional compilers that attempt a fixed (but perhaps very large) sequence of transformations. The eponymous superoptimizer [23] exhaustively generated machine instruction sequences, using various strategies to prune the search space, and using testing to weed out infeasible candidates. Also predating modern solver-based methods, Davidson and Fraser [8] constructed peephole optimizations from machine description files. In contrast, modern superoptimizers rely on solvers to perform automated reasoning about program semantics.

Souper [27] is a synthesizing superoptimizer that works on LLVM IR; it is the most directly connected previous work to Minotaur. Souper's slicing strategy is similar to Minotaur's in that it extracts a DAG of LLVM instructions that overapproximates how a given SSA value is computed. However, unlike Souper, Minotaur extracts memory operations and multiple basic blocks, so it is capable of (we believe) strictly more transformations than Souper is able to perform. Additionally, Souper's undefined behavior model does not capture all of the subtleties of undefined behavior in LLVM, whereas we reuse Alive2's model, which is the most widely used formalization of these semantics, and the one that is most widely recognized as being correct. Finally, Minotaur focuses on vector-related transformations, whereas Souper supports neither LLVM's portable vector instruction set nor its platform-specific intrinsics. It is worth noting that, over the years, the LLVM developers have implemented numerous optimizations discovered by Souper. These are all, of course, present in LLVM 18, the compiler that is the baseline for our experimental evaluation. In other words, Minotaur is an effective superoptimizer on top of a previous solver-based superoptimizer (and Souper was effective on top of an even earlier LLVM superoptimizer [26]).

Minotaur is also strongly inspired by Bansal and Aiken's work [4]; their superoptimizer operated on x86 assembly code and was able to make interesting use of vector instructions. Starting from unoptimized assembly produced by GCC, it was able to produce code competitive with higher optimization levels. The overall structure of this superoptimizer, where program slices are extracted, canonicalized, checked against a cache, and then optimized in the case of a cache miss, is very similar

to Minotaur, but there are many differences in the details, particularly in Minotaur's slice extractor which allows its synthesis specification to approximate the original code's effect much more closely. Another assembly superoptimizer, STOKE [28–30], is not as closely related; it is based on randomly perturbing assembly-language functions. STOKE can potentially perform transformations that Minotaur cannot, but we believe that its results are more difficult to translate into standard peephole optimizations than are Minotaur's.

Several recent projects have focused not on optimizing individual programs but rather on generating program rewrite rules. OptGen [5] finds scalar peephole optimizations that meet a specified syntactic form. Even at small rewrite sizes, it was able to find numerous optimizations that were missing from the 2015 versions of GCC and LLVM. VeGen [6] generates SLP vectorization rules—an SLP vectorizer [16] merges a set of scalar operations into vector instructions. VeGen parses the Intel Intrinsics Guide [14] and uses this to build pattern matchers for x86 vector instructions. VeGen applies the pattern matchers to an input scalar program, and replaces scalar expressions with vector instructions when it finds a profitable match. VeGen uses syntactic pattern matching rather than solver-based equivalence/refinement checking. Diospyros [35] is another vector rewrite rule generator, it takes an equality saturation [32] approach and uses a translation validator to reject unsuitable candidates. As an equality saturation-based tool, Diospyros builds its search space with existing rewrite rules.

Program synthesis—generating implementations that conform to a given specification—is intimately related to superoptimization. Rake [1] performs instruction selection for vectorized Halide [25] expressions using a two stage synthesis algorithm. First, Rake synthesizes a data-movement-free sketch [31], and then in the second stage it concretizes data movement for the sketch via another synthesis query. Rake targets Hexagon DSP processors [33] which share some functionally similar SIMD instructions with x86. Cowan et al. [7] synthesized quantized machine learning kernels. Their work introduces two sketches: a compute sketch, which computes a matrix multiplication, and a reduction sketch that collects the computation result to the correct registers. It relies on Rosette [34] to generate an efficient NEON [3] implementation that satisfies the specifications for those two sketches. Swizzle Inventor [24] is another tool built on Rosette; it synthesizes data movement instructions for a GPU compute kernel, and it requires user-defined sketches describing the non-swizzle part of the program. MACVETH [12] generates high-performance vector packings of regular strided-access loops, by searching for a SIMD expression that is equivalent to a gather specification. All of these works show good performance results, but they focus on relatively narrow tasks, whereas Minotaur attempts to improve SIMD programs in general.

Most previous superoptimizers and program synthesizers use simple cost models. For example, Souper [27] assigns each kind of instruction a weight and uses the weighted sum as the cost of a rewrite. This kind of cost model is not a very good predictor of performance on a modern out-of-order processor. Minotaur and MACVETH [12] use the LLVM-MCA [10] microarchitectural performance analyzer, which can still lead to mispredictions, but it is generally more accurate than simple approaches are.

7 Conclusion

We created Minotaur because we noticed that LLVM appeared to be missing relatively obvious optimizations in code containing both its portable vector instructions and also its platform-specific intrinsic functions that provide direct access to hardware-level primitives. Minotaur cuts loop-free DAGs of instructions—including branches and memory operations—out of LLVM functions and then attempts to synthesize better implementations for them. When improved code is found, the optimization is performed and also the synthesis result is cached. On the libYUV test suite, Minotaur gives speedups up to 1.64x, with an average speedup of 2.2%.

