Appendix A: Reproducing the Benchmark Results

Please refer to the artifact on how to reproduce all benchmark results in the paper.

Appendix B: Placeholder Macro Implementation

Please find a slightly simplified implementation (for the purpose of exposition) of the placeholder macros, as introduced in Section 5.1, below.

```
#define DECLARE_PLACEHOLDER_SYMBOLS(ord)                    \
  extern "C" void __function_placeholder_symbol ## ord(); \
  extern char __constant_placeholder_symbol ## ord        \
    __attribute__ ((__aligned__(1)));
DECLARE_PLACEHOLDER_SYMBOLS(0)
DECLARE_PLACEHOLDER_SYMBOLS(1)
DECLARE_PLACEHOLDER_SYMBOLS(2)
/* ... more ... */

#define DEF_CONSTANT_IMPL(ord, ...)              \
  using _T ## ord = __VA_ARGS__;                 \
  union _U ## ord { uint64_t d; _T ## ord v; }; \
  const uint64_t _tmp_ ## ord = (uint64_t)      \
    &__constant_placeholder_symbol ## ord;      \
  const _T ## ord CONSTANT_ ## ord =            \
    ((const _U ## ord *)&_tmp_ ## ord)->v;

// usage example: DEF_CONSTANT_0(int);
#define DEF_CONSTANT_0(...) DEF_CONSTANT_IMPL(0, __VA_ARGS__)
#define DEF_CONSTANT_1(...) DEF_CONSTANT_IMPL(1, __VA_ARGS__)
#define DEF_CONSTANT_2(...) DEF_CONSTANT_IMPL(2, __VA_ARGS__)
/* ... more ... */

#define DEF_CONTINUATION_IMPL(ord, ...)                  \
  using _F ## ord = __VA_ARGS__;                         \
  const _F ## ord CONTINUATION_ ## ord = (_F ## ord)    \
      __function_placeholder_symbol ## ord;

// usage example: DEF_CONTINUATION_0(void(*)(uintptr_t));
#define DEF_CONTINUATION_0(...) DEF_CONTINUATION_IMPL(0, __VA_ARGS__)
#define DEF_CONTINUATION_1(...) DEF_CONTINUATION_IMPL(1, __VA_ARGS__)
#define DEF_CONTINUATION_2(...) DEF_CONTINUATION_IMPL(2, __VA_ARGS__)
/* ... more ... */
```

The full implementation can be found here:
`https://github.com/sillycross/PochiVM/blob/591c5067a31474cb3c3fec260b7f83212fcbdf15/fastinterp/dynamic_specialization_utils.hpp`.

Appendix C: Raw Data for Microbenchmarks and TPC-H

Raw Data for Figure 23(a)

Figure 23(a) plots the Fibonacci sequence microbenchmark.
The input is $n = 40$, the output is 102334155, the 40th term of Fibonacci sequence.
The following table is used to generate Figure 10(a). The unit is in seconds.

	Copy+Patch	LLVM -O0	LLVM -O1	LLVM -O2	LLVM -O3	Peloton Interp	AST Interp
Codegen Time	0.0000012	0.0001578	0.0006634	0.0016148	0.0016167	0.0000157	0.0000014
Execution Time	0.3374227	0.3859762	0.2718486	0.1493952	0.1495836	39.4030755	17.3390853

Since this is a synthetic microbenchmark, the absolute execution time does not really matter (one can make it arbitrarily high or low by changing the input). The point is the comparison of codegen time and execution time between implementations.

The C++ code (using our metaprogramming system) constructing the AST can be found below:

```
using FnPrototype = uint64_t(*)(int) noexcept;
auto [fn, n] = NewFunction<FnPrototype>("fib");
fn.SetBody(
    If(n <= 2).Then(
        Return(Literal<uint64_t>(1))
    ).Else(
        Return(Call<FnPrototype>("fib", n - 1)
               + Call<FnPrototype>("fib", n - 2))
    )
);
```

The AST is used as input for all algorithms except Peloton, for which we used their API to emit logic.

Raw Data for Figure 23(b)

Figure 23(b) plots the Euler's sieve microbenchmark. It computes the number of primes in $[1, n]$ by Euler's sieve algorithm.

