A Copy-and-Patch JIT for Threaded Interpreters

Nishil Kapadia and Dave Mason

Toronto Metropolitan University

Code Generation Strategies

- AoT - Ahead of Time
  - can produce excellent results because can grind
  - difficult with dynamic languages
  - lots of available technologies, but a lot of work to get quality results
- Interpreter/Threaded Execution
  - easier to write, but much poorer performance
- JIT - Just in Time
  - can work well for dynamic languages because we can know dynamic types
  - lot of work to get quality results
- CnP - Copy and Patch
  - intermediate quality
  - less work

Threaded Execution

- A compiled method is a sequence of threaded code entries and operands.
- Each threaded code entry points to a threaded function.
- Threaded functions share a common VM interface: PC, SP, Process, Context, Extra.

```
pub fn someThreadedFn(pc: PC, sp: SP, process: *Process, context: *Context, sig:
    Signature) SP {
    return @call(tailCall, ...);
}
```

- Execution advances by tail-calling the next threaded function.
- This keeps the VM simple, but repeated indirect transfers add hardware instruction dispatch overhead.

Threaded Execution

```
PUSH_LITERAL 1
PUSH_LITERAL 2
ADD
STORE_LOCAL x
```

(a) Bytecode Interpretation

```
&pushLiteral, 1
&pushLiteral, 2
&add
&storeLocal, x
```

(b) Threaded Code Representation

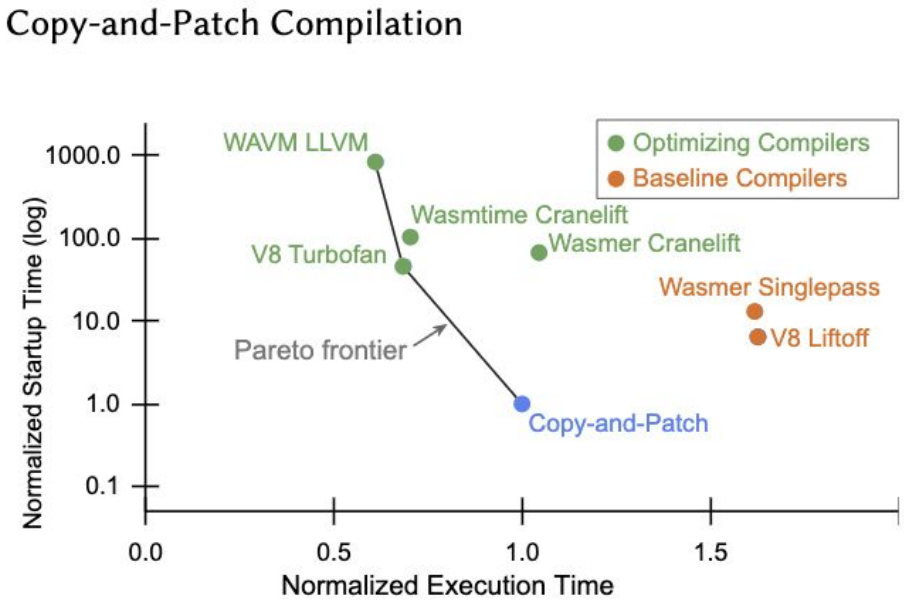
Figure 1: Side-by-side comparison of bytecode and threaded code for `x := 1 + 2`.

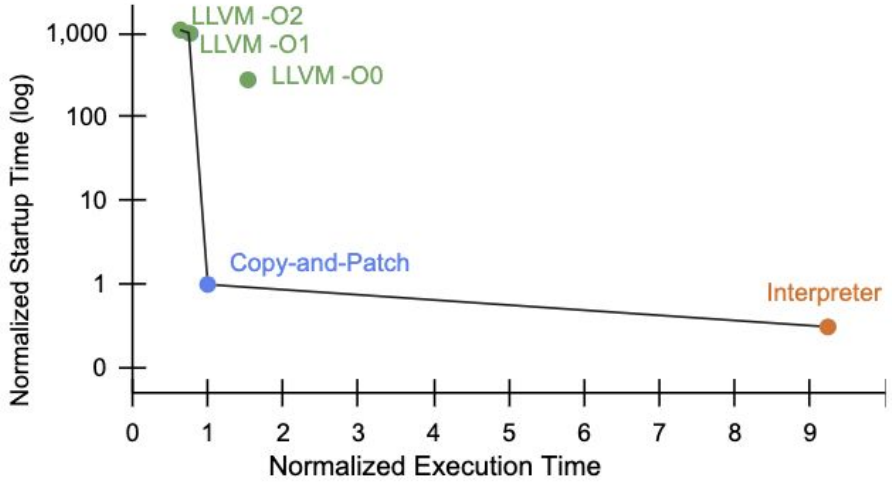
Copy-and-Patch Compilation

Copy-and-patch avoids building machine code instruction-by-instruction from a full compiler backend.

- Operations are represented by precompiled machine-code templates.
- Code generation copies the needed templates into an executable buffer.
- Template parameters and control-flow edges are patched after copying.
- The result is specialized executable code with lower compiler complexity.

Copy and Patch





[1] Haoran Xu and Fredrik Kjolstad. 2021. Copy-and-patch Compilation: A Fast Compilation Algorithm for High-Level Languages and Bytecode. Proceedings of the ACM on Programming Languages 5, OOPSLA, Article 136 (2021), 136:1–136:30 pages. doi:10.1145/3485513

Synergy with threaded and CnP

Threaded execution already gives CnP the right building blocks.

- Zag’s threaded functions are already native code.
- Each threaded function implements one VM operation with the same calling convention.
- CnP treats these functions as templates instead of generating new code.
- The method’s threaded-code sequence determines which templates to copy.
- Patching changes only control flow between functions, not their semantics.

