Deoptimisation support in MysoreScript

Joshua Send
Trinity Hall

***Abstract*—Mysorescript is a simple scripting language which emulates basic Javascript. It is primarily interpreted, but has support for method-granularity just-in-time compilation using LLVM. While this offers large speed gains in many cases, code compiled at runtime is immutable. Addressing this fundamental point requires implementing deoptimisation, the process of modifying or replacing the compiled code to better utilize new runtime information. This report discusses deoptimisation for MysoreScript and evaluates it within the context of call-site inline caching.**

I. Introduction

JUST in time (JIT) compilation is an old technique which combines interpretation, and static ahead-of-time compilation. By initiating computation in the interpreter, the system can achieve fast startup times; selectively compiling oft-used code paths can then be used to improve execution speeds, thus combining some of the best parts of both approaches. The idea dates back to the 1960s, with LISP and later Smalltalk [1] [2].

One downside to running statically compiled software is that it cannot adapt and optimize in relation to the current system load, while a runtime which includes a JIT and a deoptimiser can dynamically recompile according to runtime information. Thus, JIT compilation may at some point be able to run more quickly than statically compiled code. The MysoreScript language includes only a JIT, so it is bound to provide bad compiled code in some situations. This report discusses the inclusion of deoptimisation in Mysorescript.

A. MysoreScript Summary

MysoreScript utilizes a parser which produces an actual abstract syntax tree (AST), with nodes in the tree represented by classes. The runtime initiates interpretation starting at the root block of statements, recursively evaluating each child of the root. In effect, the interpreter is performing a depth-first execution. The use of classes to define AST nodes, each with their own `interpret` method induces a call stack, which will have to be rebuilt when resuming execution in the interpreter (Section II-B3).

Another relevant feature of the language is that all closures, methods and functions are represented as the same object but with different calls for interpretation and compilation. This type of object is the unit of AST compilation, since the language only supports method-granularity compilation. Bound variables for closures are handled by copy, rather than reference: on creation of a closure the value of outer bound variables are copied into inner symbol tables.

Compilation is always initiated from the interpreter, which calls either a `compileMethod` or `compiledClosure` and saves the resulting compiled function. Because sub-calls within a function are not necessarily also compiled, a series of trampolines provide support for calling into the interpreter from the compiled functions.

B. Report Outline

This report discusses the inclusion of deoptimisation to Mysorescript. The new functionality is evaluated with an implementation of inline caching, which includes type assumptions and method locations hardcoded into the fast path. The following sections discuss deoptimisation, inline caching, and an evaluation in turn.

II. Deoptimisation

A. Overview

The key step to implementing deoptimization is the being able to exit a compiled function and resume execution in the interpreter, if a slow path is hit (that is, some kind of compile time assumption is invalidated). To be achieve this, stackmaps [1] can be used, which use a special data section of the compiled executable to record the value of variables. This section can be used to re-assemble an interpreter context with local, bound, and argument variables.

Once the interpreter context has been reconstructed, an interpreter evaluation stack has to be re-created to perform the correct tree execution of the AST nodes starting from the node that execution was suspended at. A simplifying assumption, that execution only stops on statement boundaries, was made here. In practical terms, this requires that MysoreScript code be written in SSA-like forms.

Once the execution has been resumed and completed in the interpreter, the result of the function is passed back to the compiled code to return immediately.

B. Resuming execution in the Interpreter

1) Stackmaps and Patchpoints: Stackmaps are a program section emitted by the compiler which can be used to record a set of values or locations given at compile time. There are two LLVM intrinsics which initiate the creation of a Stackmap entry: `llvm.experimental.stackmap` and `llvm.experimental.patchpoint`, with the latter coming in `void` and `i64` (64 bit integer) return type variations.

