Copy-and-Patch Compilation

A fast compilation algorithm for high-level languages and bytecode

HAORAN XU, Stanford University, USA, USA
FREDRIK KJOLSTAD, Stanford University, USA, USA

Fast compilation is important when compilation occurs at runtime, such as query compilers in modern database systems and WebAssembly virtual machines in modern browsers. We present copy-and-patch, an extremely fast compilation technique that also produces good quality code. It is capable of lowering both high-level languages and low-level bytecode programs to binary code, by stitching together code from a large library of binary implementation variants. We call these binary implementations stencils because they have holes where missing values must be inserted during code generation. We show how to construct a stencil library and describe the copy-and-patch algorithm that generates optimized binary code.

We demonstrate two use cases of copy-and-patch: a compiler for a high-level C-like language intended for metaprogramming and a compiler for WebAssembly. Our high-level language compiler has negligible compilation cost: it produces code from an AST in less time than it takes to construct the AST. We have implemented an SQL database query compiler on top of this metaprogramming system and show that on TPC-H database benchmarks, copy-and-patch generates code two orders of magnitude faster than LLVM -O0 and three orders of magnitude faster than higher optimization levels. The generated code runs an order of magnitude faster than interpretation and 15% faster than LLVM -O0. Our WebAssembly compiler generates code 4.9×–6.5× faster than Liftoff, the WebAssembly baseline compiler in Google Chrome. The generated code also outperforms Liftoff's by 46%–63% on the Coremark and PolyBenchC WebAssembly benchmarks.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**; **Domain specific languages**.

Additional Key Words and Phrases: Fast Compilation, Binary Code Variant Library, Binary Code Patching

ACM Reference Format:
Haoran Xu and Fredrik Kjolstad. 2021. Copy-and-Patch Compilation: A fast compilation algorithm for high-level languages and bytecode. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 1 (November 2021), 30 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Fast compilation is important, particularly when the compilation occurs at runtime. Two representative use cases of runtime compilation are the WebAssembly virtual machines in modern browsers and the query engines of modern SQL databases, where WebAssembly modules and SQL queries are compiled to executable code and then executed. Since the latency experienced by the user is the sum of the time to generate the code (startup delay) and the time to execute the generated code, it is not enough to simply use the most optimizing but slowest compiler. The system must instead balance startup delay with execution performance. As an example, the TurboFan optimizing compiler [Backes 2018] in the Google Chrome browser needs 51 CPU seconds to compile the

Authors' addresses: Haoran Xu, Stanford University, 353 Jane Stanford Way, Stanford, CA, 94305, USA, USA, haoranxu@stanford.edu; Fredrik Kjolstad, Stanford University, 353 Jane Stanford Way, Stanford, CA, 94305, USA, USA, kjolstad@cs.stanford.edu.

2475-1421/2021/11-ART1
https://doi.org/10.1145/nnnnnnn.nnnnnnn

WebAssembly module that powers the online AutoCAD Web App [AutoCAD 2018] on our system, which is too long for users to wait. As another example, the MemSQL [MemSQL 2020a] database takes up to 4.5 seconds to compile a TPC-H [TPC 2020] database benchmark query using LLVM `-O3` on our system. And business-intelligence software may generate complex queries that take minutes to compile [MemSQL 2020c].

To balance startup delay and execution performance, runtime execution environment typically contain several execution tiers that occupy different points on the startup delay–execution performance Pareto frontier. Typical choices include interpreters, baseline compilers (also called template JITs), and choices of different optimizing levels for an optimizing compiler (e.g., LLVM `-O0`, `-O1`, `-O2`, and `-O3`). Modern databases, including Hyper [Neumann 2011], Peloton [Menon et al. 2017], PostgreSQL [PostgresSQL 2020], and MemSQL [MemSQL 2020a], all employ tiered execution strategies that adds an interpreter, LLVM `-O0`, or both below the most optimizing LLVM `-O3` tier. Web browsers, on the other hand, use dedicated baseline compilers instead of interpretation or the `-O0` version of an optimizing compiler. For example, the Google Chrome WebAssembly virtual machine first compiles using the fast Liftoff baseline compiler [Backes 2018] and then recompiles in the background using the optimizing TurboFan compiler. The motivation for using dedicated baseline compilers over `-O0` compilation with optimizing compilers is startup delay. Baseline compilers such as Liftoff translate bytecode to machine code in one pass, without going through the intermediate representations of an optimizing compiler. As they move through the bytecode stream, they inspect each bytecode and emit appropriate machine code, either using a platform-dependent assembler as in Liftoff or pre-compiled from a high-level language like C or Java [Ertl and Gregg 2003; Iliasov 2003; Piumarta and Riccardi 1998; Wimmer et al. 2013]. Thus, baseline compilers offer faster compilation than the `-O0` compilation of an optimizing compiler—often measured in tens of megabytes of code generated per second—as well as better execution performance than an interpreter.

Interestingly, according to our survey of the 15 code-generating databases enumerated by Pavlo [2021], not a single database has implemented a dedicated baseline compiler. This is not a coincidence. Database query compilers need to express the generated logic in a high-level metaprogramming language for expressiveness. And writing a baseline compiler to efficiently translate a high-level language with many complex language features directly to executable code is far more challenging than doing so for bytecode, where the input has already been processed into a stream of low-level opcodes that map closely to machine instruction. We therefore categorize compilers into *full compilers* that compile from a high-level language to machine code and *bytecode assemblers* that assemble a linearized stream of low-level bytecode to machine code, as illustrated in Figure 1.

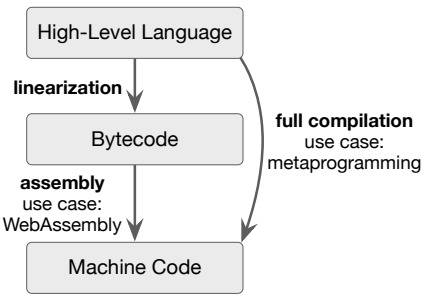


Fig. 1. Phases of compilation: linearization, assembly, and full compilation.

In this paper, we propose a new algorithm for template-JIT-style baseline compilers called copy-and-patch. Unlike prior baseline compiler techniques, copy-and-patch can be used to create both *full compilers* and *bytecode assemblers*. In addition, it shifts the startup delay–execution performance Pareto frontier by a large margin in both worlds.

In the world of bytecode assemblers, we implemented a WebAssembly compiler based on copy-and-patch. Our compiler achieves both lower startup delay and better execution performance than prior baseline compilers. Figure 2 shows the performance of six WebAssembly compilers on the PolyBenchC benchmark [Louis-Noel Pouchet 2011], normalized to our performance. Our compiler has 6.5× lower startup delay than Liftoff, while generating on average 63% better-performing code.

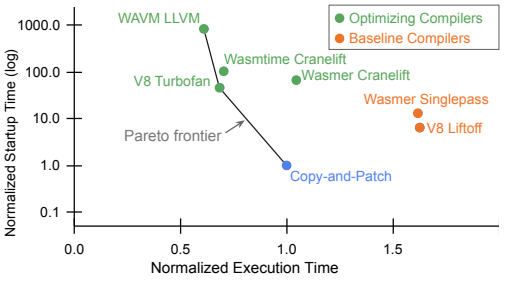


Fig. 2. A scatter plot of normalized startup delay against execution time for seven WebAssembly compilers averaged over the PolyBench benchmarks. Our copy-and-patch compiler replaces baseline compilers on the Pareto frontier.

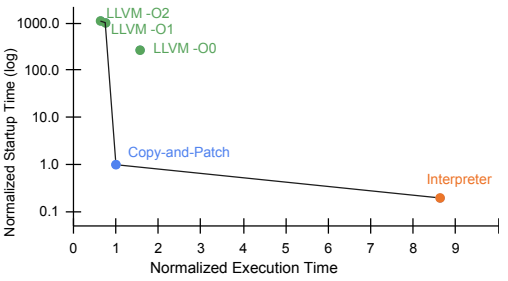


Fig. 3. A scatter plot of normalized startup execution times against execution times for five strategies for executing the sixth TPC-H query implemented in a high-level language. Our copy-and-patch compiler replaces LLVM `-O0` on the Pareto frontier.

In addition to Liftoff, our compiler also displaces Wasmer SinglePass [Zhou 2018], the baseline compiler used in Wasmer [Akbary et al. 2018], and even Wasmer Cranelift [Akbary 2018], a relatively slow optimizing compiler, from the Pareto frontier.

