# Copy-and-Patch Binary Code Generation

Haoran Xu
haoranxu@stanford.edu
Stanford University
USA

Fredrik Kjolstad
kjolstad@stanford.edu
Stanford University
USA

## Abstract

Runtime compilation of runtime-constructed code is becoming standard practice in libraries, DSLs, and database management systems. Since compilation is expensive, systems that are sensitive to compile times such as relational database query compilers compile only hot code and interprets the rest with a much slower interpreter.

We present a code generation technique that lowers an AST to binary code by stitching together code from a large library of binary AST node implementations. We call the implementations stencils because they have holes where values must be inserted during code generation. We show how to construct such a stencil library and describe the copy-and-patch technique that generates optimized binary code.

The result is a code generator with negligible cost: it produces code from an AST in less time than it takes to construct the AST. Compared to LLVM, compilation is two orders of magnitude faster than -O0 and three orders of magnitude faster than higher optimization levels. The generated code runs an order of magnitude faster than interpretation and runs even faster than LLVM -O0. Thus, copy-and-patch can effectively replace both interpreters and LLVM -O0, making code generation more effective in compile-time sensitive applications.

## 1 Introduction

Modern applications such as machine learning, image processing, and data processing require vast amounts of expensive computation. The cost is reflected in the operating budgets of data centers, the battery lifetime of mobile phones, and the response time of interactive database applications. But two trends have made it harder to write efficient software libraries. Many libraries are becoming small languages whose programs are often not known until runtime [16, 26] and hardware is becoming more diverse [17, 47]. These trends increase the need for runtime systems that customize computation to their environment. Thus, it is desirable to use metaprogramming techniques to optimize the performance of libraries and systems.

The development of the LLVM compiler library [22] made it practical to write portable libraries and systems that generate code at runtime. As a consequence, we have seen libraries and systems that use LLVM to compile database queries [31],

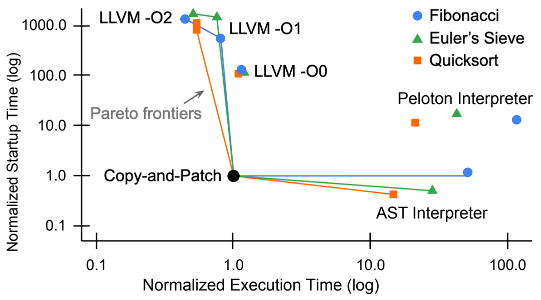


**Figure 1.** Normalized startup and execution times of several alternatives over three microbenchmarks. The Pareto frontiers show that copy-and-patch replaces -O0. It also replaces interpreters since startup time is negligible for both.

image processing pipelines [42], sparse tensor algebra [19], and deep neural networks [5]. LLVM provides a complete compiler infrastructure with many optimizations, such as register allocation, instruction selection, and many IR rewrite passes. The optimizations are grouped into optimization levels, -O0, -O1, -O2, and -O3, that provide progressively better performance at progressively higher compilation cost.

Thus, there is a trade-off between preprocessing time and execution time: the more you prepare the faster you go. The trade-off lies in the choice among LLVM optimization levels and an interpreter. The Pareto frontiers of these options for three microbenchmarks are shown as dashed lines in Figure 1. Each point in the figure plots log-scaled execution time on the x-axis against log-scaled startup time on the y-axis. The lines represent benchmarks, and each point along the line represents a different optimization level, in descending order from left to right, followed by an AST interpreter and the Peloton interpreter [21, 37]. The library or system developers choose among these points, or develops a hybrid approach, depending on their expectation for the compilation and running times of their code.

The cost of LLVM compilation is an obstacle when execution times do not reliably dominate compilation times. An example is database query processing. In the last decade, in-memory database management systems have optimized CPU-bounded workloads by using LLVM to generate code. Examples including Hyper [31], Peloton [30], PostgreSQL [40], and MemSQL [27]. But LLVM compilation is expensive compared to the execution time of many queries. PostgresSQL therefore estimates query cost and interprets queries if costs are low.

For more expensive queries, it uses LLVM `-O0`. LLVM optimization is only enabled when query cost is very high [39]. MemSQL uses an interpreter for the first execution of a query, while it compiles in the background in case it is needed again [28]. It is in general not possible to predict whether a query will be used once or many times, since queries are often generated by applications or provided by users. The compilation may therefore be wasted. To mitigate these complications and provide better response time for one-off queries, we want a technique that offers the execution performance of LLVM at the startup cost of an interpreter.

Our solution is the copy-and-patch technique. Copy-and-Patch produces binary implementations from abstract syntax trees (AST) by copying stencils—binary fragments with missing values—into contiguous memory, followed by a pass to patch missing values into the binary code. The stencils come from a pre-built stencil library, and the AST may express imperative languages, functional languages, and domain-specific languages. We find copy-and-patch in Figure 1 as a black point. It dominates LLVM `-O0` on the Pareto frontier, decreasing compilation cost by two orders of magnitude. Furthermore, its compilation overhead is low enough to be negligible, effectively replacing interpretation for all but the smallest functions. The negligible compilation cost comes from the stencil library that lets us push costly compilation work—optimization, register allocation, and instruction selection—to library installation time. Our contributions are:

1. We introduce copy-and-patch, an algorithm with negligible running time that lowers high-level languages to binary code that outperforms LLVM `-O0`.
2. We describe the copy-and-patch compiler for a C-like language, and the MetaVar language for describing and compiling stencil generators. The implementation is in Appendix A of the supplemental material.
3. We show how to leverage Clang+LLVM to generate the library of binary stencils.

We evaluate these contributions on microbenchmarks and TPC-H database queries [46]. Our results show that copy-and-patch-generated code outperforms interpreters by an order of magnitude, consistently outperforms LLVM `-O0` with an average 15% speedup, and on average reaches 74% of the performance of LLVM `-O3`. Meanwhile, compile times are negligible at less than the cost of creating the AST, two orders of magnitudes faster than LLVM `-O0`, and three orders of magnitudes faster than LLVM `-O1` or higher. The execution performance is less than LLVM `-O3` because each stencil is independently optimized at build time, blocking global optimization. It is an order of magnitude faster than interpreters, and faster than LLVM `-O0`, because it makes indirect jumps, branches, and calls direct, removes unnecessary jumps, writes literals into instructions, does register allocation across AST nodes, and uses super-node stencils that implement multiple AST nodes.

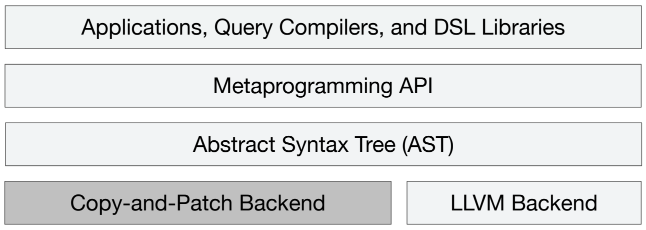

Applications, Query Compilers, and DSL Libraries
Metaprogramming API
Abstract Syntax Tree (AST)
Copy-and-Patch Backend
LLVM Backend

**Figure 2.** Copy-and-Patch can be used by metaprogramming systems to speed up their interpreters past LLVM `-O0`. And it can be combined with LLVM to let the user request higher optimization levels when the increased performance can amortize the orders of magnitude higher compilation cost.

## 2 Overview

Since the copy-and-patch algorithm combines negligible code generation overhead with efficient generated code, we believe that it can replace interpreters in metaprogramming systems. Figure 2 shows the role of copy-and-patch in a metaprogramming system that generates code based on runtime values, such as database queries and domain-specific languages like tensor expressions. The user of this metaprogramming system can either use copy-and-patch or compile with LLVM `-O1` or higher at a much higher startup cost.

We have developed a system that implements the copy-and-patch code generation technique. Its input is a set of AST node types together with a stencil generator for every type of AST node. Each stencil generator generates many stencil variants for its AST node type (about 4000 on average). A stencil is a binary implementation of a node or several nodes, specialized with types, storage location of operands (registers or stack), and a specific shape. We provide a standard library of AST node types that implement a simple imperative language with some object-orientation. It supports all C language constructs, local C++ objects, the ability to call external C++ functions, and C++ exceptions. Furthermore, users can expand the library with their own AST nodes, which is useful when a construct can not be implemented in the language and an external function call is too slow. For example, we provide an addition expression with C overflow semantics. But a user implementing a database query compiler might need an addition expression with SQL overflow semantics, implemented with low-level compiler intrinsics. Although the techniques we describe stand on their own, we believe our implementation can be used directly by metaprogramming systems such as Julia [2], Halide [42], TACO [19], Hyper [31], Peloton [37], and Terra [7].