Listing 1. Stackmap Section Layout

```
Header {
  uint8  : Stack Map Version (current is 3)
  uint8  : Reserved (expected to be 0)
  uint16 : Reserved (expected to be 0)
}
```

[1] https://llvm.org/docs/StackMaps.html

```
uint32 : NumFunctions
uint32 : NumConstants
uint32 : NumRecords
StkSizeRecord[NumFunctions] {
  uint64 : Function Address
  uint64 : Stack Size
  uint64 : Record Count
}
Constants[NumConstants] {
  uint64 : LargeConstant
}
StkMapRecord[NumRecords] {
  uint64 : PatchPoint ID
  uint32 : Instruction Offset
  uint16 : Reserved (record flags)
  uint16 : NumLocations
  Location[NumLocations] {
    uint8  : Register | Direct | ...
    uint8  : Reserved (expected to be 0)
    uint16 : Location Size
    uint16 : Dwarf RegNum
    uint16 : Reserved (expected to be 0)
    int32  : Offset or SmallConstant
  }
  uint32 : Padding (only if not 8byte aligned)
  uint16 : Padding
  uint16 : NumLiveOuts
  LiveOuts[NumLiveOuts]
    uint16 : Dwarf RegNum
    uint8  : Reserved
    uint8  : Size in Bytes
  }
  uint32 : Padding (only if not 8byte aligned)
}
```

The `stackmap` intrinsic only requires an ID and a list of values to record, which are saved into a stackmap section. The `patchpoint` intrinsics take an ID, bytes to reserve for code patching (not used), a function to call, arguments to the function, and any values to save to the stackmap.

In order to call back into the interpreter without causing disruption to the register allocation for the fast path in the compiled code, the `AnyReg` calling convention (Section II-B3) should be used – this is only available with the `patchpoint` intrinsic. Its return value will additionally be used to get the result of the function from the interpreter.

Listing 1 shows the memory layout of the stackmap section. Each function has an associated number of records, one per `patchpoint` or `stackmap` intrinsic. Due to the method-granularity compilation, a separate Stackmap section is emitted per method, and number of functions is always one.

While it should be possible to have more than one `StkMapRecord` per function, I failed to implement this as the value of the `Function Address` was always 0. Normally, the way to find an the desired stackmap record is to calculate the $FunctionAddress + InstructionOffset$ which should be equal to the return address of the intrinsic function call. Thus for evaluation I was forced to restrict myself to examples with a single call site per function.

The `patchpoint.i64` intrinsic stores arguments in a set of `Location`'s, the first of which indicates the return register when using the AnyReg calling convention.

To interface with a stackmap section easily, a `StackMapParser` was created, which takes the address of the stackmap and returns saved values, transparently handling the raw memory addressing.

It is recommendable to save only live variables to the stack map. However, I ran out of time to implement this. Stackmap records already contain a set of live registers at the call site, but this would be difficult to match back to variable names. A live variable analysis could be written over the abstract syntax tree instead.

2) Intercepting JIT Program Sections: To be able to initialize a `StackMapParser` the address of the stackmap section is required. This was obtained by registering an event listener with the JIT execution engine. The callback receives a notification on each section created, but only the one named `.llvm_stackmaps` is saved.

Since multiple functions my be compiled and produce independent stackmap sections, the runtime maintains a mapping from compiled functions to the correct stackmap sections – a side exit from a compiled method is not able to convey any immediate information to the callee when using the AnyReg calling convention.

The first solution to this involved extending the built in, simple JIT memory manager to record section addresses, but this would not consistently operate correctly. A second, fully functional solution was built along the lines of Pyston's program section interceptor [2].

3) JIT to Interpreter and Back: The AnyReg calling convention was used to make the compiled fast path even more efficient. However, in contrast to the C System V x86_64 calling convention, which assigns certain registers to be used as arguments, some as return holders, and some registers as caller- and callee-preserved, AnyReg preserves registers from before the call and places the burden of finding arguments to the function on the callee via the stackmap, which stores records in order of arguments.

To deal with this, an x86-64 assembly trampoline was written, which pushes all registers onto the stack, and does a C style call into the interpreter function `reconstructInterpreterContext`, passing the location of the registers pushed onto the stack. The interpreter then retrieves the current stackmap from the runtime, which keeps track of the running JIT'ed function via a simple pointer, and uses the stackmap record locations and the stack of register values to retrieve variables saved at the patchpoint location. These values are used to reconstruct a valid interpreter context. The `resumeInInterpreter` function is then called.