In the world of full compilers, we used copy-and-patch to implement a compiler for a C-like high-level language intended for metaprogramming. The language is implemented as a DSL embedded in C++. To demonstrate that one can get a full compiler for a powerful language from copy-and-patch with reasonable efforts, our language supports a variety of complex features, including the C typesystem, all major C language constructs, the ability to use C++ classes and call C++ functions/methods in the host code, C++ exceptions semantics and destructor semantics, and so on. We then built a simple SQL database query compiler on top of this metaprogramming system. To the best of our knowledge, this implementation is the first database query compiler equipped with a dedicated baseline compiler. We evaluated its performance on eight TPC-H database queries [TPC 2020]. Figure 3 shows the Pareto frontier of one of them. The compilation time of our compiler is so low that it is less than the time it takes to construct the AST of the program. Compared with interpreters, both have negligible startup delay (since constructing ASTs takes longer), but our execution performance is an order of magnitude faster. Compared with LLVM `-O0`, our implementation compiles two orders of magnitude faster and generates code that performs on average 15% better. Therefore, we conclude that copy-and-patch renders both interpreters and LLVM `-O0` compilation obsolete in this use case.

At a high level, copy-and-patch works by having a pre-built library of composable and parametrizable binary code snippets that we call binary stencils.[1] At runtime, optimization and code generation become the simple task of looking up a data table to select the appropriate stencil, and instantiate it to the desired position by copying it and patching in the missing values. Our contributions are:

(1) The concept of a binary stencil, which is a pre-built implementation of an AST node or bytecode opcode with missing values (immediate literals, stack variable offsets, and branch and call targets) to be patched in at runtime.
(2) An algorithm that uses a library with many binary stencil variants to emit optimized machine code. There are two types of variants: one that enumerates different parameter configurations (whether they are literals, in different registers, or on the stack) and one that enumerates

[1]The WebAssembly compiler uses 1666 stencils taking 35 kB and the high-level compiler uses 98,831 stencils taking 17.5 MB.

different code patterns (a single AST node/bytecode or a supernode of a common AST subtree/bytecode sequence).

(3) An algorithm that linearizes high-level language constructs like if-statements and loops, and generates machine code by composing multiple binary stencil fragments.

(4) A system called MetaVar for generating binary stencils, which allows the user to systematically generate the binary stencil variants in clean and pure C++, and leverages the Clang+LLVM compiler infrastructure to hide all platform-specific low-level detail.

We evaluate our algorithm by evaluating the copy-and-patch-based compilers we built for WebAssembly and our high-level language.[2] We compare the WebAssembly compiler to six industrial WebAssembly compilers, including those from Google Chrome and Wasmer. Our results show that our algorithm replaces all prior baseline compilers on the Pareto frontier and moves first-tier compilation closer to the performance of optimizing compilers. And we compare the high-level language compiler based on copy-and-patch with compiler implementations based on LLVM using different optimization levels, showing that our technique compiles two orders of magnitude faster than LLVM `-O0` while producing better code. We also show a breakdown of the performance contributed by different features in our compiler.

2 OVERVIEW

The topic of our paper is the copy-and-patch compilation algorithm and the associated MetaVar compiler. But to motivate their use and to evaluate them, we also built two compilers: one for WebAssembly and one for a high-level language. In this section, we first give an overview of the copy-and-patch algorithm and its surrounding ecosystem of tools and then give an overview of the two compilers.

2.1 Copy-and-Patch and MetaVar Systems

The copy-and-patch system consists of two components: the MetaVar compiler and the copy-and-patch code generator. Figure 4 shows their relationship. The key to the copy-and-patch algorithm is the concept of a binary stencil, which is a partial binary implementation of a bytecode instruction or an AST node of a high-level language. The MetaVar compiler generates many binary stencils that implement different optimization cases for every bytecode or AST node. The MetaVar compiler takes as input bytecode/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 generator, together with a bytecode sequence or an AST that implements a function. The code generator then produces binary code that implements the function, by copying and patching together stencils that implement the bytecodes or 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 for each bytecode/AST node, and the copy-and-patch code generator produces optimized code by selecting among these variants. For example, if the AST contains an addition with a constant, then copy-and-patch 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 place the stencils of operators that follow each other in consecutive locations in memory, which lets it remove the

[2]The source code of both compilers is available open source in the supplemental material. In addition, the high-level language compiler is at the time of writing maintained at https://github.com/sillycross/PochiVM.

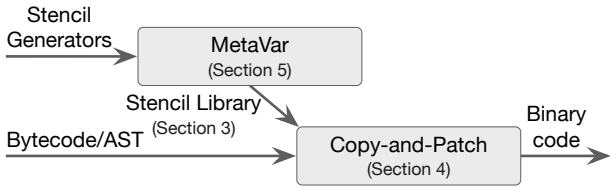


Fig. 4. The copy-and-patch system compiles a high-level language AST or a bytecode sequence to binary code. It consists of the MetaVar compiler, which compiles stencil generators to binary stencils, and the Copy-and-Patch code generator, which generates the executable code for an AST or a bytecode sequence by copying and patching stencils.

jump between them. Together, these and other optimizations by stencil variant selection produces code that outperforms an interpreter, a bytecode baseline compiler, and even LLVM `-O0`.

2.2 Use case: WebAssembly Compiler

WebAssembly is a bytecode format designed as a portable compilation target for programming languages, with the goal of enabling untrusted code to be executed safely and efficiently on any platform. Since WebAssembly code cannot be executed directly and an interpreter is too slow [Backes 2018], it must be assembled to machine code at runtime before it can execute.

WebAssembly modules can often be large. For example, the AutoCAD Web App [AutoCAD 2018] is powered by a WebAssembly module of 47.5 MB; and *clang.wasm* [Smith 2018], the clang compiler in WebAssembly, is 30.5 MB. Since the user cannot interact with an application until the code starts executing, a fast baseline compiler is critical for a good user experience. As such, major web browsers like Chrome, Firefox, and Safari and major non-web WebAssembly runtimes like Wasmer and Wasmtime provide baseline compilers for WebAssembly, which prioritize a low startup delay at the expense of lower execution performance [Backes 2018; Bastien et al. 2017; Clark 2018; Gohman 2018; Wingo 2020; Zhou 2018]. On the other hand, performance is also important, since the major selling point of WebAssembly is that it lets native applications run at near-native speed on the Web, and the code generated by the baseline compiler will be executed until the optimizing compiler finishes, which can take a long time. The need for both extremely fast compilation and good execution performance makes WebAssembly a representative use case for copy-and-patch.

We implemented a WebAssembly baseline compiler using copy-and-patch. Figure 5 shows its role in replacing the Tier 1 baseline compiler in a web browser, for both lower startup delay and better execution performance. Our compiler supports the full WebAssembly 1.0 core specification [Group 2017], as well as a subset of the WASI embedding [Group 2018] necessary to run the benchmarks in Section 6. We note that an embedding only defines an agreement on how the imported functions of a WebAssembly module shall be implemented, so supporting more embeddings has nothing to do with code generation.

2.3 Use case: High-Level Language Compiler

Our second use case for copy-and-patch is a metaprogramming language embedded in a C++ library that can be used to generate code at runtime. Example uses include database query engines and DSL libraries. Since a database query engine is expected to provide low latency, a metaprogramming system serving them must provide an execution path with low startup latency in addition to an optimizing path. In the database world the optimizing path is typically LLVM `-O3`, and the low startup latency path is either an interpreter or LLVM `-O0`. Figure 6 shows the role of copy-and-patch in such a metaprogramming system. Copy-and-Patch provides better execution performance than LLVM `-O0` at a negligible startup delay similar to an interpreter, thus rendering both interpreters

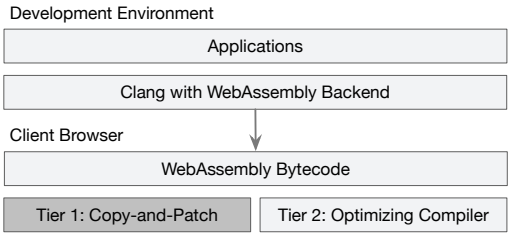


Fig. 5. Copy-and-Patch can be used by WebAssembly compilers in browsers to both speed up Tier 1 compilation and produce better code. And it can be combined with a Tier 2 optimizing compiler to recompile hot code when the increased performance can amortize the orders of magnitude slower compilation.

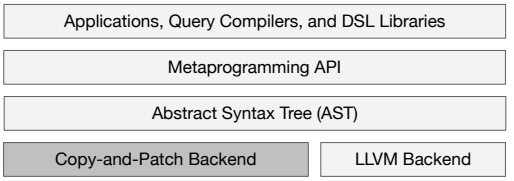


Fig. 6. Copy-and-Patch can be used by metaprogramming systems to speed up their interpreters past LLVM `-O0`. And it can be combined with LLVM to let the user request higher optimization levels when the increased performance can amortize the orders of magnitude higher compilation cost.

and LLVM `-O0` obsolete in this use case. The user of this metaprogramming system can either use copy-and-patch or compile with LLVM `-O1` or higher at a much higher compilation time.

We have developed a compiler that implements the copy-and-patch code generation technique for the high-level metaprogramming language. It supports all major imperative 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 [Bezanson et al. 2017], Halide [Ragan-Kelley et al. 2012], TACO [Kjolstad et al. 2017], Hyper [Neumann 2011], Peloton [Pavlo et al. 2017], and Terra [DeVito et al. 2013].