The copy-and-patch system consists of two components: the MetaVar compiler and the copy-and-patch code generator, and Figure 3 shows their relationship. The MetaVar compiler takes as input the AST stencil generators and produces a library of binary stencils at library installation time. The stencil library becomes an input to the copy-and-patch code

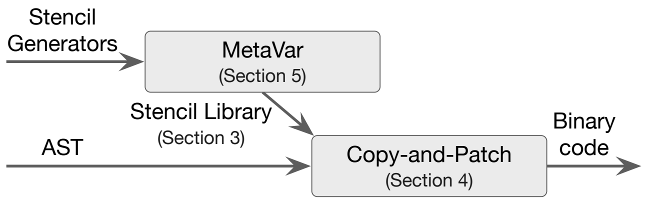


**Figure 3.** The copy-and-patch system compiles an AST to binary code. It consists of the MetaVar compiler, which compiles stencil generators to binary stencils, and the Copy-and-Patch code generator, which generates a binary implementation of an AST by copying and patching stencils.

generator, together with the runtime-constructed AST that implements a function. The code generator then produces binary code that implements the AST, by copying and patching together stencils that implement the AST nodes. The patching step rewrites pre-determined places in the binary code, which are operands of machine instructions, including jump addresses and values of constants (stack offsets and literal values). Despite patching binary code, however, the system does not need any knowledge of platform-specific machine instruction encoding and is thus portable.

There are many stencil variants of each AST node, and the copy-and-patch code generator optimizes the code by selecting variants. For example, if the AST contains an addition with a constant, then the copy-and-patch code generator will choose a variant that adds to a literal and then patch in the literal value. It will also perform register allocation by choosing among stencils that operate on values in registers and ones that operate on values on the stack, depending on register availability. And as a final example, the copy-and-patch algorithm will as far as possible place stencils of operators that follow each other in consecutive locations in memory, which allows it to remove the jump between them. Together, these and other optimizations by stencil variant selection produces code that is far superior to the performance of an interpreter and even faster than LLVM `-O0`.

## 3 The Stencil Library

The stencil library contains binary implementations of AST node types that are stitched together at runtime by the copy-and-patch algorithm to generate code for an AST. We call the binary implementations of AST nodes stencils, because they have holes where copy-and-patch inserts missing values to specialize them for the specific runtime AST. The stencil library contains many stencil variants for each AST node type that are specialized for different operand types, value locations, and more. The variations allows copy-and-patch optimize the generated code and do simple register allocation. Since the configuration options compose as a Cartesian product, the stencil library can grow large and our implementation contains 98,831 stencils (17.5 MB of binary code). It is therefore impractical to hand-write every stencil.

A binary stencil is a binary code function that implements an AST node type, where literals, jump addresses, and stack offsets are missing. During copy-and-patch code generation, as we describe in Section 4, the stencils are copied and the missing values inserted. For example, if an instruction uses a literal then the value is filled in, and if it has a branch instruction to the next operation then the address is inserted.

The binary stencils use continuation-passing style (CPS) [43] to pass control to the next stencil. With continuation-passing, control is passed directly to the next operation instead of being returned to the parent operation. Figure 4 shows how the resulting continuation-passing control flow moves bottom-up through an expression. Since function calls to pass on control are tail calls, the Clang C++ compiler that the MetaVar system uses to pre-compile stencils lowers them to jump instructions. Combined with GHC calling convention [44], in which all registers are saved by the caller and all parameters are passed in registers, continuation-passing removes most of the calling overhead between stencils.

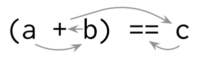


**Figure 4.** CPS

To sum up, the stencils come in many variants per AST node type, use continuation-passing style, and leave missing values to be filled in by the copy-and-patch algorithm. Figure 5 shows conceptual C++ code for four stencils, replacing the missing values with blue numbered boxes. We show the implementations of these boxes in Figure 8. The C++ stencils are compiled by MetaVar at installation time to produce binary stencils, as described in Section 5.2. Figures 5a–5c show three of the stencils that implement equality expressions. Figure 5a takes the expression's operands as arguments, compares them, and passes control and context to the next stencil by calling a continuation function. The blue box is the missing address to the next stencil. Missing addresses, and other types of missing values, are filled in when copy-and-patch generates code for an expression at runtime. Since MetaVar compiles the stencils with the GHC calling convention, the binary code assumes arguments are in registers. Thus, this stencil compares two integers whose values are stored in registers. Figure 5b is an equality variant where the first operand has been spilled to the stack, while the second operand is a literal. It has three missing values: the stack offset, the literal value, and the continuation address. And Figure 5d shows the stencil for an if-then-else, taking the result of the test as an argument and calling one of two continuation functions.

The copy-and-patch algorithm composes stencils to implement an AST. It attempts to keep values in registers by selecting variants that work on arguments, like the one in Figure 5a. If a value is stored in a register and is needed by a later operation, then copy-and-patch chooses stencils variants that pass these values through, like the one in Figure 5c. The pass-through stencil, by inserting arguments passed through to the continuation, forces Clang to ensure registers

```
void eq_int(uintptr_t stack, int lhs, int rhs) {
  bool result = (lhs == rhs);
  (void(*)(uintptr_t, bool) [1] )(stack, result);
}
```

**(a)** Equality test of two integer values in registers. Only the continuation-passing call to the next code fragment is missing.

```
void eq_int_lvar_rconst(uintptr_t stack) {
  int lhs = *(int*)(stack + [1]);
  int rhs = [2];
  bool result = (lhs == rhs);
  (void(*)(uintptr_t, bool) [3] )(stack, result);
}
```

**(b)** Equality test of an integer value on the stack to a constant. The first value's stack offset and the integer constant are missing.

```
void eq_int_pt(uintptr_t stack, uint64_t r1, int rhs) {
  int lhs = [1];
  bool result = (lhs == rhs);
  (void(*)(uintptr_t,uint64_t,bool) [2] )(stack, r1, result);
}
```

**(c)** Equality test that passes through a value computed by a prior operation and stored in a register for use by a later operation.

```
void if(uintptr_t stack, bool test) {
  if (test)
    (void(*)(uintptr_t) [1] )(stack);
  else
    (void(*)(uintptr_t) [2] )(stack);
}
```

**(d)** General if-then-else statement. The calls to the stencils of the then and else statements are missing.

**Figure 5.** The conceptual logic of four out of 98,831 stencils in the stencil library, generated by template instantiation.

values at function entry are unchanged when the continuation is called. This encourages Clang to use other available registers in the function body. When available registers are insufficient to hold a temporary value for its lifetime, the copy-and-patch algorithm composes a variant that spills it to the stack with a variant that uses the spilled value.

The MetaVar system compiles C++ stencils to binary stencils that contain binary code and information about where the missing values are located. Figure 6 shows the `Stencil` struct that stores a stencil. The `binaryCode` array contains the binary code, followed by three arrays that store the locations of the stencil's missing values, so that they can be patched in by the patching phase of copy-and-patch.

The stencil library maps stencil configurations that identify each stencil to the stencils themselves:

$$\langle\text{configuration}\rangle \longrightarrow \langle\text{stencil}\rangle.$$

The configurations contain what AST node type the stencil implements, what types it operates on, whether it operates on constants, registers, or stack locations, and so forth. The stencil library is generated by the MetaVar system at installation time, as described in Section 5.2 and contains 98,831

```
struct PatchRecord {
  uint32_t binaryOffset;
  uint32_t ord;   // ordinal of the missing value
};
struct Stencil {
  std::vector<uint8_t>     binary;
  std::vector<uint32_t>    pc32Patches;
  std::vector<PatchRecord> symbol32Patches;
  std::vector<PatchRecord> symbol64Patches;
};
// patches[i] is desired value for missing value ordinal i
void Stencil::copyPatch(uintptr_t dst, uint64_t* patches) {
  memcpy((void*)dst, binary.data(), binary.size());
  for (auto binaryOffset : pc32Patches)
    *(uint32_t*)(dst + binaryOffset) -= dst;
  for (auto p : symbol32Patches)
    *(uint32_t*)(dst + p.binaryOffset) += patches[p.ord];
  for (auto p : symbol64Patches)
    *(uint64_t*)(dst + p.binaryOffset) += patches[p.ord];
}
```

**Figure 6.** The data structure that stores binary stencils, and the method that materializes a stencil at a given address. The struct includes the binary code and the locations of missing values to patch in during copy-and-patch code generation.

stencils. The library is then used by the copy-and-patch algorithm in Section 4, which weaves together the binary stencils for the AST nodes to create the generated function.

