Copy-and-Patch Compilation

Haoran Xu and Fredrik Kjolstad

Stanford University

The Need For Fast Compilation

* Many modern applications require compilation at runtime.
  - database engine: SQL query $\rightarrow$ machine code
  - web browser: WebAssembly module $\rightarrow$ machine code
  - and many more...

The Need For Fast Compilation

* The delay experienced by user is compilation time + execution time.
* A traditional "very optimizing, but very slow" compiler is not enough.
  - LLVM -O3: 4.5s to compile TPC-H Q19. [MemSQL 2020]
  - Google Chrome TurboFan: 51 CPU seconds to compile the WebAssembly module powering the AutoCAD Online App.
  - But a 200ms latency can already be perceived by users and cause them to visit a webpage less frequently [Google 2009].
* Need a fast compiler to get the code started quickly.
* Two-tier strategy: tier 1 fast compiler, tier 2 slow optimizing compiler. Employed in all browsers and many industrial databases.

The Need For Fast Compilation

* The delay experienced by user is compilation time + execution time.
* A traditional "very optimizing, but very slow" compiler is not enough.
  - LLVM -O3: 4.5s to compile TPC-H Q19. [MemSQL 2020]
  - Google Chrome TurboFan: 51 CPU seconds to compile the WebAssembly module powering the AutoCAD Online App.
  - But a 200ms latency can already be perceived by users and cause them to visit a webpage less frequently [Google 2009].
* Need a fast compiler to get the code started quickly.
* Two-tier strategy: tier 1 fast compiler, tier 2 slow optimizing compiler. Employed in all browsers and many industrial databases.

The Need For Fast Compilation

* The delay experienced by user is compilation time + execution time.
* A traditional "very optimizing, but very slow" compiler is not enough.
  - LLVM -O3: 4.5s to compile TPC-H Q19. [MemSQL 2020]
  - Google Chrome TurboFan: 51 CPU seconds to compile the WebAssembly module powering the AutoCAD Online App.
  - But a 200ms latency can already be perceived by users and cause them to visit a webpage less frequently [Google 2009].
* Need a fast compiler to get the code started quickly.
* Two-tier strategy: tier 1 fast compiler, tier 2 slow optimizing compiler. Employed in all browsers and many industrial databases.

Existing Approach for Tier 1: In Databases

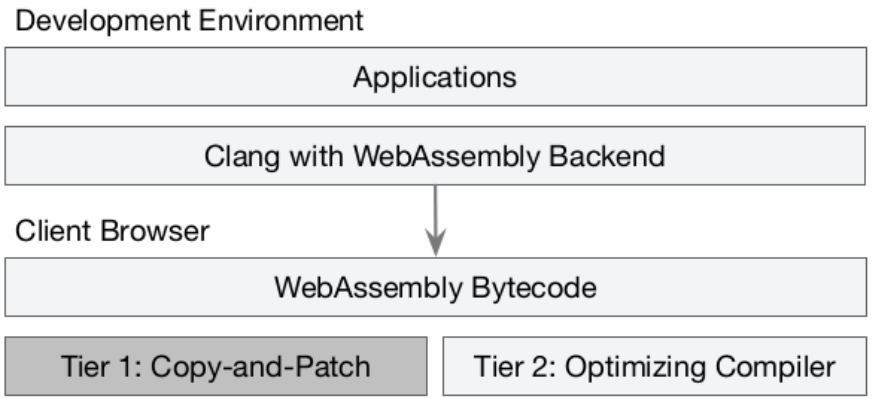
* Tier 1: Interpreter or LLVM -O0. Tier 2: LLVM -O3.
* Standard approach for databases: PostgresSQL, MemSQL, PeletonDB, etc..
* Advantage:
  - Relatively little extra engineering work. LLVM -O0 requires no additional work, and interpreter is needed for testing anyway.
* Disadvantage:
  - An interpreter is too slow.
  - The startup delay of LLVM -O0 is still high.