3 THE STENCIL LIBRARY

The stencil library contains binary implementations of bytecode or AST node types that are stitched together at runtime by the copy-and-patch algorithm to generate code for a bytecode module or a function in a high-level language. We call the binary implementations 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 bytecode or AST node type that are specialized for different operand types, value locations, and more. The variants let 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 to a significant number of stencils. Our WebAssembly implementation contains 1666 stencils, taking 35 kB of memory. Our high-level language implementation is larger because it includes many supernodes and contains 98,831 stencils, taking 17.5 MB of memory. Although we believe these library sizes are practical for our respective use cases, there are too many stencils for it to be practical to write them by hand.

A binary stencil is a binary code function that implements a computation logic fragment, where literals, jump addresses, and stack offsets are missing. Each computation logic fragment implements the semantics of one AST node or bytecode, or a commonly-used shape of an AST subtree or bytecode sequence (we call such stencils supernodes). Supernodes allow optimizations across node boundaries, resulting in better quality of generated machine instructions. During copy-and-patch

code generation, as we describe in Section 4, the runtime AST or bytecode sequence are pattern-matched to stencils and supernode stencils. The selected 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) [Steele 1977] 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 7 shows how 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 compile stencils lowers them to jump instructions. Combined with the GHC calling convention [GHC and LLVM 2020], 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.

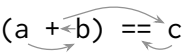


Fig. 7. CPS

Register allocation is another important optimization required for fast binary code. The obvious way to pass a temporary value between stencils is to reserve a slot in the stack frame whose offset is represented by a stencil hole. However, this is suboptimal in term of performance (since each read/write is a memory access), and we want to allow temporary values to be passed around in CPU registers. The trick to accomplish this also lies in the GHC calling convention, where all function parameters are passed in registers. Therefore, to pass a value as a parameter to the continuation is to pass this value in register to another stencil. In other words, we repurpose the function prototype and the calling convention as a register allocation protocol, where each function parameter implicitly corresponds to some physical register determined by the calling convention. We generate different variants of stencils with different function prototypes for different register configurations, so that at runtime we can pick the right one based on the circumstance.

We cannot naively enumerate all possible combinations of function prototypes for the different types of values that may be passed through, since the total number of combinations grows exponentially. The crucial observation is that each stencil only cares about its own inputs. The contents stored in the other registers do not matter, as long as they are not clobbered by the stencil. Therefore, for those registers, it is sufficient to always represent it by the longest type (`uint64_t` or `double`), and pass it from the argument to the continuation verbatim. We demonstrate this with a concrete example as shown in Figure 8. In this example, we have three stencils. Stencil 1 produces a temporary value `x` of type `int`, which is to be consumed by stencil 3. But stencil 2 is executed in between, so it must be instructed to not clobber the register holding the value. As shown in the figure, stencil 1 calls its continuation with the temporary value `x` as a new parameter. This puts the value in the register. Stencil 2 does not care about what is stored in the register, but it must not clobber it. This is achieved by having `x` passed directly from the parameter to its continuation: we call it a pass-through parameter. There are two points worth mentioning. First, despite that the true type of `x` is `int`, the type of the pass-through parameter is `uint64_t`. This prevents the exponential explosion of different type combinations

```
void stencil1(uintptr_t stack) {
  int x = /* assign value to x */;
  (void(*)(uintptr_t, int) □ )(stack, x);
}

void stencil2(uintptr_t stack, uint64_t x) {
   // computation unrelated to x
   (void(*)(uintptr_t, uint64_t) □ )(stack, x);
}

void stencil3(uintptr_t stack, int x) {
   // do something with x
}
```

Fig. 8. Three stencils that are executed in the order of the arrows. The first stencil produces a temporary value `x` that we want to pass in a register to the third stencil. The second stencil is executed in between, so it must not clobber the register. This is achieved by the pass-through parameter in the second stencil.

```
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 that passes through a value stored in a register by a prior operation 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.

Fig. 9. The conceptual logic of four out of 98,831 stencils in the stencil library for the high-level language compiler, generated by instantiating 25 stencil generators written as C++ templated functions. Blue boxes indicate the holes in stencils, whose implementations are explained in Section 5.1.

as explained earlier. Second, the value comes in as the second parameter and is passed to the continuation as the second parameter as well, which guarantees both correctness and performance. Correctness is guaranteed by the C language semantics: the callee should see whatever passed by the caller. Performance is guaranteed by the calling convention, due to which we can expect `x` to live in the same register. Therefore, passing `x` to the continuation is a no-op in the generated machine code. The pattern shown in the example can be generalized. A stencil can assign a value to a register by adding it as a new parameter to the continuation. A stencil can access the value by having a parameter with matching type at the same parameter ordinal, and can protect the value for future use by passing it through to the continuation. The lifetime of the value ends when it is no longer passed to the continuation, and the corresponding register is then free for future use. In our current implementation, we only use registers to store temporary values while evaluating expression trees. However, we note that this mechanism can be used to implement the `mem2reg` optimization to keep hot local variables in registers as well (see Section 4).

To summarize, the stencils come in many variants per bytecode or AST node type, use continuation-passing style, and leave missing values to be filled in by the copy-and-patch algorithm. Figure 9 shows conceptual C++ code for four stencils, replacing the missing values with blue numbered boxes. We show the implementations of these boxes in Figure 13. The C++ stencils are compiled by MetaVar at compiler installation time to produce binary stencils, as described in Section 5.2. Figures 9a–9c show three of the stencils that implement equality expressions. Figure 9a 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 9b 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 9d 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 9a. 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 9c. The pass-through stencil, by inserting arguments passed through to the continuation, forces Clang to ensure register values at function entry are unchanged when the continuation is called. This encourages Clang to use other available registers in the function body. When the 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.

```
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];
}
```

Fig. 10. The data structure that stores a binary stencil and the method that materializes it 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.

The MetaVar system compiles C++ stencils to binary stencils that contain binary code and information about where the missing values are located. Figure 10 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:

$$(\text{configuration}) \rightarrow (\text{stencil}).$$

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. 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.

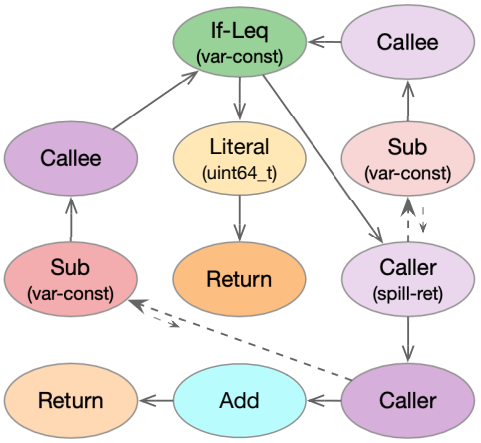
4 COPY-AND-PATCH CODE GENERATION

The copy-and-patch algorithm can be used to compile both bytecode and high-level languages. We will only describe compilation of high-level ASTs here, but the algorithm can be adapted to compile bytecode by removing the step that linearizes a high-level AST using the CPS graph.

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.

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

(a) C++ code that constructs the AST of the Fibonacci function.



(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).

Fig. 11. Fibonacci function AST, its CPS call graph, and the assembly of the binary code generated by copying and patching each CPS call graph node. Each node or supernode has the same color in each representation.

```
void if_leq(uintptr_t stack) {
  int lhs = *(int*)(stack + 1);
  int rhs = 2;
  if (lhs <= rhs) {
    ((void(*)(uintptr_t) 3 )(stack);
  } else {
    ((void(*)(uintptr_t) 4 )(stack);
  }
}
```

(a) The template-instantiated C++ logic for the If-Leq stencil in Figure 11b.

```
0xb8 0x00 0x00 0x00 0x00 2
0x41 0x39 0x85 0x00 0x00 0x00 0x00 1
0x0f 0x8f 0xee 0xff 0xff 0xff 4
0xe9 0xe9 0xff 0xff 0xff 3
```

(b) Clang generates object code, with holes indicated by linker relocation records.

```
binary: { /* omitted, see Figure (b) */ }
pc32Patches:  { 14 /*binaryOffset*/, 19 /*binaryOffset*/ }
sym32Patches: {
  {  1 /*binaryOffset*/, 2 /*holeOrdinal*/ },
  {  8 /*binaryOffset*/, 1 /*holeOrdinal*/ },
  { 14 /*binaryOffset*/, 4 /*holeOrdinal*/ },
  { 19 /*binaryOffset*/, 3 /*holeOrdinal*/ }
}
sym64Patches: {}
```

(c) The MetaVar compiler parses the object file, and generates the Stencil struct as described in Figure 10.