The input is $n = 10^8$, the output is 5761455, the number of primes between $[1, 10^8]$.

The following table is used to generate Figure 23(b). The unit is in seconds.

	Copy+Patch	LLVM -O0	LLVM -O1	LLVM -O2	LLVM -O3	Peloton Interp	AST Interp
Codegen Time	0.0000020	0.0002265	0.0028846	0.0033439	0.0033391	0.0000339	0.0000010
Execution Time	0.7586930	0.9057672	0.5732302	0.3844239	0.3842725	32.1482666	21.3643674

Since this is a synthetic microbenchmark, the absolute execution time does not really matter (one can make it arbitrarily high or low by changing the input). The point is the comparison of codegen time and execution time between implementations.

The C++ code (using our metaprogramming system) constructing the AST can be found below:

```
using FnPrototype = int(*)(int, int*, int*) noexcept;
auto [fn, n, lp, pr] = NewFunction<FnPrototype>(fnName);
auto cnt = fn.NewVariable<int>();
auto i = fn.NewVariable<int>();
auto j = fn.NewVariable<int>();
fn.SetBody(
    Declare(cnt, 0),
    For(Declare(i, 2), i <= n, Increment(i)).Do(
        If(lp[i] == 0).Then(
            Assign(lp[i], i),
            Assign(pr[cnt], i),
            Increment(cnt)
        ),
        For(Declare(j, 0), j < cnt && pr[j] <= lp[i] && i * pr[j] <= n, Increment(j)).Do(
            Assign(lp[i * pr[j]], pr[j])
```

```
        )
    ),
    Return(cnt)
);
```

Raw Data for Figure 23(c)

Figure 23(c) plots the Quicksort microbenchmark.
The input is a fixed random permutation of length $n = 5 \times 10^6$.
The following table is used to generate Figure 23(c). The unit is in seconds.

	Copy+Patch	LLVM -O0	LLVM -O1	LLVM -O2	LLVM -O3	Peloton Interp	AST Interp
Codegen Time	0.0000026	0.0002805	0.0021038	0.0028798	0.0028879	0.0000295	0.0000011
Execution Time	0.7666924	0.8319309	0.4119245	0.4118269	0.4115762	16.1601391	11.2265396

Since this is a synthetic microbenchmark, the absolute execution time does not really matter (one can make it arbitrarily high or low by changing the input). The point is the comparison of codegen time and execution time between implementations.

The C++ code (using our metaprogramming system) constructing the AST can be found below:

```
using FnPrototype = void(*)(int*, int, int) noexcept;
auto [fn, a, lo, hi] = NewFunction<FnPrototype>("quicksort");
auto tmp = fn.NewVariable<int>();
auto pivot = fn.NewVariable<int>();
auto i = fn.NewVariable<int>();
auto j = fn.NewVariable<int>();
fn.SetBody(
    If(lo >= hi).Then(Return()),
    Declare(pivot, a[hi]),
    Declare(i, lo),
    For(Declare(j, lo), j <= hi, Increment(j)).Do(
        If(a[j] < pivot).Then(
            Declare(tmp, a[i]),
            Assign(a[i], a[j]),
            Assign(a[j], tmp),
            Increment(i)
        )
    ),
    Assign(a[hi], a[i]),
    Assign(a[i], pivot),
    Call<FnPrototype>("quicksort", a, lo, i - 1),
    Call<FnPrototype>("quicksort", a, i + 1, hi)
);
```

Raw data for Figure 24(left)

The following table is used to generate Figure 24(left), the startup time delay comparison of Copy-and-Patch and LLVM on TPC-H queries. The unit is in seconds.

	Copy+Patch	LLVM -O0	LLVM -O1	LLVM -O2	LLVM -O3
Q1	0.0000527	0.0069334	0.0578107	0.0655658	0.0660590
Q3	0.0000890	0.0106740	0.1017739	0.1132111	0.1136530
Q5	0.0001775	0.0179830	0.2229477	0.2538737	0.2547383
Q6	0.0000111	0.0030583	0.0110413	0.0119456	0.0120161
Q10	0.0001391	0.0138133	0.1486148	0.1677385	0.1966974
Q12	0.0000737	0.0086935	0.0856067	0.0949418	0.0950891
Q14	0.0000385	0.0054426	0.0398965	0.0433711	0.0434078
Q19	0.0000606	0.0081130	0.0850491	0.0854244	0.0854345

Raw data for Figure 24(right)

The following table is used to generate Figure 24(right), the execution performance comparison of Copy-and-Patch and LLVM on TPC-H queries. The unit is in seconds.