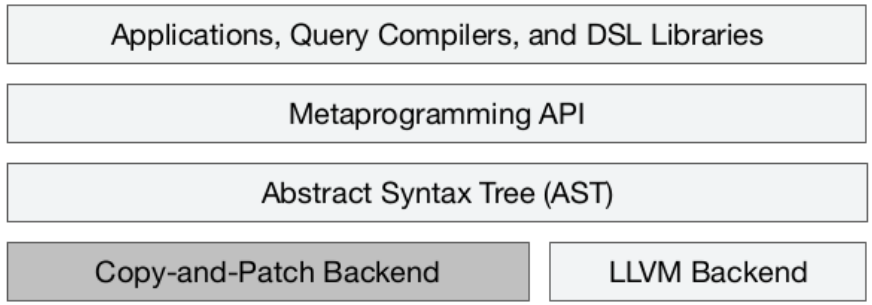
Existing Approach for Tier 1: In WebAssembly

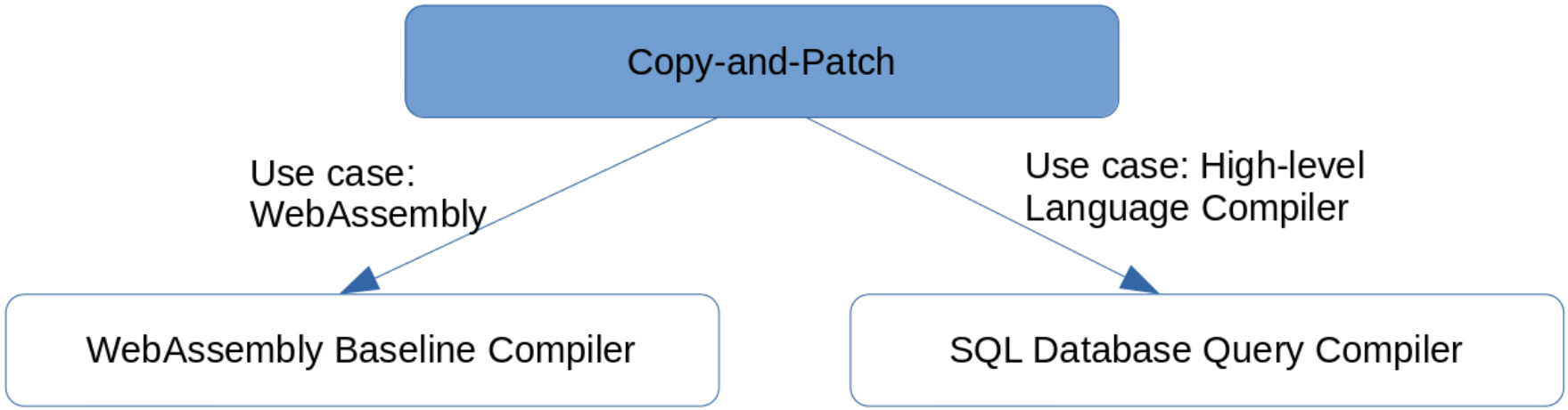
* Tier 1: Dedicated baseline compiler. Tier 2: Optimizing compiler.
* Standard approach for WebAssembly engine in both browsers and non-web runtimes: Chrome, Firefox, Wasmer, Wasmtime, etc..
* Advantage:
  - Much lower startup delay than LLVM -O0: the baseline compiler only moves through the WebAssembly bytecode once, and as it moves, emits machine code using a machine code assembler.
* Disadvantage:
  - Can only assemble low-level bytecode. Much harder to do this for high-level languages (which is why no databases use this approach).
  - High amount of engineering work, platform dependent, etc.

The Copy-and-Patch Approach

* Our solution: copy-and-patch.
* Can be used for **both** high-level language compilation (e.g. for databases) **and** low-level bytecode assembling (e.g. for WebAssembly).
* Achieves **both** much lower startup delay **and** better execution performance than LLVM -O0 and dedicated baseline compilers (e.g. Google Chrome's Liftoff baseline compiler).







	Compilation Time	Execution Time
Google Chrome Liftoff (Baseline Compiler)	4.9 – 6.5	1.46 - 1.63
Google Chrome TurboFan (Optimizing Compiler)	30 – 47 (small module) 88 – 91 (large module)	0.69 - 0.85

* Benchmarks: PolyBenchC, Coremark, AutoCAD, Clang.wasm.

	Compilation Time	Execution Time
Interpreter	0.3 - 0.5	6 - 36
LLVM -O0	79 - 267	1.02 - 1.57
LLVM -O1 or higher	936 - 1384	0.61 - 0.96

* Benchmarks: TPC-H queries.