## 4 Copy-and-Patch Code Generation

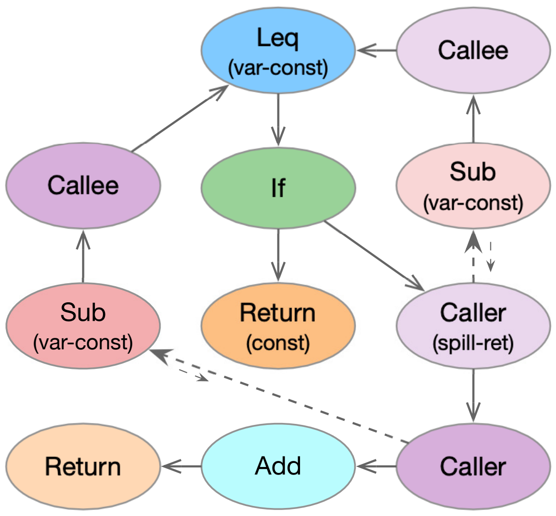
The copy-and-patch binary code generation algorithm lowers an AST that describes a high-level language to binary code. It executes at runtime and produces code performing better than LLVM `-O0` at negligible cost. In most cases, compilation time is less than the time to construct the AST. The algorithm performs two post-order traversals of the AST: once to plan register usage, and once to select stencil configurations for AST nodes and construct a compact continuation-passing style (CPS) call graph. Next, the algorithm traverses the call graph depth-first. At each node, it copies the binary code of the node's stencil into the memory region immediately after the previously copied stencil. Finally, it patches the missing values into the stencil's binary code, including literal values used in the AST, stack offsets for local variables and spilled temporaries, and branch, jump, or call addresses to other stencils.

An AST is lowered to binary code through a CPS call graph. Figure 7 shows the representations that the Fibonacci function goes through on its way from the AST shown as printed code in Figure 7a, through the CPS call graph in Figure 7b, to machine code shown in assembly form in Figure 7c. In each representation, the code that corresponds to each AST node is color-coded the same way, so that each node can be followed individually through the stages. We will use this example in our descriptions of each stage.

The copy-and-patch algorithm does light-weight register allocation to keep temporary values in registers to minimize

```
If(n <= 2).Then(
  Return(1)
).Else(
  Return(Call<FibFn>("fib", n-1)
       + Call<FibFn>("fib", n-2))
)
```

**(a)** Fibonacci AST printed as text.



**(b)** CPS call graph between stencils. Solid arrows are tail calls compiled to jump instructions, while dotted lines require call instructions.

```
00:  mov    0x8(%r13),%r12d
07:  mov    $0x2,%eax
0c:  sub    %eax,%r12d
0f:  mov    %r12d,0x8(%rbp)
13:  mov    %rbp,%r13
20:  mov    $0x2,%eax          ← fib function entry
25:  cmp    %eax,0x8(%r13)
2c:  jg     40
32:  movabs $0x1,%rbp
3c:  mov    %rbp,%rax
3f:  retq
40:  sub    $0x38,%rsp
44:  mov    %r13,0x8(%rsp)
49:  lea    0x10(%rsp),%rbp
4e:  callq  90
53:  mov    0x8(%rsp),%r13
58:  mov    %rax,0x10(%r13)    ← only spilled value
5f:  add    $0x38,%rsp
63:  sub    $0x38,%rsp
67:  mov    %r13,0x8(%rsp)
6c:  lea    0x10(%rsp),%rbp
71:  callq  00
76:  mov    0x8(%rsp),%r13
7b:  mov    %rax,%rbp
7e:  add    $0x38,%rsp         ← jumps between
82:  add    0x10(%r13),%rbp      consecutive code
89:  mov    %rbp,%rax            blocks are removed
8c:  retq
90:  mov    0x8(%r13),%r12d
97:  mov    $0x1,%eax
9c:  sub    %eax,%r12d
9f:  mov    %r12d,0x8(%rbp)
a3:  mov    %rbp,%r13
a6:  jmpq   20
```

**(c)** Assembly code generated by copying the call graph node stencils and patching in missing values (light blue).

**Figure 7.** The Fibonacci function's AST, its CPS call graph, and the assembly of the binary code generated by copying and patching each CPS call graph node. Each AST node or super-node identified by C&P has the same color in each representation.

the number of spills. Register allocation is done in a post-order traversal of the AST. The traversal maintains the stack of outstanding temporary operands and, at each step, marks everything below the maximum number of register water-mark to be spilled. Temporary values that cross a subsequent call must also be spilled because the GHC calling convention used for stencils assigns all registers to the callee. Figure 7c shows the effectiveness of our simple register allocation: no temporary values were spilled except the result of the first function call on line 58, since its lifetime crosses the second function call.

The first lowering step converts the AST to a CPS call graph, by selecting stencils that implement the AST nodes and linking them in the order they will be executed. The CPS call graph is constructed in the second post-order traversal of the AST. For each node, the algorithm selects the most specific stencil configuration. It also does a simple tree pattern matching to find sub-trees that can be implemented with an efficient node or super-node stencil. Super-nodes allow Clang to optimize larger regions of code, e.g., leverage advanced assembly instructions supported by the target architecture. For example, we provide super-nodes for array accesses indexed by a constant or local variable, which can be turned into fewer instructions using advanced x86-64 addressing modes. The user can also extend this system to insert new super-nodes, such as fused multiply-add. When a stencil is selected for one or several nodes, the stencil's configuration is added to the CPS call graph and a call edge is set to point to the next node in the post-order traversal. Figure 7b shows the call graph for Fibonacci. Most calls are tail calls (solid arrows), which the stencil compiler turns into jump instructions. Only two true calls are left: the two call expressions in the AST. Finally, note that the CPS construction algorithm chose specialized version for the Sub expression that operates on a variable and a constant, thus improving the quality of the resulting code.

The second lowering step converts the CPS call graph to binary code by copying the binary stencil code of each node to contiguous memory. The copy step traverses the CPS call graph in depth-first order starting at any node that has no predecessors. At each call graph node, it retrieves the stencil corresponding to the node's configuration from the stencil library. It then copies the stencil's binary code into the memory region following immediately the binary code of the stencil belonging to the preceding call node. The purpose of copying these into consecutive locations in depth-first order is to maximize the number of stencil binary codes that jump to a stencil binary code right after it. As these jumps are fruitless, the stencil copy simply elides them from the copy. Figure 7c shows the binary code in assembly form resulting from the Fibonacci function. Most jumps were successfully elided (e.g., line 82), while only two jumps could not be removed (line 2c and a6). If a stencil node has a fixed predecessor, and the predecessor is not a conditional branch, then our algorithm is guaranteed to elide the jump instruction. Therefore, all remaining jump instructions must correspond to some form of control-flow redirection statement (e.g., if-branches, loops, calls) in the input AST, and are thus necessary.

The final step patches missing values into the copied binary code. For each stencil, it iterates through missing values to insert literal values from the AST, stack offsets for variables and temporaries, and branch, jump, or call targets to other stencils (for jumps that were not elided). Figure 6 contains the stencil struct that stores information about missing values, and the logic to patch them. Figure 7c shows the filled in missing values in blue. For instance, the jump target on line a6 jumps to the Leq stencil on line 20, and the value 0x2 on line 20 is the literal 2 from the Leq AST node in Figure 7a.