```
20: b8 02 00 00 00        mov $0x2, %eax
25: 41 39 85 08 00 00 00  cmp %eax, 0x8(%r13)
2c: 0f 8f 0e 00 00 00     jg  40
32: e9 e9 ff ff ff        (jmp removed to fallthrough)
```

(d) At runtime, C&P generates executable code by copying the object code and patching the holes with runtime known values.

Fig. 12. The lifetime of the If-Leq (var-const) supernode stencil in Figure 11b.

An AST is lowered to binary code through a CPS call graph. Figure 11 shows the representations that the Fibonacci function goes through on its way from the AST shown as printed code in Figure 11a, through the CPS call graph in Figure 11b, to machine code shown in assembly form in Figure 11c. 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 reduce the number of spills. In our benchmarked implementations, we only use registers to preserve temporary values produced while evaluating an expression. Specifically, given a budget on the maximum number of registers and an expression tree where each expression node produces a value, we perform register allocation using the algorithm described as follows. We perform a post-order traversal of the AST to abstractly evaluate the expression, and we always evaluate the children from left to right for each expression node. The traversal maintains the stack of outstanding temporary operands and, at each step, marks everything below the maximum number of register watermark to be spilled. Additionally, 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 (i.e., all registers are caller-saved). This algorithm is a simplified version of the Simple Sethi-Ullman Algorithm [Sethi and Ullman 1970; Wikipedia 2021] that does not choose between the orders of evaluating a node's children. We chose to use our modified algorithm primarily due to its very low overhead and little loss of practical effectiveness. The Simple Sethi-Ullman produces better decision than ours when it is beneficial to evaluate a right subtree first, but this requires the expression tree being both large and skewed toward the right side, which is not common. Nevertheless, we note that implementing the Simple Sethi-Ullman algorithm or the Advanced Seth-Ullman algorithm are both possible: it is only a trade-off between startup delay and execution performance, not a limitation of our technique.

As an example, Figure 11c shows the effectiveness of our register allocation for the Fibonacci function: no temporary values were spilled except the result of the first function call on line 58, since its lifetime crosses the second function call. All other temporary values are passed in registers to the stencils that consume them: to name a few, the second Caller stencil to the Add stencil, the Add stencil to the Return stencil, and the Sub (var-const) stencil to the Callee stencil. There are also examples where a temporary value is protected by a pass-through parameter: the Caller stencil produces a temporary value of the new stack frame address, which is to be consumed by the Callee stencil. The Sub (var-const) stencil is executed in between, so a variant of the Sub (var-const) stencil that takes one pass-through parameter is selected to protect the value from being clobbered.

We have also explored the possibility of performing `mem2reg` optimization to promote the storage of hot local variables from the stack to registers. Our prototype `mem2reg` implementation gives up to 10% execution performance boost, but results in about 3× slower compilation. We deemed this trade-off as not worthwhile in our use cases, so in our benchmarks, we choose to not perform this optimization. However, in other use cases it may be worthwhile to include this optimization.

During the AST traversal, we also plan the stack frame layout for the function, assigning a storage offset in the stack frame for each local variable and spilled temporary value. As another small optimization, once the lifetime of a spilled temporary value ends, its slot in the stack frame can be reused for another spilled temporary. This reduces stack frame size and improves locality.

The next 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 variant, depending on the AST tree shape and the context (e.g., whether the inputs live on the stack or in registers). It does a simple tree pattern matching to find sub-trees that can

be implemented with an efficient supernode stencil. Supernodes allow Clang to optimize larger regions of code. For example, by leveraging advanced assembly instructions supported by the target architecture (e.g., advanced x86-64 addressing modes), an array access indexed by a constant or local variable can be compiled into fewer instructions. Having more supernodes allows better local optimizations and improves execution performance, at the cost of a slightly higher startup delay (since more branches are executed to perform the pattern matching) and a higher static memory footprint. Therefore, for our high-level language compiler, designed for database use cases in which memory footprint is less of a concern, we generated close to 100,000 supernodes, covering a large set of common logic patterns in programs. One example is demonstrated by the If-Leq (var-const) supernode in Figure 11b, which implements an if-branch with a condition clause doing a less-or-equal comparison between a local variable and a constant `int`. We stress that this is not a special case, and not even the most complex case: just to name a few more examples, logic like `if (a[i] <op> b[j])`, or `c = a[i] <op> b[<literal>]` can be implemented by one supernode as well, for any compatible types of local variables `a,b,c,i,j`. Extensive supernode generation is made possible by our powerful MetaVar system, which allows us to systematically generate large numbers of supernodes easily (see Section 5). However, for applications where a small static memory footprint is desired (e.g., WebAssembly), we can simply remove the supernode stencils to get a small stencil library. As an example, the stencil library of our WebAssembly compiler is only 35 kB. The user can also add new supernode stencils specific to his/her use case, such as fused multiply-add. When a stencil is selected for one or several AST 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 11b 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.

The CPS call graph is then lowered 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 11c 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 10 contains the stencil struct that stores information about missing values, and the logic to patch them. Figure 11c shows the filled in missing values in blue. For instance, the jump target on line a6 jumps to the If-Leq (var-const) stencil on line 20, and the value 0x2 on line 20 is the literal 2 from the `int` constant literal AST node in the if-branch condition of Figure 11a.

The copy-and-patch technique also supports external function calls. External functions are important in database applications like the TPC-H queries in Section 7.3, 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 13. 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.

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 10. 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.

```
struct ArithAdd {
  template<typename T /* OperandType */,
           bool spillOutput,
           NumPassthroughs numPassThroughs,
           typename... Passthroughs>
  static void g(uintptr_t stack, Passthroughs... pt, T a, T b) {
    T c = a + b;
    if constexpr (! spillOutput) {
      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 /* OperandType */,
           bool spillOutput,
           NumPassthroughs numPassThroughs>
  static constexpr bool f() {
    if (numPt > numMaxPassthroughs - 2) return false;
    return !std::is_same<T, void>::value;
  }

  static auto metavars() {
    return createMetaVarList(
      typeMetaVar(),
      boolMetaVar(),
      enumMetaVar<NumPassthroughs::X_END_OF_ENUM>());
  }
};

extern "C" void generate(StencilList* result) {
  runStencilGenerator<ArithAdd>(result);
}
```

Fig. 13. A simplified addition stencil generator written with C++ template metaprogramming. The generator is processed by MetaVar to generate stencils for addition.

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[3] [Matz et al. 2020]. 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 13 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

[3]There is an architecture idiosyncrasy involved here. To make the most efficient use of the x86-64 instruction set, Clang by default assumes that the address of an extern symbol fits in signed 32 bits (the "small code model" assumption), despite that the address is 64 bits. If the value we want to encode could be larger, we need to tell Clang the correct assumption.

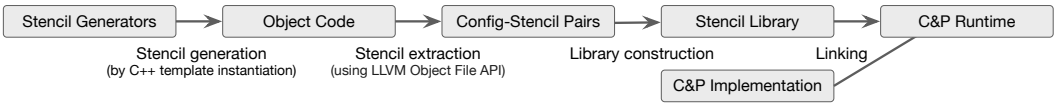


Fig. 14. MetaVar compiles stencil generators to a library that is linked to the copy-and-patch implementation.

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 14 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 13) 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 ⟨configuration, function pointer⟩ 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 in Figure 13, a stencil generator contains a `generate()` stub function, containing a piece of boilerplate code that invokes our `runStencilGenerator` function and returns a list of stencil configuration and function pointer pairs. Stencil extraction uses the LLVM JIT machinery to execute this `generate()` 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 [Matz et al. 2020]. Although this computation rule is technically architecture-dependent, it applies to most

```
struct ArithOp {
  template<typename OperandType,
           AstArithmeticExprType operatorType,
           bool isLhsInRegister
           bool spillOutput,
           NumPassThroughs numPassThroughs,
           typename... Passthroughs>
  void f // ... implementation omitted ...
};
```