The Pareto Frontier

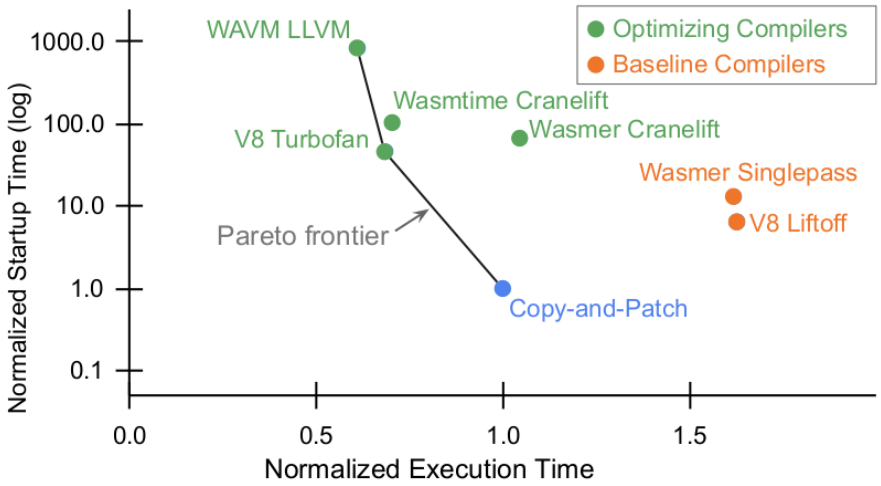


Fig. 2. A scatter plot of normalized startup times against execution times for seven WebAssembly compilers averaged over the PolyBench benchmarks. Our copy-and-patch compiler replaces baseline compilers on the Pareto frontier.

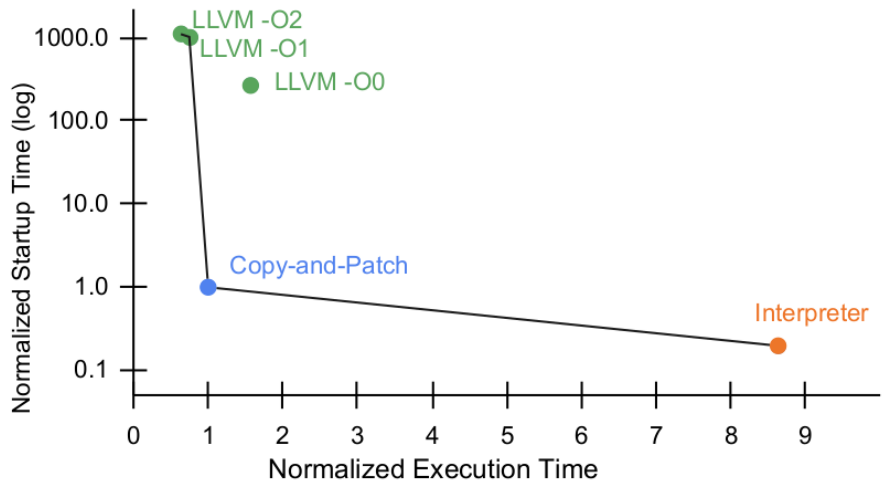


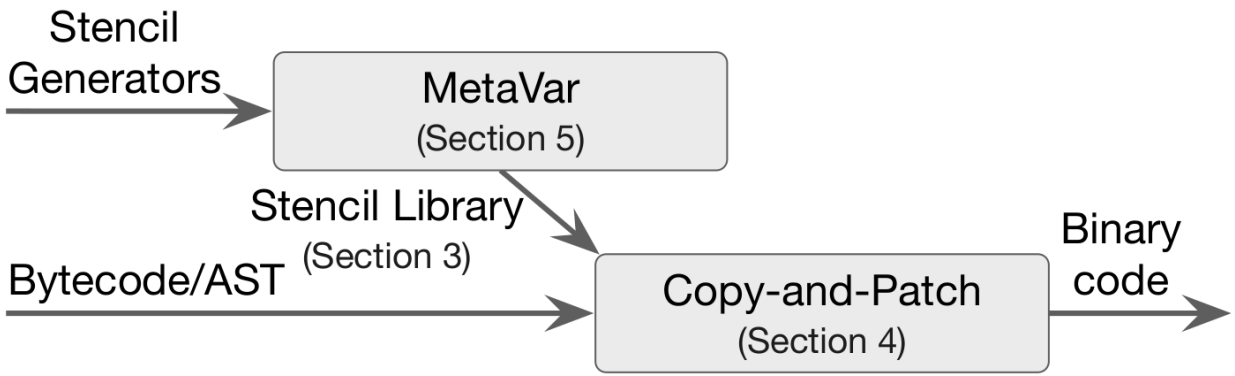
Fig. 3. A scatter plot of normalized startup execution times against execution times for five strategies for executing the sixth TPC-H query implemented in a high-level language. Our copy-and-patch compiler replaces LLVM `-O0` on the Pareto frontier.