The copy-and-patch technique also supports external function calls. External functions are important in database applications like the TPC-H queries in Section 6.3, where data is stored in C++ data structures that must be iterated and accessed from generated code. The external call node expects the callee to take a single `void*` parameter, pointing to an array with the actual parameters. Template metaprogramming techniques can be used to automatically wrap any C++ function into this form, including functions with non-primitive parameters passed by value, overloaded functions, and method calls. In fact, the metaprogramming system we built on top of copy-and-patch supports calling *any* C++ function in a type-safe manner. Code generated by copy-and-patch can also propagate C++ exceptions thrown by functions it calls. The wrapper function catches any thrown exception and stores it to a thread-local variable. The copy-and-patch external call node has a boolean return value that the wrapper functions use to signal that an exception was thrown. The return value is checked by generated code, and if true, branches to code that calls destructors, propagates it through the generated function call stack, and returns it to the calling host code that must re-throw the exception. Through these features, the copy-and-patch-generated code efficiently interoperates with the host language such as C++, allowing it to call host code and to manage the host code's exceptions. The description of a complete metaprogramming system built upon the C&P technique is, however, outside the scope of this paper.

## 5 Stencil Library Construction

The MetaVar compiler constructs the stencil library from programmer-specified stencil generators. The programmer specifies one stencil generator for each AST node by writing C++ code that uses template meta-variables to express variants, and uses special macros to express missing values to be patched at runtime. The MetaVar compiler then iterates at compile-time over the values of the meta-variables and instantiates the template for every valid combination. The instantiated templates are compiled by the Clang C++ compiler to object code. The stencil library builder then parses the object code to retrieve the stencil configurations and binary stencils, which are used to build the stencil library that is linked to the copy-and-patch runtime.

### 5.1 Stencil Generators

Stencil generators are templated C++ functions whose template instantiations produce stencils. Their template parameters are called metavars, which are defined by a fixed set of values. For instance, the `PrimitiveType` metavar enumerates primitive types, while a boolean metavar enumerates false and true. The MetaVar compiler iterates through the values of the metavars, subject to user-defined filter template functions, to instantiate stencils at library installation time.

More precisely, let $S_1, \ldots, S_n$ be the sets that enumerate the values of each metavar used in a stencil generator, where each $S_i$ is either a finite set of types or a finite set of values. Let $L = S_1 \times \cdots \times S_n$ be their Cartesian product. Given a template filter function, $f : L \to \text{bool}$, and a template generator function $g : L \to \text{stencil}$, the MetaVar system generates a list of pairs $\langle l, g(l) \rangle$, for all $l \in L$ where $f(l)$ is true. In other words, the MetaVar system generates a list of tuples that map valid metavar configurations to C++ stencils.

Metavars and filters work together to produce valid stencils. Within stencils, missing functions and values are defined by special macros such as `DEF_CONTINUATION_0` and `DEF_CONSTANT_1`. We demonstrate these features by the simplified addition generator and filter functions in Figure 8. The generator g has three metavars: the operand type `T`, whether the result should be spilled, and the number of pass-through variables to be preserved in registers across this stencil. It produces only stencils whose operands `a` and `b` are stored in registers, while a more sophisticated generator would support these having been spilled by a previous operation. It would also handle the case where one side has a simple shape (e.g. literal, variable, or simple array indexing). The compile-time conditional inside `g` determines whether the generated stencil spills the result or keeps it in a register. If spill is false, then the generated stencil simply passes the result `c` to the continuation as an argument, which will be stored in a register in the GHC calling convention. But if spill is true, then the result `c` is instead stored to the stack.

```
template<typename T, bool spill, NumPassthroughs numPt,
         typename... Passthroughs>
void g(uintptr_t stack, Passthroughs... pt, T a, T b) {
  T c = a + b;
  if constexpr (!spill) {
    DEF_CONTINUATON_0(void(*)(uintptr_t,Passthroughs...,T));
    CONTINUATON_0(stack, pt..., c);  // continuation
  } else {
    DEF_CONSTANT_1(uint64_t);
    *(T*)(stack + CONSTANT_1) = c;
    DEF_CONTINUATON_0(void(*)(uintptr_t,Passthroughs...));
    CONTINUATON_0(stack, pt...);     // continuation
  }
}

template<typename T, bool spill, NumPassthroughs numPt>
constexpr bool f() {
  if (numPt > numMaxPassthroughs - 2) return false;
  return !std::is_same<T, void>::value;
}

auto metavars() {
  return createMetaVarList(
    typeMetaVar(),
    enumMetaVar<NumPassthroughs::X_END_OF_ENUM>(),
    boolMetaVar());
}
extern "C" void generate(StencilList* result)
{ runStencilGenerator<&metavars>(result); }
```

**Figure 8.** A simplified addition stencil generator written with C++ template metaprogramming. The generator is processed by MetaVar to generate stencils for addition.

The missing values in a stencil are defined using special macros that also assign an ordinal to each missing value. For example, `DEF_CONTINUATION_0` defines ordinal 0 to be a function of a specified type, and `DEF_CONSTANT_1(int)` defines ordinal 1 to be a constant of type `int`. At runtime, the stencil can be patched by specifying the desired value for each ordinal, as shown in Figure 6. Internally, a special macro expands to a piece of code that declares a local variable of the given function or constant type and assigns to it the address of a pre-defined extern variable. We forcefully cast extern variables to their assigned types, using `reinterpret_cast` for functions and a C `union` hack to bit-cast constants: see Appendix B in the supplemental material for the implementation. Since extern variables in C++ are by definition defined outside the current module, this forces Clang to emit information into the object code that identifies the locations of those missing values in the binary code. This information can be used to figure out how a stencil shall be patched, as elaborated in Section 5.2.

The macro-defined constants can be used just like normal constant variables, except that you cannot compare equality between two macro-defined constants, or between such a constant and 0. The reason for these limitations is that two different extern symbols are never equal, and that symbols are never null. Additionally, in x86-64 architecture, MetaVar must compile the stencils using the suitable code model [25]. Nevertheless, these limitations are easy to work around.

Finally, the pass-through arguments let the MetaVar system generate variants that avoid clobbering registers that the copy-and-patch algorithm uses for the results of other operations. For example, when computing $\text{term}_1+\text{term}_2$, the result of the first term must be stored while computing the second term. In this case, the copy-and-patch algorithm would choose a stencil variant with one pass-through variable to protect the register that stores this result. The example in Figure 8 shows how pass-through template variables are used in a generator: they are passed from the arguments of the generator to its continuations. For ease of exposition, we only show one set of pass-through variables. In x86-64 integer and floating-point values are passed in separate sets of registers, so our implementation needs two sets of pass-through variables for integral and floating-point values respectively.

Any number of temporaries may need to be stored to compute an expression, but a given machine only has a limited number of registers. The filter function `f`, in addition to removing the `void` type that cannot be added, also bounds the number of pass-through variables for which stencils will be generated. As we have seen in Section 4, the copy-and-patch algorithm chooses stencils to store temporary values in registers to avoid as many spills as possible, and uses pass-through stencils for subsequent operations. If a temporary cannot be kept in register for its lifetime, C&P will instead select the stencil that spill it to the stack.

## 5.2 The MetaVar Compiler

MetaVar compiles stencil generators to the stencil library, which maps stencil configurations to stencils. As described in Section 3, a stencil configuration describes a stencil, including what node type it implements and whether it spills the result, while the stencil consists of binary code and the locations of missing values. Figure 9 shows the stages of the MetaVar system as arrows, with boxes showing inputs and outputs. MetaVar leverages both C++ template metaprogramming and the Clang+LLVM compiler infrastructure to generate the binary stencils, avoiding the need to implement an optimizing compiler. The result is a concise system that can generate code for any platform that LLVM supports.

The first stage of the MetaVar system, stencil generation, converts stencil generators to stencils. Stencil generation is a C++ template program (the `runStencilGenerator` function in Figure 8) that for each stencil generator iterates over the Cartesian combination of metavar values using template recursion. For each combination of metavar values, which we call a configuration, the stencil generation checks its validity by calling the stencil generator's filter template function. It then stores all valid $\langle$configuration, function pointer$\rangle$ pairs into a list. As a side effect of taking the function pointer, all valid stencils are also instantiated by Clang and their implementations are compiled to object code.