Because the JIT'ed function is called with standard C calling conventions, it may have preserved caller registers that need to be recovered from the stack, while also retuning a value. To achieve this, the x86 trampoline places the return value from the interpreter resumption, into the register indicated by the first stackmap location. The JIT slow path then immediately returns this value.

If there were no spilled variables, a faster method would be to have the x86 trampoline pop off the JIT stack frame, except the return address and jump or call into the `resumeInInterpreter` function, which would return straight to the caller of the JIT function. This would provide

[2]Thanks to the now defunct Pyston! https://github.com/dropbox/pyston

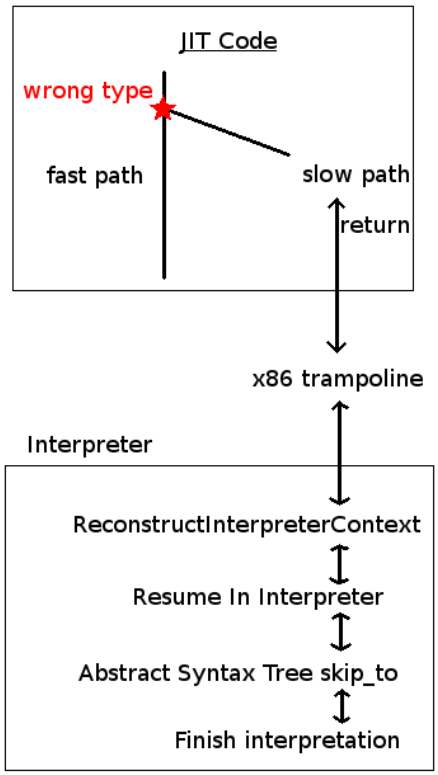


Fig. 1. Program control flow when resuming in interpreter.

speedups by not requiring passing return values back through various stack frames.

4) Skipping to correct AST Node: `resumeInInterpreter` utilizes the pointer to the currently executing JIT'ed function, an AST node, as the root of a DFS tree search searching for the AST node execution was paused at. This was implemented by mirroring the `interpret` methods on AST nodes, with some significant modifications.

The main task is retrieving the function value computed by the time a return statement is encountered. This may mean switching from skipping, and not executing commands while locating the desired AST node, to full interpretation. This is well exemplified in Listing 2.

Listing 2. The skip_to method on a block of Statements

```
void Statements::skip_to(Interpreter::Context &c,
        Statement* ast_node) {
  for (auto &s : statements) {
    if (c.isReturning) {
      return;
    }
    if (c.astNodeFound) {
      s->interpret(c);
    } else{
      s->skip_to(c,ast_node);
    }
  }
}
```

Return values and the flag for when to switch from skipping nodes to full interpretation are both attached to the interpreter context.

C. Recompilation

1) Profiling: Another key element of the deoptimization process is the actual replacement of compiled functions. This requires some kind of profiling of the data. In this case, the focus is on inline caching.

A simple replacement strategy was implemented using a `current_type_assumption` pointer to a class attached to every call site (actually, every AST node inherits the same ability to set assumptions, and usable at compile time to optimize the fast path), a pointer to an alternative type, `alternative_type`, and a counter `alternative_type_count`.

The simple algorithm used here is outlined in Listing 3. This strategy requires at least $threshold$ contiguous alternative types to initiate a recompile. The effect of this value is explored in Section IV-E

Listing 3. Recompilation Strategy Pseudocode

```
...
type = object->isa; // get the type
if type == current_type_assumption:
  alternative_type_count = 0
  ...execute fast path...
else:
  if type == alternative_type:
        alternative_type_count++;
        if alternative_type_count > threshold:
          set recompile flag on the current closure
          current_type_assumption = type
          alternative_type = nullptr
          alternative_type_count = 0
  else:
        alternative_type = type
        alternative_type_count = 0
  ... execute slow path...
```

More sophisticated schemes are possible that require more data to be collected lend themselves to heterogeneous, randomized mixtures of data types rather than the sequences of identical types optimized for here.