```
template<> struct StencilSelector<ArithOp> {
  static Stencil* Select(
    TypeId OperandType,
    AstArithmeticExprType operatorType,
    bool isLhsInRegister,
    bool spill,
    NumPassThroughs numPassThroughs);
}
```

Fig. 15. An example of the API between the stencil library and the copy-and-patch runtime. The left side is the stencil generator that generates the various stencil variants of the `ArithOp` stencil class. The API on the right side is automatically generated by the MetaVar compiler from the definitions on the left side and exposed to the copy-and-patch runtime.

major architectures including x86-64 [Matz et al. 2020], ARM [LinuxBase 1998], and SPARC [Oracle 2020a]. The rule yields the 3 patch vectors and the patch algorithm in Figure 10 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 10, and the API for the runtime to select the stencil from the hash map. The library is linked together with the copy-and-patch implementation to form the copy-and-patch runtime.

As a concrete example, Figure 12 illustrates the lifetime of the If-Leq (var-const) stencil (the green node in Figure 11b) used in the Fibonacci example. The stencil generator is a templated C++ function that generates various stencil variants for the logic shape `if (lhs op rhs)` where `op` is a comparison operator and `lhs` and `rhs` has a simple shape. The C++ template instantiation where `lhs` is an `int` local variable, `rhs` is an `int` constant, and `op` is `<=` yields the conceptual C++ logic of the stencil shown in Figure 12a. The logic contains four holes: the offset of the local variable in the stack frame, the constant literal, and two continuations for the true and false branches. Each hole is identified by an ordinal, so at runtime we can specify its desired value. Clang compiles the logic and generates the object code shown in Figure 12b, where the holes are identified by linker relocation records. The MetaVar compiler can figure out the ordinal of each hole in the object file, because the C macro trick we used to create holes associates each hole ordinal to an unique external variable, which translates to a unique symbol referenced by the linker relocation records. The MetaVar compiler parses the object file, and prints out the code shown in Figure 12c to construct the C++ Stencil struct (as defined in Figure 10) for this stencil. The code is compiled and becomes part of the stencil library. At runtime, when we compile the Fibonacci AST, this stencil is selected to implement the if-branch in the Fibonacci function, its holes are filled with concrete values (for example, the local variable `n` has offset `8` in the stack frame), and its tail jump instruction is removed and becomes a fallthrough to the continuation (the logic in the true branch), as shown in Figure 12d. This generates the eighteen bytes of executable code in `[0x20, 0x32)` of Figure 11c. Note that, most of the above work happens at library build time. The only work that happens during compilation at runtime is matching the AST with this stencil through a tree pattern matching, a hash table lookup to retrieve the stencil, a `memcpy` of those eighteen bytes, and a few scalar additions to patch the holes, which are all cheap operations.

Finally, we give an example of the API interface between the stencil library and the copy-and-patch code generation runtime in Figure 15. In the example, the stencil `ArithOp`'s C++ template parameters contain the type of the operands, the arithmetic operation kind, etc. MetaVar then creates an API, where the C++ template parameters are converted to function parameters. The

typename template parameter becomes a special TypeId enum, while all enum and boolean template parameters are converted unchanged. At runtime, copy-and-patch selects the stencil by calling the StencilSelector<ArithOp>::Select function and supplying the runtime-known configuration values. The Select function uses the parameters as the key to lookup the hash table and returns the stencil. The runtime may then invoke the copy-and-patch API described in Figure 10 to specify the desired values for the holes and generate executable code. The result is a flexible and powerful interface capable of handling the complex language features required for a high-level language.

For WebAssembly, since the bytecode instructions are very low-level and operate on a stack machine, we can further unify the APIs to a single interface for even faster code generation, as shown in Figure 16. The opCode, numInRegInts, numInRegFloats, spillOutput together determines the stencil variant, and the numSpilledInts, numSpilledFloats and opData determines the patch values. This lets copy-and-patch process most of the WebAssembly instructions (all except control instructions) through this single function API. And all this function does is to look into an array to retrieve the stencil, and then do copy-and-patch to instantiate this stencil to the destination address. There is not even a switch case on the opcode or on the register configuration. This is why our code generator runs so fast.