The stencil extraction stage extracts configurations and corresponding stencils from the object code. The configurations are extracted by executing object code. As we saw

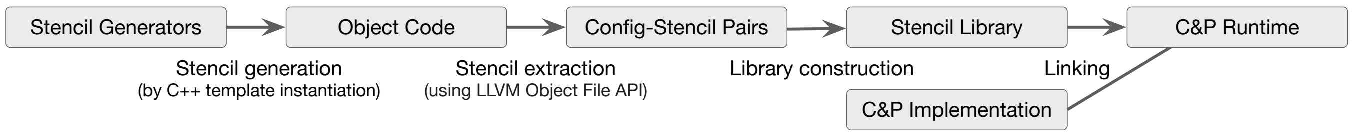


**Figure 9.** MetaVar compiles stencil generators to a stencil library that is linked to the copy-and-patch (C&P) implementation.

in Figure 8, a stencil generator contains a `generate()` function that returns a list of stencil configuration and function pointer pairs. Stencil extraction uses the LLVM JIT machinery to execute this function to retrieve the configuration. We then use an LLVM JIT API to get symbol names of the stencil functions from the function pointers in the list. The next step uses the LLVM object file parser to locate the functions based on the symbol names of the stencils we got in the previous step. It then extracts their binary code and the linker relocation records containing information about the extern symbols that were inserted by the placeholder macros, which is used to record the offsets to the missing values and how to patch them. Specifically, each missing value is a 32 or 64-bit scalar, and the patch shall be computed by initializing it with a fixed constant, optionally subtracting its memory address, and optionally adding the memory address of a symbol [25]. Although this computation rule is technically architecture-dependent, it applies to most major architectures including x86-64 [25], ARM [24], and SPARC [34]. The rule yields the 3 patch vectors and the patch algorithm in Figure 6 and, together with the binary code, give us a stencil.

The final stage constructs the stencil library from the configuration-stencil pairs. The library is a C++ file containing a static constant hash map that maps stencil configurations to the binary stencils in Figure 6. The library is linked together with the copy-and-patch implementation to form the copy-and-patch runtime.

## 6 Evaluation

We evaluate our central claim that copy-and-patch replaces both LLVM `-O0` compilation and interpreters. It generates code that executes faster than `-O0` compilation and an order of magnitude faster than interpretation, with negligible code generation overhead. Higher LLVM optimization levels are still desirable for long-running computations, but copy-and-patch narrows the range in which they are useful, as their compilation overhead is three orders of magnitude higher than copy-and-patch code generation. We also quantify the effect of the copy-and-patch optimizations, to shed light on where the performance of its generated code comes from.

### 6.1 Methodology

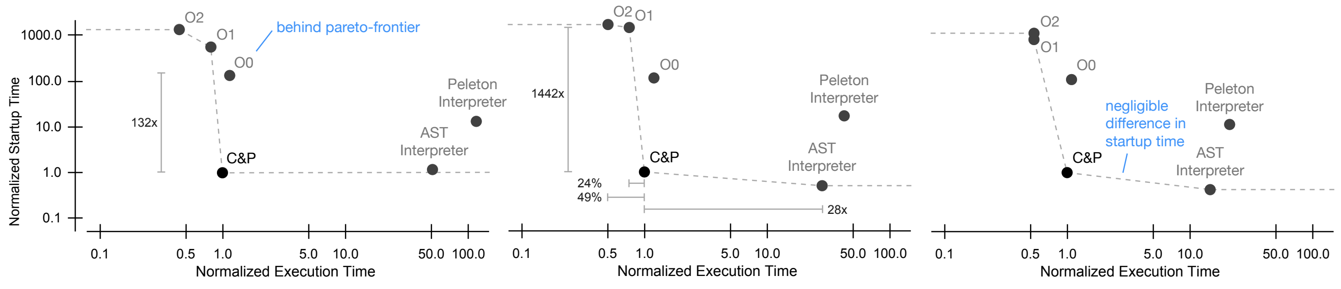
All experiments are run on a single-socket 4-core Intel i7-7700HQ CPU at 2.80GHz. We compiled the stencils and the copy-and-patch runtime using Clang++ 10 with the options `-O3 -DNDEBUG`. The MetaVar system and the LLVM backend of our metaprogramming language use LLVM library v10.0.0, the latest version at the time of writing. Since all algorithms we used are deterministic and single-threaded, we pin each benchmark to a fixed CPU and report the best execution time out of 10 runs, to reduce noise as much as possible. We measured execution times with the `clock_gettime()` Linux high resolution clock API. LLVM has an additional fixed startup cost of about $1$ ms when compiling a module. Therefore, for microbenchmarks where the modules are small, we amortized out that cost by cloning the function $100$ times and report $1/100$ of total code generation time, to reflect the true time LLVM spent generating code.

We implemented three microbenchmarks and eight relational queries from the standard TPC-H database management system benchmark suite [46]. We generate the TPC-H benchmark database using the official generator `dbgen` with a scale factor of $0.3$ (about 380MB data). This scale factor is a typical size of the database partition assigned to each CPU when TPC-H is used to benchmark a distributed database [13, 29]. To emphasize how LLVM optimization levels and interpreters compare to copy-and-patch, we report their startup and execution times normalized to multiples of copy-and-patch. We report the absolute running time for every experiment in Appendix C in the supplemental material.

### 6.2 Startup–Execution Time Pareto Frontier

We explore the startup–execution time Pareto frontier of several approaches to online code generation. The Pareto frontiers are the set of Pareto efficient points for which no gain can be had in startup time without giving up some execution time and vice versa. We compare copy-and-patch (C&P) to the LLVM compilation levels `-O0`, `-O1`, `-O2`, and `-O3`. We also include the Peloton interpreter [21] from the query execution engine of the Peloton database management system [30, 37]. It interprets a subset of the LLVM IR, but this IR is low-level and leads to high interpretation cost. Therefore, to get a better baseline for interpretation, we developed a higher-level AST Interpreter that runs approximately $1.5\times$ faster than Peloton's interpreter.

We used these systems to prepare and execute three microbenchmarks and recorded their startup time and execution time on synthetic input. The microbenchmarks include a function to compute Fibonacci numbers (Figure 7a), an implementation of Euler's sieve [15], and an implementation of quicksort. The Fibonacci function is small with two recursive calls, so it demonstrates the efficiency of the generated code across calls. Euler's sieve is heavy on arithmetic



**(a)** Fibonacci numbers **(b)** Euler's sieve **(c)** Quicksort

**Figure 10.** The Pareto frontier of LLVM's compilation levels, C&P, and interpreters on three microbenchmarks. C&P dominates the LLVM -O0 optimization level: it produces better code in two orders of magnitude less time. C&P also replaces interpretation in practice: both have negligible startup overhead, but C&P's generated code runs an order of magnitude faster.

and demonstrates the efficiency of generated code across compute stencils. And quicksort is a mix of these traits.

The C&P technique moves the Pareto frontier of the three microbenchmarks, effectively rendering both -O0 compilation and interpretation obsolete. Figure 10 plots the startup time of each approach against the execution times for the three microbenchmarks. LLVM -O2 and -O3 have the same startup and execution times, so we show only LLVM -O2. All times are normalized to multiples of C&P. In all three cases, LLVM -O0 compilation falls behind the Pareto frontier, meaning C&P is a strictly better choice. The AST interpreter generally has a lower startup cost than C&P, but in both cases the cost is negligible (1 µs vs 1 −3 µs) compared to all but trivial program executions. We therefore posit that using the copy-and-patch technique is preferable to interpreters in metaprogramming systems. Moreover, the Pareto frontier line between C&P and LLVM -O1 is steep, meaning the improved execution performance comes at a high compilation cost. For example, for Euler's sieve, a 1442× higher compilation cost yields only a 24% execution performance gain. This decreases the number of applications that benefit from optimizing compilation, and it should therefore be reserved for functions that will be run for a significant period of time.

Finally, we measured the startup and execution time of the microbenchmarks implemented in Java, using OpenJDK 11's Java bytecode interpreter [4] and HotSpot JIT [35]. This compares C&P to an industry-strength interpreter and JIT. We note that this is not an apple-to-apple comparison, since Java has the advantage of working on pre-compiled and pre-optimized bytecode, instead of a high-level AST generated at runtime. Nevertheless, the Java bytecode interpreter's execution time is 4.4×–36× slower than C&P. The HotSpot JIT's compilation time, measured with JITWatch [32], is 0.6–1.2 times LLVM -O3, while its execution time is 1.2–1.7 times LLVM -O3. Thus, the conclusion of our comparison between C&P and LLVM holds for Hotspot JIT as well.