	Copy+Patch	LLVM -O0	LLVM -O1	LLVM -O2	LLVM -O3
Q1	0.0848048	0.0950188	0.0524248	0.0516497	0.0531984
Q3	0.0633628	0.0654842	0.0540211	0.0520513	0.0514186
Q5	0.0960826	0.1030986	0.0844909	0.0949717	0.0949333
Q6	0.0215792	0.0332219	0.0167107	0.0139852	0.0139302
Q10	0.0391011	0.0423414	0.0332492	0.0316442	0.0317773
Q12	0.1982652	0.2212142	0.1726455	0.1584227	0.1594629
Q14	0.0294566	0.0330065	0.0193599	0.0189456	0.0191718
Q19	0.0939544	0.0979586	0.0724331	0.0672137	0.0680636

Raw data for Figure 25(left)

The following table is used to generate Figure 25(left), the start delay comparison of Copy-and-Patch, AST interpreter, and the cost to construct the AST from query plan on TPC-H queries. The unit is in seconds.

	Copy+Patch	AST Interp	Build AST from Query Plan
Q1	0.0000527	0.0000256	0.0000936
Q3	0.0000890	0.0000443	0.0001637
Q5	0.0001775	0.0000868	0.0003256
Q6	0.0000111	0.0000034	0.0000119
Q10	0.0001391	0.0000678	0.0002553
Q12	0.0000737	0.0000343	0.0001289
Q14	0.0000385	0.0000182	0.0000645
Q19	0.0000606	0.0000378	0.0001343

Raw data for Figure 25(right)

The following table is used to generate Figure 25(right), the execution performance comparison of Copy-and-Patch and AST interpreter on TPC-H queries. The unit is in seconds.

	Copy+Patch	AST Interp
Q1	0.0848048	2.2748882
Q3	0.0633628	0.7334789
Q5	0.0960826	1.1655509
Q6	0.0215792	0.1993574
Q10	0.0391011	0.3869788
Q12	0.1982652	1.1393818
Q14	0.0294566	0.2554241
Q19	0.0939544	0.7180035

Raw data for Figure 26

The following table is used to generate Figure 26, the scalability comparison. The number is the time it took to compile a function with that many statements of `a+=b`. The unit is in seconds.

# of statements	Copy+Patch	LLVM -O0	LLVM -O1	LLVM -O2	LLVM -O3
10K	0.0006466	0.0781449	0.2161661	0.2237115	0.2327550
50K	0.0039844	0.4367103	2.2650114	2.3161486	2.3230557
100K	0.0081096	1.0850422	7.4833087	7.5712563	7.6028616
200K	0.0156865	3.3642559	26.6158640	26.7890489	26.8480993
300K	0.0235379	8.9341055	57.7023578	58.1016596	58.6130917
400K	0.0315619	18.1412538	104.3427366	105.0832562	105.3841183
600K	0.0473712	45.1463359	257.7254170	256.5314960	257.9314789
800K	0.0632495	84.2222097	490.5130872	489.8637896	490.2477051