```
void WasmEmitStencil(
  uint8_t*& dstAddr,
  uint8_t opCode,
  uint32_t numInRegInts,
  uint32_t numInRegFloats,
  bool spillOutput,
  uint32_t numSpilledInts,
  uint32_t numSpilledFloats,
  uint64_t opData);
```

Fig. 16. The API that handles the codegen for most of the WebAssembly opcode.

6 EVALUATION: WEBASSEMBLY BASELINE COMPILER

We evaluate our claim that our copy-and-patch-based WebAssembly compiler achieves significantly lower startup delay and better execution performance compare with the conventional baseline compiler design, and that it narrows the execution performance gap between baseline compilers and optimizing compilers.

6.1 Methodology

The experiments are run on a single-socket 8-logical-core Intel i7-7700HQ CPU at 2.80GHz with turbo boost on, running Ubuntu 20.04. The machine has 32GB RAM: large enough so that nothing is swapped out. We evaluate the startup delay and execution performance of our compiler against eight WebAssembly compilers used in four industrial software products, including three baseline compilers and five optimizing compilers:

- Two compilers used in Google Chrome 81's V8 Engine: the Liftoff [Backes 2018] baseline compiler and the TurboFan [Backes 2018] optimizing compiler.
- Three compilers used in Wasmer 1.0 [Akbary et al. 2018]: the Wasmer SinglePass [Zhou 2018] baseline compiler, and two optimizing compilers named Wasmer Cranelift [Akbary 2018] and Wasmer LLVM [Lewycky 2018], using CraneLift [Alliance 2018] and LLVM [Lattner 2002] as backend respectively.
- Two compilers used in Wasmtime 0.26 [Gohman et al. 2018a]: the Lightbeam [Gohman 2018] baseline compiler and an optimizing compiler [Gohman et al. 2018b] using Cranelift as backend.
- One compiler used in WAVM [Scheidecker and Lin 2020]: an optimizing compiler using LLVM [Lattner 2002] as backend.

Wasmer LLVM and WAVM both use LLVM as backend, but Wasmer LLVM performs strictly worse than WAVM in both code generation time and execution time on all benchmarks we tested, so we removed Wasmer LLVM from the results. We are also unable to report the numbers for Wasmtime Lightbeam, because it reports an "unsupported" error or crashes on all benchmarks we tested. We

report code generation time and execution performance for copy-and-patch and the remaining six compilers. For in-browser compilers (Google Chrome's Liftoff and TurboFan), code generation time measures the time to execute Javascript "`new WebAssembly.Module(data)`", after the module is fully read into the array `data`, to avoid any overhead related to the disk, network, or Javascript. We use browser developer flags to select the desired WebAssembly compiler implementation, following the official instruction [Google 2019]. For the other non-browser WebAssembly compilers, we first read the whole WebAssembly module to memory to avoid any disk overhead, then measure the code generation time by timing the C API provided by the respective compiler that compiles a module to executable code. The measurement errors (fluctuations of performance across runs) are within a few percents except for Chrome, which has a fluctuation of up to 10 percents. However, such levels of fluctuations do not invalidate any of our claims. Some of the compilers support multi-threaded code generation. To make the performance results easier to understand, we limit all implementations to use only one CPU in the code generation time benchmarks. However, we note that WebAssembly is designed so that compilation is trivially parallelizable at function level, so our implementation can be easily extended to support multi-threaded compilation with the same scalability as the other compilers, and all conclusions should still hold in a multi-threaded environment. We additionally note that our single-threaded implementation is faster than all benchmark rivals using 8 CPUs.

We benchmarked the compilers on four benchmarks. Two of the benchmarks measure both the execution performance and the startup delay of the compilers:

- CoreMark 1.0 [EEMBC 2009], an industrial-standard benchmark to measure the performance of a CPU. We used the default random seed and settings of the benchmark. We confirmed that all compilers produced the correct final checksum.
- PolyBenchC 4.2.1 [Louis-Noel Pouchet 2011], the benchmark used in the original WebAssembly paper [Haas et al. 2017]. It consists of 30 numerical computation kernels extracted from operations in various application domains.

The other two benchmarks test the most important use case of baseline compilers: compiling real-world WebAssembly modules that are very large. We used the following real-world modules:

- AutoCAD Web App [AutoCAD 2018], a 47.5 MB WebAssembly module containing a 40.2 MB WebAssembly code section. The module is downloaded directly from the app website using its public URL. Notably, Liftoff also used this as a benchmark [Backes 2018].
- *clang.wasm* [Smith 2018], a 30.5 MB WebAssembly module containing a 27.5 MB WebAssembly code section. The module is downloaded from the official mirror.

For these two benchmarks, we only measure the code generation time, as there is not a clear criterion to quantitatively measure their execution performance. All benchmarks are executed three times and the average is reported.

6.2 Code Generation Performance

We measured the startup delay of all compilers on all benchmarks. The AutoCAD WebAssembly module appears to trigger a bug in LLVM, causing all LLVM-based compilers (WAVM and Wasmer LLVM) to enter a dead loop (did not finish in 4 hours). All other compilations are successful. Figure 17 shows the absolute throughput of each compiler, in terms of megabytes of WebAssembly code processed per second. Since the high throughput of copy-and-patch renders the throughput bar of many compilers barely visible, we plot the normalized log-scale startup delay (with copy-and-patch normalized to 1) in Figure 18.

As shown in Figure 18, the startup delay of our compiler is consistently lower than all the other compilers. Compared with baseline compilers, it is 4.9×–6.3× faster than Liftoff, and 13×–19.3×

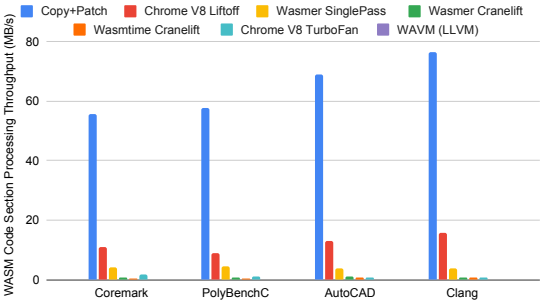


Fig. 17. The absolute throughput of each WebAssembly compiler, in terms of megabytes of WebAssembly code section processed per second. The PolyBenchC column records the average throughput of the 30 PolyBenchC benchmark modules. Higher is better.

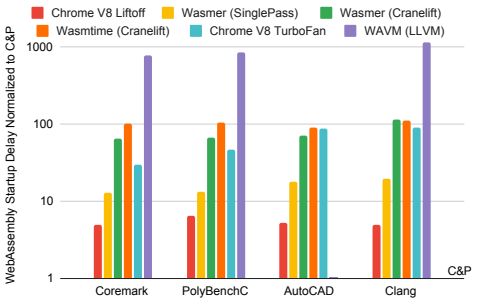


Fig. 18. The log-scale normalized startup delay of each WebAssembly compiler, with copy-and-patch normalized to 1. The PolyBenchC column records the average startup delay of the 30 PolyBenchC benchmark modules.

faster than Wasmer SinglePass. Compared with optimizing compilers, the startup delay is two to three orders of magnitudes lower.

The code generation throughput of our compiler on Coremark and PolybenchC is lower than on AutoCAD and Clang. This is because the code generation memory manager in our implementation takes about 0.2 ms to initialize. This cost is negligible for large modules, but the Coremark and PolyBenchC benchmark modules are small, containing only about 30 kB of WebAssembly code. Copy-and-Patch needs 0.6 ms to generate code for one such module, so the 0.2 ms initialization cost takes one third of the time.

Figure 19 demonstrates the throughput of machine code generated by copy-and-patch. As one can see, on large modules like AutoCAD and Clang, we are capable of generating more than 300MB of machine code per second using a single CPU. Such throughput is made possible by copy-and-patch's design that turns the task of code generation into, literally, copy (by memcpy) and patch (by a few scalar additions).

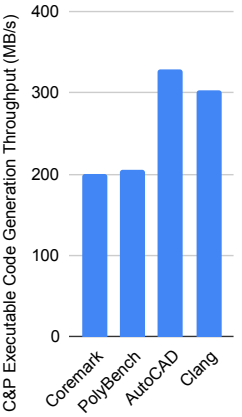


Fig. 19. Throughput of machine code generated by copy-and-patch (MB/s).

6.3 Execution Performance

Copy-and-Patch not only generates code fast, but also generates fast code. We measure the execution performance of all compilers on the Coremark and PolyBenchC benchmarks. Figure 20 reports the time to execute 10000 Coremark iterations. Figure 21 reports the normalized execution time of each of the 30 PolyBenchC benchmarks, with copy-and-patch's execution time normalized to 1. The average normalized execution time of PolyBenchC is reported in Figure 22.

As one can see, on Coremark and all 30 PolyBenchC benchmarks, copy-and-patch consistently performs better than all baseline compilers. Compared with Liftoff, we are 46% faster on Coremark, and on average 63% faster on PolyBenchC. The speedup compared with Wasmer SinglePass is similar. The performance of copy-and-patch is even comparable with one optimizing compiler (Wasmer Cranelift), being 2.4% slower on Coremark but 4.6% faster on PolyBenchC. Therefore, we conclude that copy-and-patch replaces the role of baseline compilers, and narrows the performance gap between baseline compilers and optimizing compilers.

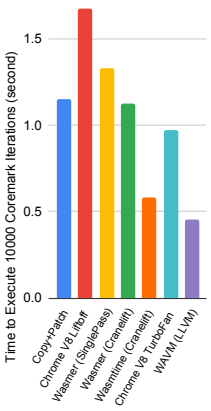


Fig. 20. WebAssembly exec time, Coremark.

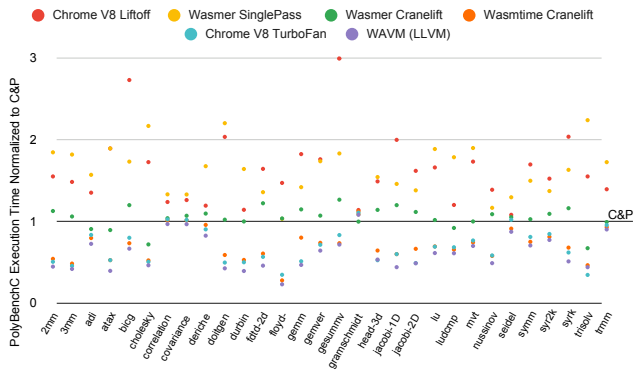


Fig. 21. WebAssembly normalized execution time on each of the 30 PolyBenchC benchmarks, normalized to copy-and-patch.

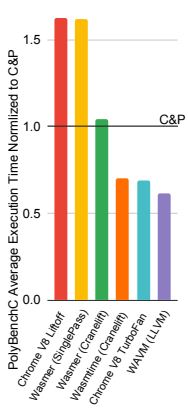


Fig. 22. Average of Figure 21.

6.4 Other Metrics

Our WebAssembly implementation contains 1666 stencils taking 35 kB of memory, which is built in less than a minute at compiler installation time. The stencil library is small because we did not implement supernodes (e.g., arithmetic operations taking a constant operand, branch if a relational operation is true). We anticipate that our execution performance could be further improved, at a cost of a larger stencil library and a slightly higher startup delay, if we had implemented supernodes.

7 EVALUATION: HIGH-LEVEL LANGUAGE COMPILER

In this section, we evaluate our C-like high-level language compiler built on top of copy-and-patch. The compiler is implemented as a domain-specific language (DSL) library embedded in the C++ host language. The language constructs are exposed to users as a set of C++ APIs (for example, `Declare`, `For`, `If`, `Call`), which build the user program AST behind the scene. Figure 11a demonstrates an example program in our DSL. Once a user program is built, it may be interpreted using the AST interpreter, compiled to binary code using copy-and-patch, or lowered to LLVM IR and then compiled to binary code using LLVM. The three backends are functionally equivalent and operate on the same input program. This allows an apple-to-apple comparison of the startup delay and execution performance tradeoffs presented by the techniques.

Our compiler is a general purpose compiler since it compiles a C-like language. Section 7.2 evaluates its performance on a few well known microbenchmark algorithms. However, one of the most important use cases of such a compiler is in SQL databases, where SQL queries are lowered to concrete program logic, then compiled to binary code using the compiler and executed. Therefore, we implemented a SQL query compiler prototype on top of our compiler, and evaluated its performance on TPC-H queries, as elaborated in Section 7.3.