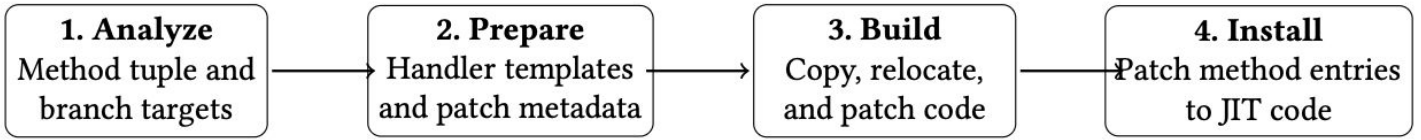


Figure 1. High-level copy-and-patch pipeline for a compiled method in Zag.

Original threaded functions

```
0x5dc4 pushLiteral
...
0x5e08 br x5       ; indirect transfer

0x75c0 inline-I
...
0x7610 br x5       ; indirect transfer

0x8d50 branchFalse
...
0x8d90 br x5       ; indirect transfer

0xa210 returnTop
```

JIT buffer

```
0x0000 pushLiteral
...
0x0050 b 0x0064    ; patched direct branch

0x0064 inline-I
...
0x00b8 b 0x00d0    ; patched direct branch

0x00d0 branchFalse
...
0x0110 b 0x0140 / b 0x0120

0x0140 returnTop
```

Figure 2. In threaded execution, continuation moves indirectly among threaded functions at scattered addresses. In the copy-and-patch tier, the same method fragment is laid out in one contiguous JIT buffer, and continuation becomes a sequence of patched direct branches.

Experimental Results

- Fibonacci (n=32..42): Integer CnP is consistently faster than threaded Integer Threaded.
- Relative gain is stable at about 5%.
- Absolute time saved grows with input size: 3 ms at fib(32) to 384 ms at fib(42).
- Microbenchmarks show larger gains when dispatch dominates.
- Best microbenchmark speedup: 1.54x on 20.

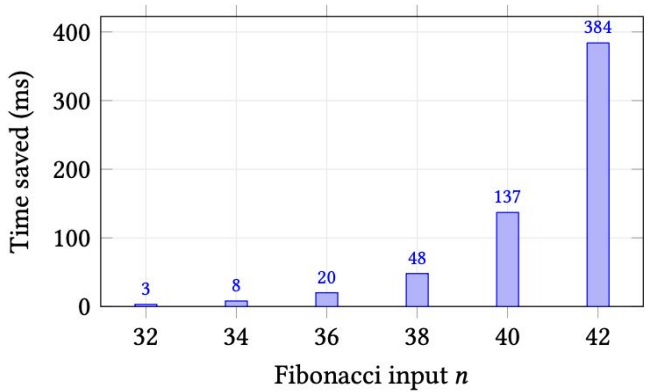
```
fibonacci
    self <= 1 ifTrue: [ ^ self ].
    ^ (self - 1) fibonacci + (self - 2) fibonacci
```

Metric	Value
sendCalls	204,668,308
getMethod	204,668,308
icHit	204,668,308
lookups	0
icPatch	0
ctxInstall	102,334,154
ctxReuse	102,334,154
sendDispatch time	3939.75 ms
sendDispatch share	51.9%

Experimental Results - Fibonacci

Copy-and-patch gives consistent steady-state gains on a send-heavy workload.

n	IntegerBr (ms)	IntegerCnP (ms)	Δ (ms)	Gain (%)	Speedup
32	57	54	3	5.3	1.056×
34	151	143	8	5.3	1.056×
36	394	374	20	5.1	1.053×
38	1034	986	48	4.6	1.049×
40	2703	2566	137	5.1	1.053×
42	7090	6706	384	5.4	1.057×

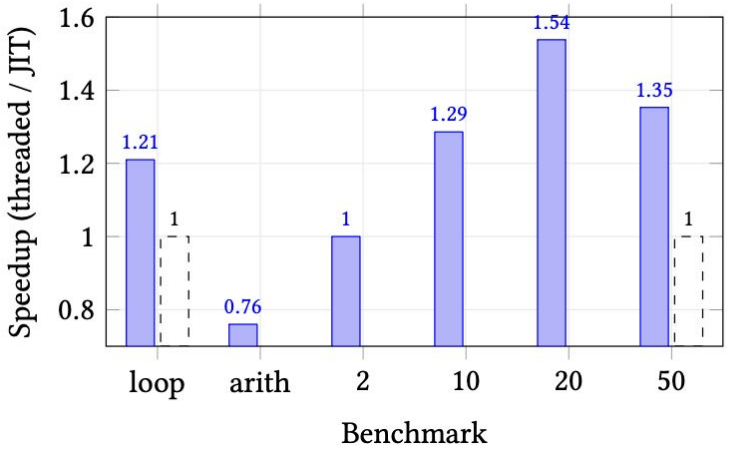


(a) Fibonacci absolute time saved.

Experimental Results - Microbenchmarks

CnP helps most when execution is dominated by threaded dispatch.

Case	Threaded (ns)	JIT (ns)	Δ (ns)	Gain (%)	Speedup
loop	342882	283327	59555	17.4	1.210×
arith	19	25	-6	-31.6	0.760×
2	2	2	0	0.0	1.000×
10	9	7	2	22.2	1.286×
20	20	13	7	35.0	1.538×
50	46	34	12	26.1	1.353×



Future work

- Improved Template Analysis
- Evaluate on broader Smalltalk workloads beyond Fibonacci and microbenchmarks.
- Add support for more architectures beyond AArch64.

Questions?

dmason@torontomu.ca or n1kapadia@torontomu.ca

https://github.com/Zag-Research/Zag-Smalltalk