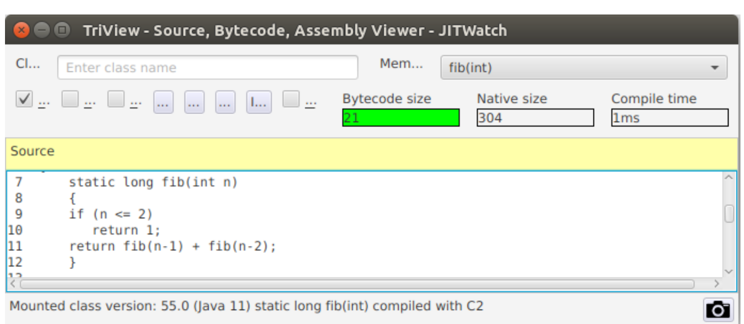
Raw data for Figure 27

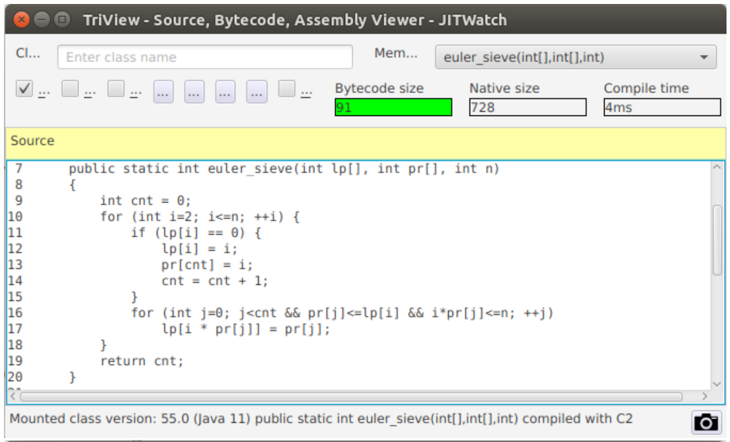
The following table is used to generate Figure 27, the performance breakdown. The unit is in seconds.

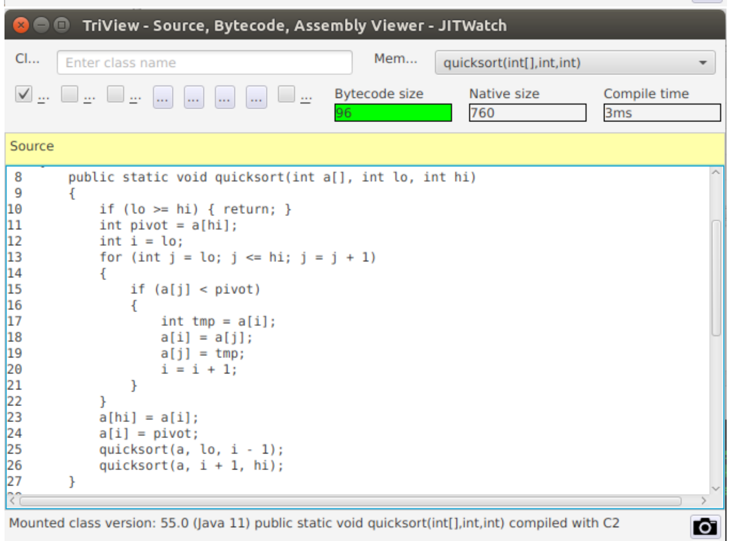
	Fibonacci	Euler's Sieve	QuickSort
AST Interpreter	17.3390853	21.3643674	11.2265396
Copy-and-Patch Basic	1.0073408	3.8418244	2.0545271
+ jump removal	0.5242140	1.2800686	1.2001584
+ pass temporary in register	0.3601368	1.0762484	0.8949352
+ super nodes	0.3511367	0.7961412	0.8352060
+ low level optimization	0.3374227	0.7586930	0.7666924

Raw data for Java performance (mentioned at the end of Section 7.2)

Please find the OpenJDK JITWatch report on Java JIT compilation time below. The three screenshots are Fibonacci sequence microbenchmark, Euler's Sieve microbenchmark, and QuickSort micorbenchmark, respectively. The time it took HotSpot JIT to compile the function is shown on top right corner.







Based on the screenshots above, the table below summarizes the compilation performance of copy-and-patch, LLVM -O3 and Java HotSpot JIT. The unit is in seconds.

	Copy-and-Patch	LLVM -O3	Java Hotspot JIT
Fib Sequence	0.0000012	0.0016167	0.001
Euler's Sieve	0.0000020	0.0033391	0.004
QuickSort	0.0000026	0.0028879	0.003

Java bytecode interpreter's execution performance is measured using Java option `-Djava.compiler=NONE` to force disable JIT. The table below summarizes the execution performance of copy-and-patch, Java interpreter, Java Hotspot JIT and LLVM -O3. The unit is in seconds.

	Copy-and-Patch	Java Bytecode Interpreter	Java Hotspot JIT	LLVM -O3
Fib Sequence	0.3374227	12.218	0.249	0.1495836
Euler's Sieve	0.7586930	5.398	0.504	0.3842725
QuickSort	0.7666924	3.345	0.498	0.4115762