In both cases, we evaluate our central claim that copy-and-patch replaces both LLVM `-O0` compilation and interpreters. Copy-and-Patch generates code that executes faster than `-O0` compilation and an order of magnitude faster than interpretation, with negligible startup delay. 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.

7.1 Methodology

The hardware environment is the same as described in Section 6.1. 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 generating 100 clones of the function inside the module and report 1/100 of total code generation time, to reflect the true time LLVM spent generating code. The measurement errors (fluctuations of performance across runs) are within a few percents.

We implemented three microbenchmarks and eight relational queries from the standard TPC-H database management system benchmark suite [TPC 2020]. 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 [Fontaine 2018; MemSQL 2020c]. 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.

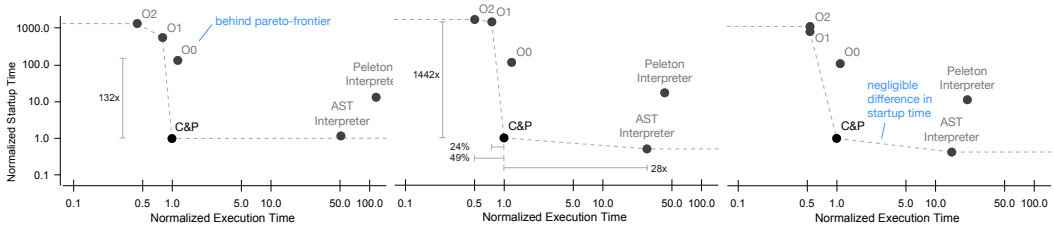
7.2 Microbenchmarks

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 delay 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 [Kost 2018] from the query execution engine of the Peloton database management system [Menon et al. 2017; Pavlo et al. 2017]. 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× faster than Peloton's interpreter.

We used these systems to execute three microbenchmarks and recorded their startup delay and execution time on synthetic input. The microbenchmarks include a function to compute Fibonacci numbers (Figure 11a), an implementation of Euler's sieve [Gries and Misra 1978], and an implementation of quicksort. The Fibonacci function is small with two recursive calls, demonstrating the efficiency of the generated code across calls. Euler's sieve is heavy on arithmetic 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 23 plots the startup delay 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 C&P. In all cases, LLVM `-O0` 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



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

Fig. 23. 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.

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 measure the startup and execution time of the microbenchmarks implemented in Java, using OpenJDK 11's Java bytecode interpreter [Center 2020] and HotSpot JIT [Oracle 2020b]. 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 interpreter's execution time is 4.4×–36× slower than C&P. The HotSpot JIT's compilation time, measured with JITWatch [Newland 2020], is from 40% faster to 20% slower than LLVM `-O3`, while its execution time is 20%–70% slower than LLVM `-O3`. Thus, the conclusion of our comparison between C&P and LLVM also holds for Hotspot JIT.

7.3 TPC-H Performance

A typical SQL query compiler, such as Hyper [Neumann 2011], PostgreSQL [PostgresSQL 2020], Peloton [Menon et al. 2017], or MemSQL [MemSQL 2020b], consists of three components: the SQL parser, the SQL query planner, and the plan executor. The SQL parser parses a user-provided SQL text to a SQL AST. The SQL query planner determines the most efficient plan to execute the SQL AST, and generates a query plan tree that describes the plan (e.g., join order, join method, operator order, and so on). Finally, the plan executor lowers the query plan tree to LLVM IR, then LLVM is invoked to compile the IR to binary code.

We have implemented such a database query compiler, but our query compiler lowers the input query plan tree to a program in our C-like language using our metaprogramming DSL library. The program may then be executed using the AST interpreter, the copy-and-patch backend, or the LLVM backend, letting us compare their performance on database workloads. Our query 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. However, we note two differences between our prototype and an industrial implementation.

First, since the SQL parser and the SQL query planner are independent from the code generation techniques, for the purpose of our benchmark, we did not implement these components. The input to our compiler is thus a hand-coded query plan tree, instead of a SQL query text. While the query plan nodes we supported should be sufficient to execute most of the TPC-H queries, many queries require complicated rewriting to yield a reasonable query plan tree, which requires a lot of labor that is unrelated to the main point of our evaluation. Therefore, we only implemented those 8 TPC-H queries whose execution plans follow directly from the query text.

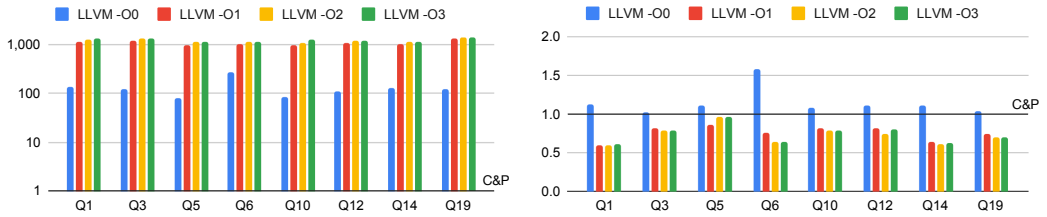


Fig. 24. Normalized startup delay (left) and execution time (right), as multiples of C&P, of code generated by the LLVM optimization levels across the TPC-H queries. C&P generates code two orders of magnitude faster (up to 267×) than LLVM `-O0` and three orders of magnitude (up to 1384×) faster than 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`.

Second, our implementation is simple, with only 600 lines of core logic (not counting comments, empty lines and curly braces) to call our DSL APIs and construct the program. Industry database management systems are much more complex, and generate much larger programs than we do 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. Therefore, conditioning on the same compilation technique, an industrial database needs much *more* time to compile a query than our prototype does, which further motivates fast compilation techniques. We validated this using the MemSQL product trial: the TPC-H queries take 4.8×–52× (average 16.4×) longer to compile than our query compiler when both are using LLVM `-O3`, and one 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 24 (left) shows the compilation time of each LLVM optimization level and Figure 24 (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 7.2 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 25 (right). As shown in Figure 25 (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.

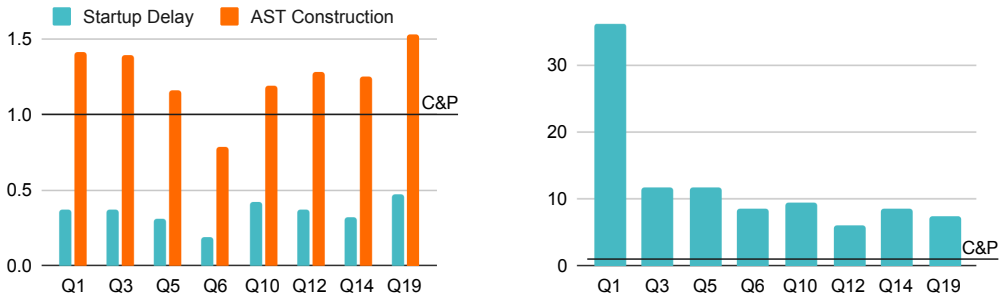


Fig. 25. Normalized startup delay (left) and execution time (right) of the AST interpreter across the TPC-H queries, as multiples of C&P. The interpreter has less than half the startup delay of C&P, but both costs are negligible as it takes longer to construct the AST. And the interpreter executes 6–36 times slower than C&P.

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.

7.4 Copy-and-Patch Scalability

As a baseline compiler, the copy-and-patch algorithm runs in linear time, requiring only two traversals of the AST and one traversal of the CPS call graph. Optimizing compilers like LLVM, on the other hand, contain non-linear algorithms. Figure 26 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` bogs 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.

7.5 Copy-and-Patch Optimization Breakdown

Copy-and-Patch employs several optimizations to produce fast code. Figure 27 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 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 supernodes, 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.

7.6 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

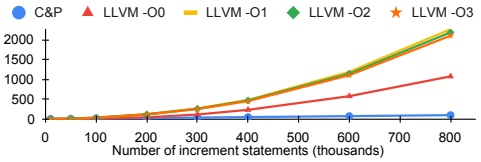


Fig. 26. The normalized startup delay of C&P and LLVM optimization levels as the size of the input program increases. To demonstrate scalability, the time it took each algorithm to compile 10k statements is normalized to 1, so perfect scaling line would end at y-axis 80. C&P scales near-linearly (ending at y-axis 98), while the other lines show worse scalability.

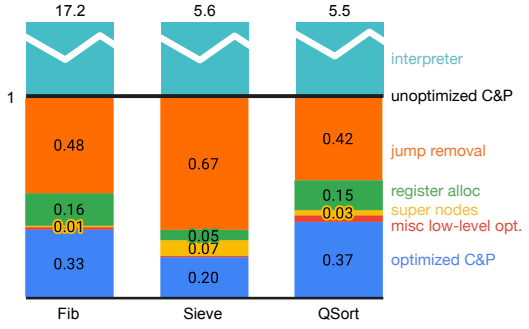


Fig. 27. Normalized microbenchmark running times with optimized C&P in dark blue. Each bar on top of that shows the running time added by removing each optimization.

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.

8 RELATED WORK

We compare the copy-and-patch algorithm and our MetaVar system with work in three areas: baseline compilers, staged compilation that generates binary code by symbolically executing an interpreter, and build-time code library generation approaches that contrast to MetaVar.

Depending on how the code is generated, baseline compilers can be categorized into two classes: baseline compilers that use platform-dependent machine instruction assemblers to generate code, and baseline compilers that generates code by concatenating pre-compiled binary snippets. The term "template JITs" is often used to refer to both, but for the purpose of distinguishing them, we will call the former "assembler-based baseline compiler" and only the latter "template JITs".

Template JITs

Template JIT is the technique that is most similar to copy-and-patch. A template JIT uses an optimizing compiler (e.g., C, Java) to generate binary snippets at build time, and at runtime generate code by concatenating those binary snippets. Examples include [Ertl and Gregg 2003; Iliasov 2003; Piumarta and Riccardi 1998; Wimmer et al. 2013]. The Maxine compiler [Wimmer et al. 2013] generates snippets from Java functions, while the other systems generate snippets from C functions.

There are three major differences between copy-and-patch and the template JIT techniques described above. First, copy-and-patch has a patching phase. None of the papers above supported patching the binary code to burn in literals, stack offsets, and jump addresses, so their technique only works if the binary code can be concatenated without modification. This implies that all jumps and calls are indirect, and that all constants must be retrieved from memory, resulting in inferior execution performance. Second, copy-and-patch has the concept of stencil variants, which allows copy-and-patch to not only generate higher-quality code by selecting the most matching variant (e.g., adding a value in register with a constant), but also perform optimizations like register allocation and super-instructions, thus improving execution performance further. Third, all above papers target low-level bytecode assembling. In contrast, the use of a CPS call graph, and the flexibility of the binary stencils allow copy-and-patch to be used not only for low-level bytecode assembling, but also for high-level language compilation.

Assembler-Based Baseline Compilers

Using a machine code assembler to generate code is another approach to baseline compilers. The baseline compiler decide what assembly instruction to emit, and then the machine code assembler assembles it to machine code. Baseline compilers like the Google Chrome Liftoff [Backes 2018], the Firefox WebAssembly baseline compiler [Clark 2018], and Wasmer SinglePass [Zhou 2018] all employ this approach. Unsurprisingly, a lot of work has been done to improve the performance of machine code assemblers. VCODE [Engler 1996] 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 [Kobalicek 2014], DynASM for Lua [Pall 1999], and as a case study in Terra [DeVito et al. 2014]. The DCG system [Engler and Proebsting 1994] also attempts greedy register allocation, but only works under the unrealistic assumption that no spilling is ever needed, and also runs 35 times slower than VCODE [Engler 1996].