Intuition of Copy-and-Patch

Core idea: code library.

At build time, build a library of composable and configurable code snippets (which we call binary stencils) implementing various language constructs.

At runtime, select the desired snippets, compose them together by copying them to continuous memory, and finally configure them by patching in missing values.



Why it generates code fast?

* With copy-and-patch, the task of code generation becomes the simple task of selecting a stencil by looking up a data table, then perform copy (by memcpy) and patch (by a few scalar additions).
* As an example, in our WebAssembly compiler, we don't even need a switch-case on the opcode (except a few special opcodes).
* The algorithm does not know or care about the machine instructions being used, the machine instruction encoding, the register encoding, etc. Everything just becomes data that we pre-built at build time.

Why it generates fast code?

* The patch phase burns literals, stack offsets, jump addresses, etc into the generated code, just like an optimizing compiler.
* We prepare many stencil variants for each AST node type or bytecode opcode. At runtime, we can pick the most matching stencil for best performance. For example, addition in which none/one/both side residing in register, whether the output is stored in register or spilled, addition with a constant literal, etc. We also perform a register allocation pass to store temporaries in registers as much as possible.
* We systematically generate many supernodes for common operations, e.g. adding a local variable with another local variable, array indexing, simple if-branch conditions, etc., for even better quality of generated code.

Binary Stencils

Think of a stencil as an analogue to a C++ templated function.

Example: a function which adds an input value by a fixed constant, then calls a fixed function to pass over the result.

```
template<int y, void(*g)(int)>
void f(int x) {
    int result = x + y;
    g(result);
}
```

A stencil is analogous to a templated function above, except that its template parameters can be specified at runtime, and instantiating it is extremely fast. Filling holes to get executable code from a stencil is analogous to instantiating the template by specifying its template parameters.

So...

How can we get functions with holes?

The core of the trick

Consider the following function:

```
extern int evaluate_lhs();
extern int evaluate_rhs();
int evaluate()
{
    int lhs = evaluate_lhs();
    int rhs = evaluate_rhs();
    return lhs + rhs;
}
```

When we compile this code...

* C++ compiler would generate an object file for us.
* The linker can link this object file to *any* definition of `evaluate_lhs` and `evaluate_rhs` and the resulted executable would work just fine.
* So... The object file must contains structured information to tell the linker how to wire the two callees to its actual definitions.
* The technical term for this wiring process is called symbol relocation.

Relocation Is Code Generation

* If we parse the symbol relocation records of the object file, we can then act as the linker ourselves, and wire `evaluate_lhs` and `evaluate_rhs` to any function of our choice.
* What is the cost? A memcpy (to make a copy of the binary code) followed by a few 32/64-bit scalar addition to modify the binary code at a few pre-determined places according to the relocation records.

Can we do better?

* In the above example, the two `evaluate_lhs` and `evaluate_rhs` are function calls (`call` instruction).
* An optimizing compiler won't emit two calls, in fact not even any branches, to compute a "addition" expression. Can we do that as well?
* We can use continuation-passing style to turn them into tail calls, thus being optimized by clang into `jmp` instructions.
* And since we have full control over where the stencils are materialized, if we put the stencil for the target of the `jmp` right after the caller stencil, we can actually eliminate this `jmp` to a fallthrough. See paper for detail.
* The result is that the generated code only contains branches and calls when actually needed, just like an optimizing compiler.

More extensions

* Runtime constants can be burnt in using a similar trick: the address of a symbol gives us 64 bits of 0s and 1s, which we can use to represent a constant of a given type.
* The function prototype can be used as a register allocation scheme: to put something in the argument is to put something in register (we need GHC calling convention to make sure all parameters are in register). This allows us to pass temporaries in register between stencils for better performance: a simple register allocation scheme by selecting different stencil variants. See paper for detail.