2) Recompilation and Replacement: Once the slow path has been taken often enough, on exit from the interpreter resumption back into the x86 trampoline, the `recompile` flag is checked. If it is set, the current function being executed is checked for the presence of a selector to determine if needs to be compiled as a method, or as a closure, and the corresponding methods are called on the root AST node.

The useful thing about embedding type assumptions in individual AST nodes mean that during their compilation, they can specialize their code based on those assumption, and by changing the assumptions in the interpreter and inducing recompilation, the changes are automatically incorporated. This in conjunction with callsite caching is explored in Section III.

I note here that the replacement may be causing memory leaks, as I do not free the memory used by previous recompilations. This would requires some effort, since the JIT function cannot be fully deleted until the x86 trampoline has returned to it, and it has returned again. However, with small functions such as the ones tested here, the memory leak is not an issue.

III. Inline Callsite Caching

A. Concept

Callsite inline caching saves a function pointer at each call site, which may be used if the desired method matches the type saved with the pointer. The policy for replacement of the pointer is also flexible. I use immediate replacement for testing.

A simple, and effective implementation is to store a pointer to a type, and a pointer to a method at each call site in the AST. The interpreter can then use these to do caching, providing modest speedups if there are multiple methods attached to the target class (compare Figure 2 to Figure 5).

The interpreter code can be directly translated into machine code during the compilation process, using pointers to the cached type and method pointers. The compiled method must check for null and special integer object types, but the slow path only does one method lookup and a store operation during a cache miss – since the cache is updated immediately, this leads to a fast compiled code execution over relatively homogeneous data streams. The experiments carried out in the Evaluation section show this as well.

B. Using Deoptimisation

The callsite inline caching implementation is somewhat different with deoptimization, in that the fast path can be very short and hard code the expected type and location of the cached method, based on the `current_type_assumption` value and the method looked up previously in the slow path.

The compiler can even use the type information to generate specialized code based – for instance, the `SmallInteger` class would normally need to checked for by examining the lower 3 bits of a pointer. However, assuming it is an integer means this check can be avoided in the compiled code.

IV. Evaluation

This section will analyze the caching optimization implemented using various sample programs. Configurations explored are: adjusting the number of times a (potentially compiled) closure is called, the number of class methods present, and the number of different types a function is applied to, which in turn induces re-compilations if deoptimization is available. The programs using a mixture of types are implemented with long sequences of each type rather than a random stream to make better use of the recompilation strategy outlined in Section II-C.

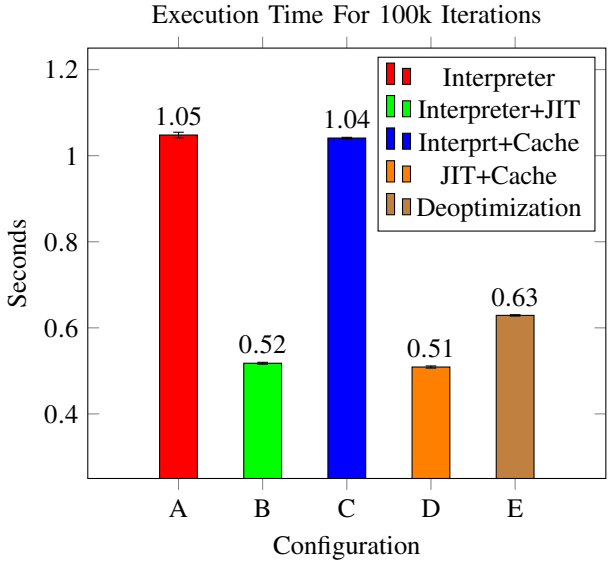
Comparisons to the original, unmodified MysoreScript code base, and to a inline callsite caching implementation following Section III without deoptimization are provided. The focus is on execution time. Memory usage is extremely stable and sits around 108KB for non-JIT'ing configurations, and 126KB to 135KB for 1-5 JIT'ed functions.

A. Baseline

The first configuration tested is a program with a simple two-class hierarchy, with one function and one instance integer-value variable. The subclass overrides the parent's implementation of the function. An external *getter* takes one of these two objects and calls the function, which simply returns the instance variable.