## 6.3 TPC-H Performance

We demonstrate the performance of the copy-and-patch algorithm by comparing it to the LLVM compiler and our AST interpreter on eight query benchmarks from the standard TPC-H collection [46], which are indicative of real-world metaprogramming targets. We implemented a small database query compiler similar to Hyper [31] and MemSQL [28], using our metaprogramming system built on top of copy-and-patch to compile query plans to executable code. Our compiler supports most of the important SQL execution plan nodes, including table scan, filter, hash join, projection, aggregation, group-by, order-by, and a number of SQL scalar operators. While the execution plan nodes we implemented should be sufficient to execute most of the TPC-H queries, many queries require complicated rewriting to yield a reasonable plan, which is unrelated to the point of our evaluation. We therefore implemented those 8 TPC-H queries whose execution plans follow directly from the query text.

Our implementation is simple, with only 600 lines of code generation logic. Industry database management systems, including MemSQL [28] and PostgreSQL [40], generate much more code to support the complexities required by a real-world database. These complexities include SQL-specific semantics (e.g. null-related behavior, collations), transaction semantics, more complex data structure and execution strategy, larger-than-RAM datasets, parallel and distributed execution, and more. As a result, measured with the MemSQL product trial, those queries take 4.8×–52× (average 16.4×) longer to compile than our query compiler using LLVM -O3, and compilation can take up to 4.5 seconds, making compilation time a concern.

**Comparison to LLVM Compilation.** We demonstrate the performance of copy-and-patch compared to LLVM on TPC-H benchmark queries. We build ASTs from query plans using our metaprogramming system and lower them to binary code with copy-and-patch and LLVM at each optimization level. Figure 11 (left) shows the compilation time of each LLVM

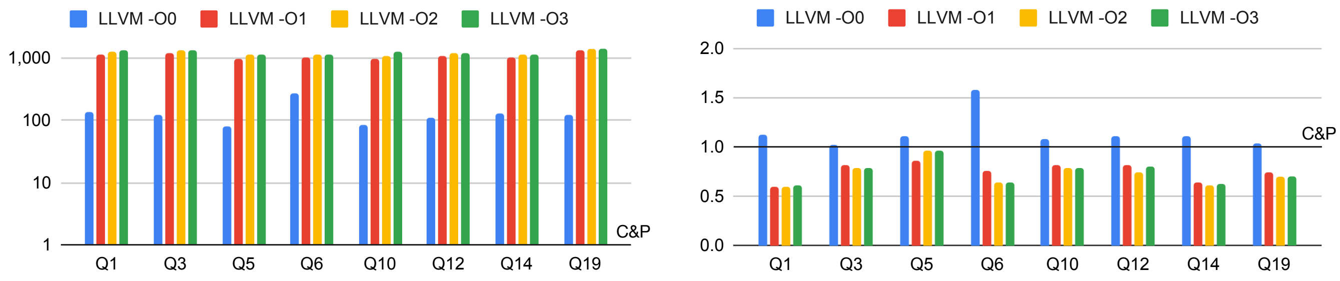


**Figure 11.** Normalized startup times (left) and execution times (right), as multiples of C&P, of code generated by the LLVM optimization levels across the TPC-H queries. C&P generates code two orders of magnitude faster (up to 267×) than LLVM `-O0` and three orders of magnitude (up to 1384×) faster than the other LLVM optimization levels. The resulting code performs on average 15% better than LLVM `-O0`, 24% slower than LLVM `-O1`, 27% slower than LLVM `-O2`, and 26% slower than LLVM `-O3`,

optimization level and Figure 11 (right) shows the execution time of the resulting code, both normalized to multiples of copy-and-patch. LLVM `-O0` takes two order of magnitude more time to compile and produces less performant code. The other LLVM optimization levels take about three orders of magnitude more time to compile (936–1377×), but produces 4%–40% better performing code (the average and median for `-O3` are both 26%). This shows that the results from Section 6.2 hold for realistic code in a metaprogramming system. Copy-and-Patch dominates LLVM `-O0`, and there are fewer cases that are beneficial to compile at a higher optimization level. While higher optimization levels used to make sense when the average 34% speedup over `-O0` could amortize a 10.5× average increase in compilation time, with copy-and-patch a smaller 26% speedup must amortize an average 1219× increase in compilation time. For example, TPC-H Q5 compiles in 0.25 s with LLVM `-O3` in our compiler, and the resulting code performs 4% better than copy-and-patch. To pay back the cost of the 1106× compilation time increase over copy-and-patch, the query would need to run for 66.7 s. On the TPC-H data set, however, the query finished execution in less than 0.1 s. Furthermore, in an industry-strength database, compilation would take several times longer, but execution would likely be faster due to better-engineered execution strategies, rendering the gap even larger.

**Comparison to AST Interpretation.** The copy-and-patch algorithm outperforms our AST interpreter on the TPC-H benchmarks by an order of magnitude, as shown in Figure 12 (right). As shown in Figure 12 (left), the startup overhead of C&P is two–three times higher than the interpreter, but both are so small that they are negligible: in most cases it takes longer to construct the AST. For example, it takes copy-and-patch 225 µs to generate code for TPC-H Q5, but constructing the AST from query plan already takes 260 µs. And it takes the interpreter 1.15 s to execute it. Thus, copy-and-patch essentially completely replaces interpreters for database query execution.

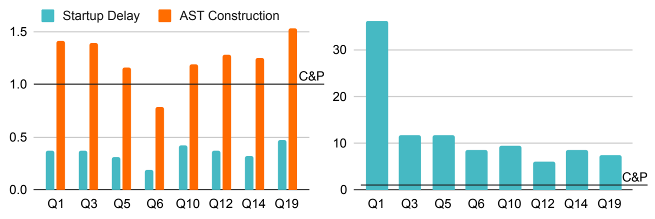


**Figure 12.** Normalized startup times (left) and execution times (right) of the AST interpreter across the TPC-H queries, as multiples of C&P. Although the interpreter requires less than half the preprocessing work of C&P, this cost is negligible since it takes longer to construct the AST. Moreover, the interpreter executes 6–36 times slower than C&P.

### 6.4 Copy-and-Patch Scalability

The copy-and-patch algorithm is linear and requires only two traversals of the AST and one traversal of the CPS call graph. Compilers like LLVM, on the other hand, contain non-linear algorithms. Figure 13 shows how the LLVM optimization levels scale as the input program size grows, on a synthetic function containing a sequence of statements that increment a variable by another variable. The performance of LLVM `-O0` becomes bogged down in instruction selection, while the higher optimization levels spend their time collapsing the increments into a single resulting statement. In both cases, however, LLVM compilation is increasingly slow compared to copy-and-patch as the source code size increases.

### 6.5 Copy-and-Patch Optimization Breakdown

Copy-and-Patch employs several optimizations to produce fast code. Figure 14 shows the impact of these optimizations on the three microbenchmarks as a stacked bar graph. The runtimes are normalized so that unoptimized C&P is at one. The blue bars show the runtime of the optimized versions, and each bar stacked on top shows the runtime added by removing one optimization. The largest gain comes from the core of the C&P technique, which generates specialized

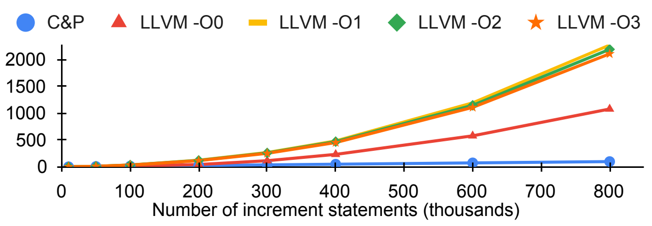


**Figure 13.** The normalized startup time of C&P and LLVM optimization levels as the size of the input program increases. To demonstrate scalability, the time it took each algorithm to compile 10k statements is normalized to 1, so perfect scaling line would end at y-axis 80. C&P scales near-linearly (ending at y-axis 98), while the other lines show worse scalability.