Automatic Generation of Stencils

* A large number of stencil variants are needed to implement different kinds of AST nodes, based on operand types, input locations (memory/register), operand shapes (similar to super-instruction) etc.
* We use a concise system grounded on C++ template metaprogramming to systematically generate all stencils. To be covered later.
* User can extend this system to add new stencils and AST nodes, e.g. various SQL scalar operators, for better performance than an external function call.

Other Optimizations

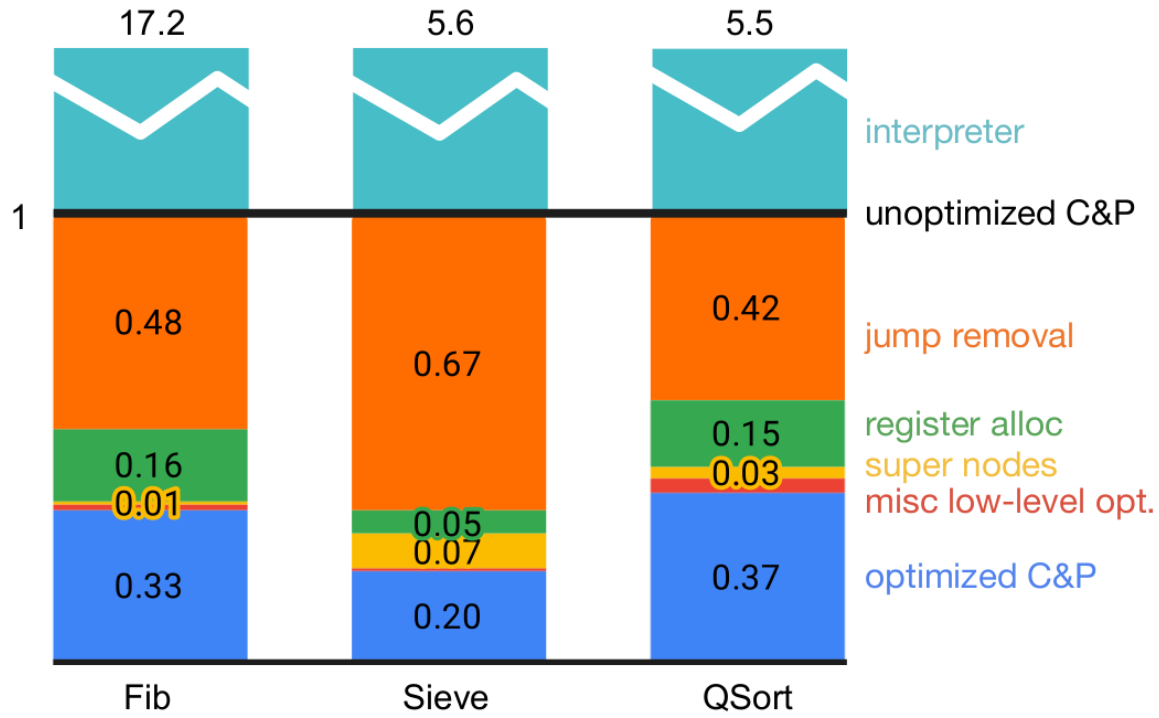


Fig. 25. Normalized microbenchmark running times with optimized C&P in dark blue. Each bar on top of that shows the running time added by removing each optimization.

So compared with an optimizing compiler, copy-and-patch supports:

* Locally optimized code (stencils generated by optimizing compiler, also has supernodes for common operations).
* Reasonably good register allocation (uses a greedy, but globally optimized register allocation algorithm).
* Reasonably good control flow (no unnecessary jumps/calls).

and only lacks:

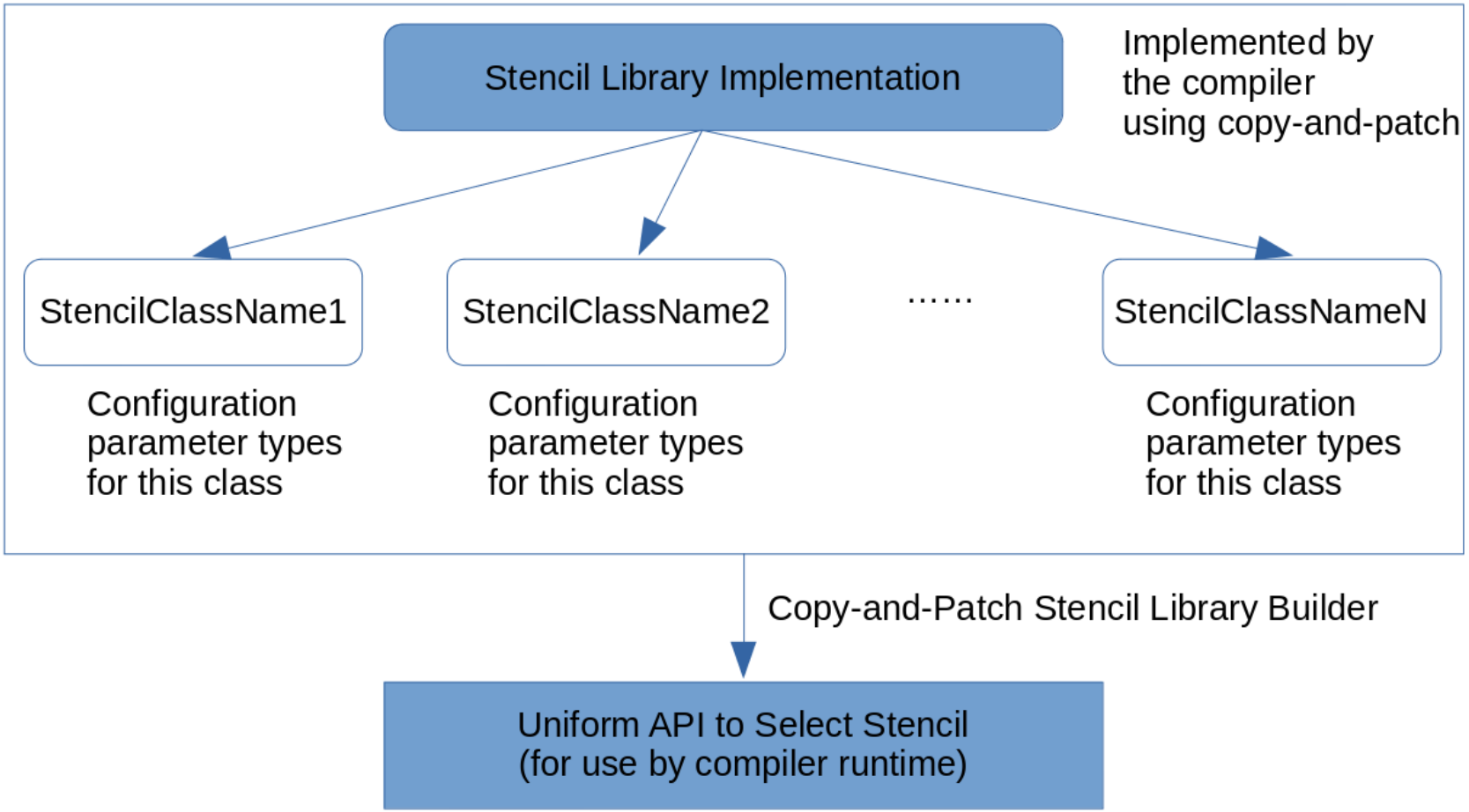
* Global optimization (e.g. various rewrite passes on the input program).

It turns out that this is already enough to beat LLVM -O0 by 15% (database use case), and baseline compilers by 46% - 63% (WebAssembly use case).

The Stencil Library Construction Framework

* To generate high quality code, we prepare many stencils. Our high level language uses about 100K stencils (17MB, most of them are supernodes). Our WebAssembly compiler uses about 1700 stencils (35kB, since we did not implement supernode), but that's still too many to implement by hand.
* We generate them automatically using a concise C++ system.

The Stencil Library Construction Framework



Stencil Class

* Each stencil class configuration is a list of elements, with each element being:
  - A C++ primitive type (void, int, int*, etc).
  - A C++ enum type.
  - A boolean.
* The implementation must then provide two C++ templated functions f and g, with the template parameter list matching the configuration.
* f is a constexpr function taking no parameters and returning a boolean, denoting if this configuration is valid or not.
* g is a function that must be instantiatable if f returns true on a template parameter configuration. g is the implementation of the stencil.