In the world of bytecode assemblers, copy-and-patch has two advantages compared with baseline compilers using a machine code assembler. First, as shown in Section 6, copy-and-patch not only emits code faster, but also emits faster code. Second, since the stencils are generated by Clang, we don't need to figure out what assembly instruction to use, thus reducing the engineering cost.

In the world of full compilers, the benefit of our approach is reflected in both engineering cost and performance. When the task is to lower a high-level program to machine code, assembling the instructions is not the most difficult part. Deciding *what* to assemble is. We need optimization, assembly instruction selection, and register allocation to produce high quality code. A machine code assembler cannot provide any of these. Copy-and-Patch solves the problem by using Clang to generate the AST node stencils. 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, and is 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.

Techniques Originated in Other Areas

Continuation-passing style [Steele 1977], which is similar to threaded code [Bell 1973], is originally used to optimize an interpreter's performance, as well as in compilation of functional languages. We use this technique 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 [Casey et al. 2003; Proebsting 1995] is a well-known technique to reduce indirect jump overhead between interpreter opcodes. Our supernode is similar to superinstruction; but in our use case, since unnecessary jumps between opcodes are already eliminated, supernodes are only a modest optimization to improve the quality of generated code.

The idea of using external variables to locate holes to burn in runtime constants is used by Noel et al. [1998]. However, their use case is runtime specialization of statically known logic and cannot be used in our use case where the logic is generated at runtime. Their technique is also more verbose than ours. QEMU [Bellard 2005] also used this trick to translate CPU instructions between architectures, but requires non-standard GCC extensions that has since been removed.

Staged Compilation and Dynamic Specialization

Staged compilation [Consel et al. 1998; Thibault et al. 2000] and dynamic specialization [Finkel et al. 2019], 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 [Frigo and Johnson 1998] 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.

9 CONCLUSION

We introduced copy-and-patch, a novel compilation technique that generates decent executable code at a negligible compilation cost. We envision copy-and-patch used in domains where fast compilation is important, notably, as the baseline compilation tier in JIT compilers. We empirically evaluated its potential in two such domains: SQL query compilation and WebAssembly compilation. In both domains, we demonstrated that copy-and-patch significantly outperforms existing techniques and approaches for fast compilation, including the current state-of-the-art baseline compiler implementations (V8 Liftoff and Wasmer Singlepass), LLVM `-O0` compilation, and interpreters.

The copy-and-patch algorithm and the stencil generator system are extensible by design. New AST nodes can be added by users seeking better performance for a new, perhaps domain-specific, language construct. And new supernode stencil generators can be added for better local optimization. The type system of the AST can also be expanded in future work to include vector types to target vectorized instructions. Therefore, we believe copy-and-patch will find a place in domains other than SQL query compilation and WebAssembly compilation as well.

We envision two areas of future work. First, new general-purposed optimizations can be implemented, such as common subexpression elimination, loop unrolling, and vectorization. Of course, these will increase compilation time, but will do so starting from a lower starting point. Second, copy-and-patch can be combined with domain-specific optimization techniques, most notably, the type profiling, type speculation, and inline caching techniques that optimize the performance of dynamic-typed languages. With these optimizations, we believe copy-and-patch can also be used as a fast profiling tier or even a fast optimizing tier in dynamic language JIT engines.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their comments that helped us improve this manuscript. We would also like to thank Alex Aiken, Saman Amarasinghe, Saam Barati, Ajay Brahmakshatriya, Cheng Chen, Stephen Chou, David Durst, Slava Egorov, Lang Hames, Pat Hanrahan, Scott Kovach, Richard Peng, Zhou Sun, Leszek Swirski, and Yinzhan Xu for helpful comments, review, and references. This work was supported by the Stanford Agile Hardware Center.

REFERENCES

Syrus Akbary. 2018. *Wasmer Cranelift backend.* Wasmer. https://github.com/wasmerio/wasmer/tree/master/lib/compiler-cranelift

Syrus Akbary, Ivan Enderlin, Mark McCaskey, Nick Lewycky, Heyang Zhou, Brandon Fish, Lachlan Sneff, and Mackenzie Clark. 2018. *Wasmer: The leading WebAssembly Runtime supporting WASI and Emscripten.* Wasmer Inc. https://wasmer.io/

Bytecode Alliance. 2018. *Cranelift Code Generator.* Bytecode Alliance. https://github.com/bytecodealliance/cranelift

AutoCAD. 2018. *AutoCAD Web App.* AutoCAD. https://web.autocad.com/

Clemens Backes. 2018. *Liftoff: a new baseline compiler for WebAssembly in V8.* Google. https://v8.dev/blog/liftoff

JF Bastien, Keith Miller, and Saam Barati. 2017. *Assembling WebAssembly.* Safari. https://webkit.org/blog/7691/webassembly/

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

Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *2005 USENIX Annual Technical Conference (USENIX ATC 05).* USENIX Association, Anaheim, CA. https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator

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.

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.

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

Lin Clark. 2018. *Making WebAssembly even faster: Firefox's new streaming and tiering compiler.* Mozilla. https://hacks.mozilla.org/2018/01/making-webassembly-even-faster-firefoxs-new-streaming-and-tiering-compiler/

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.

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* (Seattle, Washington, USA). ACM, New York, NY, USA, 105–116.

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* (Edinburgh, United Kingdom). Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/2594291.2594307

EEMBC. 2009. *CoreMark Benchmark.* EEMBC. https://www.eembc.org/coremark/

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

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

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

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

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/

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

GHC and LLVM. 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

Dan Gohman. 2018. *Introducing Lightbeam: An Optimising Streaming WebAssembly Compiler.* Bytecode Alliance. http://troubles.md/posts/lightbeam/

Dan Gohman, Pat Hickey, Alex Crichton, Andrew Brown, Benjamin Bouvier, and Nick Fitzgerald. 2018a. *WasmTime: A small and efficient runtime for WebAssembly & WASI.* Bytecode Alliance. https://wasmtime.dev/

Dan Gohman, Pat Hickey, Alex Crichton, Andrew Brown, Benjamin Bouvier, and Nick Fitzgerald. 2018b. *WasmTime Cranelift Compiler.* Bytecode Alliance. https://github.com/bytecodealliance/wasmtime/tree/main/crates/cranelift

Google. 2019. *WebAssembly compilation pipeline.* Google. https://v8.dev/docs/wasm-compilation-pipeline

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

W3C Community Group. 2017. *WebAssembly 1.0 Core Specification.* W3C Community Group. https://webassembly.github.io/spec/core/

W3C Community Group. 2018. *https://github.com/WebAssembly/WASI/blob/main/phases/snapshot/docs.md.* W3C Community Group. https://github.com/WebAssembly/WASI/blob/main/phases/snapshot/docs.md

Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363

Alex Iliasov. 2003. Templates-Based Portable Just-in-Time Compiler. *SIGPLAN Not.* 38, 8 (Aug. 2003), 37–43. https://doi.org/10.1145/944579.944588

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.

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

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

Chris Lattner. 2002. *LLVM: An Infrastructure for Multi-Stage Optimization.* Master's thesis. Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL. *See* `http://llvm.cs.uiuc.edu`.

Nick Lewycky. 2018. *Wasmer LLVM backend.* Wasmer. https://github.com/wasmerio/wasmer/tree/master/lib/compiler-llvm

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

Tomofumi Yuki Louis-Noel Pouchet. 2011. *PolyBenchC Benchmark.* Ohio State University. https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1

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

MemSQL. 2020a. *MemSQL Database.* MemSQL. https://www.memsql.com

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

MemSQL. 2020c. *Personal communication, with permission to disclose to the public.* MemSQL.

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

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

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

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.

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

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

Mike Pall. 1999. *LuaJIT DynASM.* The LuaJIT Project. https://luajit.org/dynasm.html

Andrew Pavlo. 2021. *Database of Databases.* Carnegie Mellon Database Group. https://dbdb.io/

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.

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* (Montreal, Quebec, Canada). Association for Computing Machinery, New York, NY, USA, 291–300. https://doi.org/10.1145/277650.277743

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

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.

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.

Andrew Scheidecker and Wanming Lin. 2020. *WebAssembly Virtual Machine.* WAVM. https://github.com/WAVM/WAVM

Ravi Sethi and J. D. Ullman. 1970. The Generation of Optimal Code for Arithmetic Expressions. *J. ACM* 17, 4 (Oct. 1970), 715–728. https://doi.org/10.1145/321607.321620

Ben Smith. 2018. *Clang in WebAssembly*. WAPM. https://github.com/wapm-packages/clang

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* (Seattle, Washington) *(ACM '77)*. ACM, New York, NY, USA, 153–162. https://doi.org/10.1145/800179.810196

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.

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

Wikipedia. 2021. *Simple Sethi-Ullman Algorithm*. Wikipedia. https://en.wikipedia.org/wiki/Sethi%E2%80%93Ullman_algorithm

Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. 2013. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Trans. Archit. Code Optim.* 9, 4, Article 30 (Jan. 2013), 24 pages. https://doi.org/10.1145/2400682.2400689

Andy Wingo. 2020. *firefox's low-latency webassembly compiler*. Mozilla. https://wingolog.org/archives/2020/03/25/firefoxs-low-latency-webassembly-compiler

Heyang Zhou. 2018. *Wasmer Singlepass Backend*. Wasmer. https://github.com/wasmerio/wasmer/tree/master/lib/compiler-singlepass