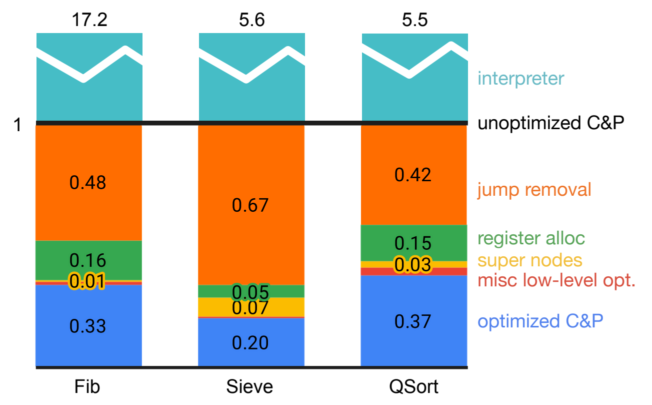


**Figure 14.** Normalized microbenchmark running times with optimized C&P in dark blue. Each bar on top of that shows the running time added by removing each optimization.

AST node implementations with direct branch instructions and directly-embedded runtime constants, yielding a 5.5× to 17.2× speedup compared with an interpreter. Inside C&P, jump removal and light-weight register allocation accounts for the bulk of the runtime saved through optimization. The Sieve benchmark also benefits from the super-nodes, because it is more dominated by memory accesses and arithmetic expressions than the other benchmarks. Finally, a few low-level optimizations, such as instruction block aligning, account for the last part of the optimization gains.

### 6.6 Other Costs and Metrics

The AST in our implementation consists of 25 types of nodes that implement the statements and expressions of an imperative language with some object-oriented capabilities.

The stencil library consists of 98,831 stencils and is constructed at library installation time in 14 min using 6 threads. The code and data in the stencil library consume 17.5 MB of memory at runtime. In contrast, the LLVM library consumes 22.8 MB of memory at runtime in our build. The numbers are measured by computing the total size of symbols belonging to the respective library (whose names start with `llvm::` and `stencil::` respectively), using the `nm` Linux command.

Finally, adding an AST node requires adding a stencil generator, but the required effort is modest. For example, adding a new generator for an add operator with SQL overflow semantics takes 82 lines of code, 17 of which are new logic, while the rest are boilerplate copied from other generators.

## 7 Related Work

We compare the copy-and-patch algorithm and our MetaVar system with work in four areas: interpreters and JIT compilers that our copy-and-patch approach may improve; runtime assemblers that generate machine code from assembly code at runtime; staged compilation that generates binary code by symbolically executing an interpreter; and build-time code library generation approaches that contrast to MetaVar.

### Interpreter Optimizations

Continuation-passing style [43], which is similar to threaded code [1], is the technique we use to weave together the control flow of stencils. In this technique, control flow passes through an AST bottom-up, letting us convert calls to jumps.

Superinstruction [3, 41] is a well-known technique to reduce indirect jump overhead between interpreter opcodes. Our super-node is similar to superinstruction; but in our use case, since unnecessary jumps between opcodes are already eliminated, super-node is only a relatively minor optimization to improve the quality of generated code.

The idea of concatenating binary snippets to form superinstructions and remove tail calls is used in [38] and [11]. Neither paper supported patching the binary code to burn in literals, stack offset, and jump addresses, however; so their technique only works if the binary code can be concatenated without modification. This implies all jumps and calls are still indirect, and all constants must be retrieved from memory. Also, since they hand-implement every operation, they must hand-pick a few super-instructions for common operations. In contrast, our system automatically generates 98,831 implementations, burn in values, and turns all jumps and calls direct. Finally, they lack a register allocation scheme, so the burden of managing registers is pushed to the user that generates the input programs.

The idea of using external variables to locate holes to burn in runtime constants is used in [33]. Their technique is more verbose than ours, yet only works for statically known application logic and thus cannot be used in our use case where the logic is generated at runtime.

### Runtime Assemblers

The desire to keep the low overhead and portability of interpreters without sacrificing performance has led to a hybrid approach where high-level languages are pre-compiled to low-level bytecode, defined here as virtual and portable assembly code. The Java bytecode [23] is a famous example. To avoid interpretation overhead, the bytecode is assembled to machine code at runtime. VCODE [9] proposed to use a library of hand-written platform-specific instruction implementations to speed up the assembling process, and this approach was also followed in AsmJIT [20], DynASM for Lua [36], and as a case study in Terra [8]. The DCG system [10] also attempts greedy register allocation, but only works under the unrealistic assumption that no spilling is ever needed. It also runs 35 times slower than VCODE [9]. However, these assemblers do not solve the most costly problems of lowering a high-level program to machine code, which are optimization, assembly instruction selection, and register allocation. In contrast, the copy-and-patch approach lowers high-level language constructs to machine code, by copying automatically generated binary code snippets, and then patching them together by rewriting values. The benefit of our approach is reflected in both engineering cost and performance. By offloading all low-level and architecture-specific details to Clang, we avoid the prohibitively high engineering cost of re-inventing the big wheel of target-optimized instruction selection and assembling for every architecture[1], and is automatically portable to any architecture supported by LLVM. Furthermore, copy-and-patch pushes the CPU cost of register allocation, instruction selection and assembling to library build time. At runtime, it only copies pre-built chunks of instructions, which is clearly faster than doing register allocation and then selecting and assembling each machine instruction. Finally, its input is an AST for high-level language constructs, which removes the user's burden to lower high-level language to assembly, and is more extensible to add new domain-specific sophisticated AST node types.

### Staged Compilation and Dynamic Specialization

Staged compilation [6, 45] and dynamic specialization [12], given an interpreter implementation and an opcode sequence, generates specialized optimized binary code by symbolically evaluating the interpreter on the opcode sequence. The major advantage is that the user only needs to write an interpreter backend, and the specializer automatically generates binary code. This code generation process, however, runs slower than a hand-written backend, so it is not suitable for the use case where compilation time matters.

### Build-time Code Library Generation

Like the stencil library, FFTW [14] employs a code library approach. At compilation time, it creates a collection of *codelets* that implement optimized variants of FFT on various fixed-length input sequences. It has a dedicated compiler that generates optimized C code implementing codelets. At runtime, input is split into smaller pieces using a divide-and-conquer strategy and, when a piece is small enough, it is dispatched to one of the pre-built codelets. The FFTW codelet library is similar in spirit to the stencil library, except that it only contains implementations of FFTW and it does not burn in constants. The FFTW algorithm calls codelets by indirect function calls, which is acceptable because each pre-built implementation does significant work. Nevertheless, the paper mentioned that reading runtime constants from memory hurts performance. We envision that our MetaVar system's ability to burn runtime constants into instruction flow and make indirect jumps and calls direct could further improve the performance of the generated FFTW codelets.

## 8 Conclusion

Copy-and-Patch code generation moves the Pareto startup-execution trade-off frontier by dominating LLVM `-O0` compilation and traditional interpreters. It replaces `-O0` through two orders of magnitude faster code generation and faster generated code. And it replaces interpreters as both have negligible startup overhead, while code generated by copy-and-patch runs an order of magnitude faster.

We envision copy-and-patch being used in interpreters for high-level languages and in metaprogramming systems where compile times are important. For example, it can be used in the query compilers that are becoming ubiquitous in database management systems. In these systems, it will alleviate the tension between spending time to compile a query that may only execute once and interpreting another query that ends up executing for a long time.

The copy-and-patch algorithm and the stencil generator library are also extensible by design. New AST nodes can be added by users seeking better performance for a new, perhaps domain-specific, language construct. And new super-node stencil generators can be added that implement several AST nodes, thus allowing Clang to better optimize them using sophisticated instructions on modern architectures. The type system of the AST can also be expanded in future work to include vector types to target vectorized instructions.

The algorithm itself can also be extended with new optimizations, implemented by composing stencils, like `mem2reg`, common subexpression elimination, loop unrolling, and vectorization. Of course, these will increase compilation time, but will do so starting from a lower starting point, as shown conceptually in Figure 15. Thus, we

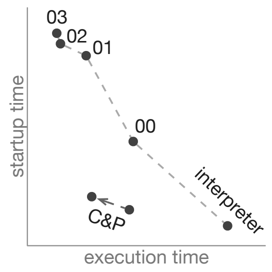


**Figure 15.** Future work

[1]As an example, x86-64 has 3684 different machine instructions [18], orders of magnitude more than the 20th-century RISC CPUs targeted by [10] or [9].

believe extensions of the copy-and-patch algorithm can fill in more points along the Pareto frontier, between optimizing compilers and interpreters.