Acknowledgments

The authors are indebted to Nuno P. Lopes for creating Alive2, for structuring it in such a way that we could programmatically access its functionality, and for helping us to use it effectively. Alexander Brauckmann and Michael O'Boyle generously provided access to a collection of loop kernels extracted using their tool, FlexC. Alastair Reid, Fabian Giesen, Sam Elliott, Raimondas Sasnauskas, Pavel Panchekha, Manasij Mukherjee, Tanmay Tirpankar, and anonymous reviewers all provided invaluable feedback on drafts of this paper. This material is based upon work supported by the National Science Foundation under Grant No. 1955688.

References

[1] Maaz Bin Safeer Ahmad, Alexander J. Root, Andrew Adams, Shoaib Kamil, and Alvin Cheung. Vector Instruction Selection for Digital Signal Processors Using Program Synthesis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, page 1004–1016, 2022.

[2] AMD. AMD Zen 3, 2023. https://www.amd.com/en/technologies/zen-core.

[3] ARM. ARM NEON Architecture, 2023. https://developer.arm.com/Architectures/Neon.

[4] Sorav Bansal and Alex Aiken. Automatic Generation of Peephole Superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 394–403, 2006.

[5] Sebastian Buchwald. Optgen: A generator for local optimizations. In *International Conference on Compiler Construction*, pages 171–189. Springer, 2015.

[6] Yishen Chen, Charith Mendis, Michael Carbin, and Saman Amarasinghe. *VeGen: A Vectorizer Generator for SIMD and Beyond*, page 902–914. 2021.

[7] Meghan Cowan, Thierry Moreau, Tianqi Chen, James Bornholt, and Luis Ceze. Automatic Generation of High-Performance Quantized Machine Learning Kernels. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, page 305–316, 2020.

[8] Jack W. Davidson and Christopher W. Fraser. Automatic Generation of Peephole Optimizations. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '84, pages 111–116, 1984.

[9] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*, pages 337–340. Springer, 2008.

[10] LLVM Developers. LLVM Machine Code Analyzer, 2023. https://llvm.org/docs/CommandGuide/llvm-mca.html.

[11] Bruno Dutertre. Solving exists/forall problems with yices. In *The 13th International Workshop on Satisfiability Modulo Theories*, 2015.

[12] Marcos Horro, Louis-Noël Pouchet, Gabriel Rodríguez, and Juan Tourino. Custom High-Performance Vector Code Generation for Data-Specific Sparse Computations. In *Proceedings of the 31st International Conference on Parallel Architectures and Compilation Techniques*, 2022.

[13] Intel. Cascade Lake: Overview, 2023. https://www.intel.com/content/www/us/en/products/platforms/details/cascade-lake.html.

[14] Intel. Intel Intrinsics Guide, 2023. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[15] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: a goal-directed superoptimizer. *SIGPLAN Notices*, 37(5):304–314, May 2002.

[16] Samuel Larsen and Saman Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, page 145–156, 2000.

[17] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[18] LLVM Developers. LoopInfo, 2023. https://llvm.org/doxygen/classllvm_1_1LoopInfo.html.

[19] LLVM Developers. MemorySSA, 2023. https://llvm.org/docs/MemorySSA.html.

[20] LLVM Developers. ORC Design and Implementation, 2023. https://llvm.org/docs/ORCv2.html.

[21] LLVM Developers. TargetTransformInfo Class Reference, 2023. https://llvm.org/doxygen/classllvm_1_1TargetTransformInfo.html.

[22] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. *Alive2: Bounded Translation Validation for LLVM*, page 65–79. 2021.

[23] Henry Massalin. Superoptimizer: A look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, page 122–126, 1987.
[24] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. Swizzle Inventor: Data Movement Synthesis for GPU Kernels. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 65–78, 2019.
[25] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Halide: Decoupling Algorithms from Schedules for High-Performance Image Processing. *Commun. ACM*, 61(1):106–115, dec 2017.
[26] Duncan Sands. Super-optimizing LLVM IR, November 2011. Presentation at the 2011 LLVM Developers' Meeting.
[27] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. Souper: A synthesizing superoptimizer. *arXiv preprint arXiv:1711.04422*, 2017.
[28] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic Superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 305–316, 2013.
[29] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic optimization of floating-point programs with tunable precision. *ACM SIGPLAN Notices*, 49(6):53–64, 2014.
[30] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. Conditionally correct superoptimization. *ACM SIGPLAN Notices*, 50(10):147–162, 2015.
[31] Armando Solar-Lezama. The Sketching Approach to Program Synthesis. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, page 4–13, Berlin, Heidelberg, 2009. Springer-Verlag.
[32] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, page 264–276, 2009.
[33] Qualcomm Technologies. Hexagon dsp sdk. https://developer.qualcomm.com/software/hexagon-dsp-sdk.
[34] Emina Torlak and Rastislav Bodik. Growing Solver-Aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, page 135–152, 2013.
[35] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 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*, ASPLOS '21, page 874–886, 2021.
[36] Jackson Woodruff, Thomas Koehler, Alexander Brauckmann, Chris Cummins, Sam Ainsworth, and Michael FP O'Boyle. Rewriting history: Repurposing domain-specific cgras. *arXiv preprint arXiv:2309.09112*, 2023.

Received 2024-04-03; accepted 2024-08-18