Example

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

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

Fig. 13. An example of the API between the stencil library and the copy-and-patch runtime. The API on the right side is generated by the MetaVar compiler and exposed to the copy-and-patch runtime.

Runtime can then use something like the following to select a stencil:

```
Stencil* s = StencilSelector<ArithAdd>::Select(
    TypeId::Get<int>(),
    static_cast<NumPassThroughs>(0),
    AstArithmeticExprType::ADD,
    false /*isLhsInRegister*/,
    false /*spillOutput*/);
```

Stencil Class Example

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

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

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

Stencil Class Example

* Inside the implementations, special macros like DEF_CONTINUATION_0(....), DEF_CONSTANT_1(uint64_t) define the holes. Each hole is also given an ordinal as shown in the macro.
* The macro defines a constant variable of specified type that can be normally used within the implementation. Internally, it uses the external variable trick we explained earlier.
* At runtime, after selecting the stencil, one can call API

  ```
  s->PopulateHole(holeOrd, value);
  ```

  to specify the desired value for those holes.

The Stencil Library

So at runtime, basically the workflow is:

* The compiler writer specifies the desired stencil class and stencil configuration, then select the stencil using the StencilSelector API.
* The compiler writer uses the PopulateHole API to populate desired values for each hole of the stencil, which can be literal values or jump/call targets. For jump/call targets, the user provides a pointer to another stencil, to which the control flow will transfer.
* After all stencils are selected and holes populated, call the "Codegen" API to generate code.

Stencil Library Conclusion Thoughts

* The result is a flexible, powerful and fast API to generate code using the stencil library system.
* With this system we can generate code for both low-level bytecode, and complex high-level languages with many language features.
* We did not cover how the library is built behind the scene from the user-provided stencil implementations and configuration specifications, but one can imagine this is clearly doable, just engineering work. See paper for detail.

Application Layers

* We have built a full metaprogramming system on top of copy-and-patch.
* Implemented as a C++ library that exposes a C-like language called Pochi.
* Pochi supports most of the C language constructs, ability to use C++ classes and call C++ functions in host code, C++ exception handling, and more, in an API syntax that closely mimics C++.
* We plan to present it in a future paper.

Pochi Example

```
1  Function* regexfn = codegen("ab.d*e");
2  using Regexs = int(*)(vector<string>*);
3  auto [regexs, inputs] = newFunction<Regexs>("regexs");
4  auto result = regexs.newVariable<int>();
5  auto it = regexs.newVariable<vector<string>::iterator>();
6  regexs.setBody(
7    Declare(result, 0),
8    For(Declare(it, inputs->begin()),
9        it != inputs->end(),
10       it++                                  Pochi loop iterates over
11   ).Do(                                     a C++ STL iterator
12     result += StaticCast<int>(
13        Call<RegexFn>(regexfn, it->c_str()))
14   ),
15   Return(result)
16 );
17
18 vector<string> input {"abcde", "abcdde",  // good input
19                       "abde" , "abcdef"}; // bad input
20 buildModule();
21 Regexs match = getFunction<Regexs>("regexs");
22 assert(match(&input) == 2);
```

```
1  using RegexFn = bool(*)(char* /*input*/);
2  Function* codegen(const char* regex) {
3    auto [regexfn, input] = newFunction<RegexFn>();
4    if (regex[0] == '\0') {
5      regexfn.setBody(
6        Return(*input == '\0')
7      );
8    } else if (regex[1] == '*') {
9      regexfn.setBody(
10       While(*input == regex[0]).Do(
11         input++,
12         If (Call<RegexFn>(codegen(regex+2), input)).Then(
13           Return(true)
14         )
15       ),
16       Return(false)
17     );
18   } else if (regex[0] == '.') {
19     regexfn.setBody(
20       Return(*input != '\0' &&
21              Call<RegexFn>(codegen(regex+1), input+1))
22     );
23   } else {                    Pochi test on
24     regexfn.setBody(          runtime regex
25       Return(*input == *regex &&
26              Call<RegexFn>(codegen(regex+1), input+1))
27     );
28   }
29   return regexfn;
30 }
```

Links

Paper: https://arxiv.org/abs/2011.13127

Code: https://github.com/sillycross/PochiVM

(The above paper version does not include the WebAssembly compiler.)