## Acknowledgments

We are grateful to the following people for helpful comments and discussion: Yinzhan Xu, Cheng Chen, Zhou Sun, Lang Hames, Stephen Chou, Richard Peng, Pat Hanrahan, David Durst, Saman Amarasinghe, Ajay Brahmakshatriya, Scott Kovach, and Alex Aiken.

## References

[1] James R Bell. 1973. Threaded code. *Commun. ACM* 16, 6 (1973), 370–372.

[2] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM review* 59, 1 (2017), 65–98.

[3] Kevin Casey, David Gregg, M. Anton Ertl, and Andrew Nisbet. 2003. Towards Superinstructions for Java Interpreters. In *Software and Compilers for Embedded Systems*, Andreas Krall (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 329–343.

[4] IBM Knowledge Center. 2020. *Disabling the Java JIT Compiler.* IBM. https://www.ibm.com/support/knowledgecenter/SSYKE2_8.0.0/com.ibm.java.vm.80.doc/docs/jit_disable.html

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18).* USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[6] Charles Consel, Luke Hornof, Renaud Marlet, Gilles Muller, Scott Thibault, E-N Volanschi, Julia Lawall, and Jacques Noyé. 1998. Tempo: Specializing systems applications and beyond. *Comput. Surveys* 30, 3es (1998), 5.

[7] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. 2013. Terra: A Multi-Stage Language for High-Performance Computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, New York, NY, USA, 105–116.

[8] Zachary DeVito, Daniel Ritchie, Matt Fisher, Alex Aiken, and Pat Hanrahan. 2014. First-Class Runtime Generation of High-Performance Types Using Exotypes. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation.* Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/2594291.2594307

[9] Dawson R Engler. 1996. VCODE: a retargetable, extensible, very fast dynamic code generation system. *ACM SIGPLAN Notices* 31, 5 (1996), 160–170.

[10] Dawson R Engler and Todd A Proebsting. 1994. DCG: An efficient, retargetable dynamic code generation system. *ACM SIGPLAN Notices* 29, 11 (1994), 263–272.

[11] Martin Anton Ertl and David Gregg. 2003. Implementation issues for superinstructions in Gforth. In *Proceedings of EuroForth 2003.* Citeseer, Herefordshire, UK, 9.

[12] H. Finkel, D. Poliakoff, J. S. Camier, and D. F. Richards. 2019. ClangJIT: Enhancing C++ with Just-in-Time Compilation. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC).* IEEE, Denver, CO, USA, 82–95. https://doi.org/10.1109/P3HPC49587.2019.00013

[13] Dimitri Fontaine. 2018. *PostgreSQL 11 and Just In Time Compilation of Queries.* CitusData. https://www.citusdata.com/blog/2018/09/11/postgresql-11-just-in-time/

[14] M. Frigo and S. G. Johnson. 1998. FFTW: an adaptive software architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, Vol. 3. IEEE, Seattle, WA, USA, 1381–1384. https://doi.org/10.1109/ICASSP.1998.681704

[15] David Gries and Jayadev Misra. 1978. A Linear Sieve Algorithm for Finding Prime Numbers. *Commun. ACM* 21, 12 (Dec. 1978), 999–1003. https://doi.org/10.1145/359657.359660

[16] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362.

[17] John L Hennessy and David A Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (2019), 48–60.

[18] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stratified Synthesis: Automatically Learning the X86-64 Instruction Set. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16).* Association for Computing Machinery, New York, NY, USA, 237–250. https://doi.org/10.1145/2908080.2908121

[19] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.

[20] Petr Kobalicek. 2014. *AsmJIT - Machine code generation for C++.* AsmJIT. https://github.com/asmjit/asmjit

[21] Marcel Kost. 2018. *PelotonDB Interpreter.* PostgresSQL. https://github.com/cmu-db/peloton-design/blob/master/bytecode_interpreter/bytecode_interpreter.md

[22] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, San Jose, CA, USA, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[23] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java virtual machine specification.* Addison-Wesley Professional, Boston, MA, USA. 600 pages.

[24] LinuxBase. 1998. *ARM ELF Relocation types.* LinuxBase. https://refspecs.linuxbase.org/elf/ARMELFA08.pdf

[25] Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell. 2020. *System V Application Binary Interface.* LinuxBase. https://refspecs.linuxbase.org/elf/x86_64-abi-0.98.pdf

[26] Wes Mckinney. 2011. pandas: a Foundational Python Library for Data Analysis and Statistics. In *Python High Performance Science Computer.* IEEE, Seattle, WA, USA, 9.

[27] MemSQL. 2020. *MemSQL Database.* MemSQL. https://www.memsql.com

[28] MemSQL. 2020. *MemSQL Query Code-Generation Documentation.* MemSQL. https://docs.memsql.com/v7.1/key-concepts-and-features/query-processing/code-generation/

[29] MemSQL. 2020. *Personal communication, with permission to disclose to the public.* MemSQL.

[30] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. 2017. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. *Proceedings of the VLDB Endowment* 11 (September 2017), 1–13. Issue 1. https://db.cs.cmu.edu/papers/2017/p1-menon.pdf

[31] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.

[32] Chris Newland. 2020. *JITWatch – Log analyser / visualiser for Java HotSpot JIT compiler.* AdoptOpenJDK. https://github.com/

AdoptOpenJDK/jitwatch

[33] Francois Noel, Luke Hornof, Charles Consel, and Julia L Lawall. 1998. Automatic, template-based run-time specialization: Implementation and experimental study. In *Proceedings of the 1998 International Conference on Computer Languages*. IEEE, Chicago, IL, 132–142.

[34] Oracle. 2020. *64-bit SPARC relocation types*. Oracle. https://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-54839.html#chapter6-24-1

[35] Oracle. 2020. *The Java HotSpot Performance Engine Architecture*. Oracle. https://www.oracle.com/java/technologies/whitepaper.html

[36] Mike Pall. 1999. LuaJIT DynASM. https://luajit.org/dynasm.html

[37] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *Conference on Innovative Data Systems Research*. CIDR, Chaminade, California, 6.

[38] Ian Piumarta and Fabio Riccardi. 1998. Optimizing Direct Threaded Code by Selective Inlining. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 291–300. https://doi.org/10.1145/277650.277743

[39] PostgresSQL. 2020. *Postgres Documentation - JIT thresholds*. PostgresSQL. https://www.postgresql.org/docs/11/runtime-config-query.html#GUC-JIT-ABOVE-COST

[40] PostgresSQL. 2020. *Postgres Documentation - Why JIT*. PostgresSQL. https://www.postgresql.org/docs/11/jit-decision.html

[41] Todd A. Proebsting. 1995. Optimizing an ANSI C interpreter with superoperators. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. ACM, San Francisco, California, 322–332.

[42] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics (TOG)* 31, 4 (2012), 1–12.

[43] Guy Lewis Steele. 1977. Debunking the "Expensive Procedure Call" Myth or, Procedure Call Implementations Considered Harmful or, LAMBDA: The Ultimate GOTO. In *Proceedings of the 1977 Annual Conference (ACM '77)*. ACM, New York, NY, USA, 153–162. https://doi.org/10.1145/800179.810196

[44] The Glasgow Haskell Team and LLVM Team. 2020. *LLVM Documentation on GHC Calling Convention*. The Glasgow Haskell Team and LLVM Team. https://releases.llvm.org/10.0.0/docs/LangRef.html?highlight=ghc#calling-conventions

[45] Scott Thibault, Charles Consel, Julia L Lawall, Renaud Marlet, and Gilles Muller. 2000. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation* 13, 3 (2000), 161–178.

[46] TPC. 2020. TPC-H. http://www.tpc.org/tpch/. Accessed: 2020-11-15.

[47] Andrew Shell Waterman. 2016. *Design of the RISC-V instruction set architecture*. Ph.D. Dissertation. UC Berkeley.