Figure 2 presents the result of evaluating the *getter* 50,000 times with an instance of the parent, and 50,000 times with the sub-class. This induces a recompilation after a the side exit is taken too many times.

Fig. 2. Running time for different MysoreScript configurations. The non-JIT'ing versions are in red and blue.



Things first thing to note here is that preventing MysoreScript from activating the JIT has severe performance penalty, of around 50-100%. Further, caching has very little impact over simple compilation without it – likely because only 1 function needs to be looked up using the $O(n)$ search strategy without it.

Lastly, note that deoptimization decreases performance, by around 25%. The difference in means between using the JIT+caching configuration and the deoptimizing configuration is 0.629216 – 0.5094416 = 0.1197744 seconds. A measure of the time taken to execute a compilation of a simple function like the ones used here gives approximately 0.1486 ± 0.005 seconds. Thus the gap is attributed to the extra recompilation that needs to be done halfway through execution. We expect this gap to shrink as the iterations increase.

B. Increasing Number of Iterations

Increasing the number of iterations in the same program from 100,000 to 1 million illustrates the decreasing gap well. Non-JIT'ing configurations are left out here.

Figure 3 shows that all three of basic JIT, inline-caching JIT, and deoptimizing inline-caching are competitive with each other. The inline-caching JIT is slightly ahead of the others.

Increasing the number of iterations (Figure 4) by another factor of 10 to 10 million leads to inconclusive results: all three configurations are within one standard deviation of each other. However, it is reassuring that the deoptimizing JIT is no slower

Fig. 3. Running time for different JIT'ing configurations – non compiling versions are left out here. 1 million iterations.

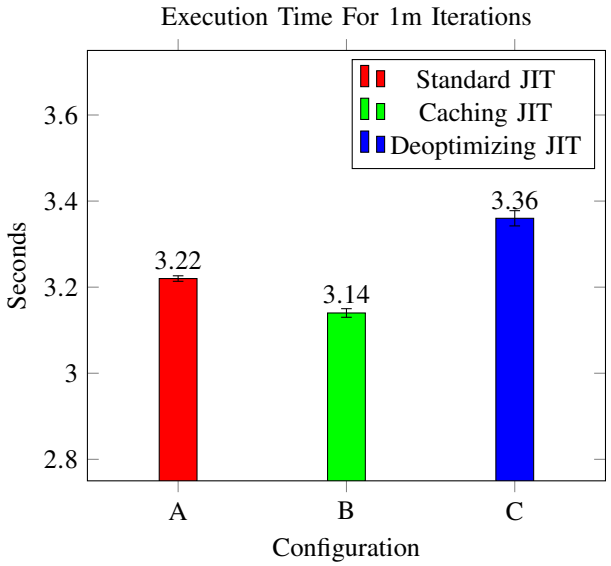
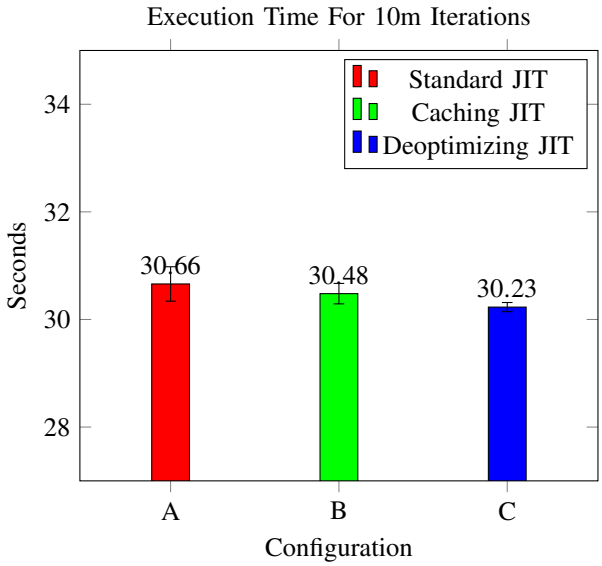


Fig. 4. Running time for different JIT'ing configurations – non compiling versions are left out here. 10 million iterations. Note the graph is plotting very similar values quite differently.



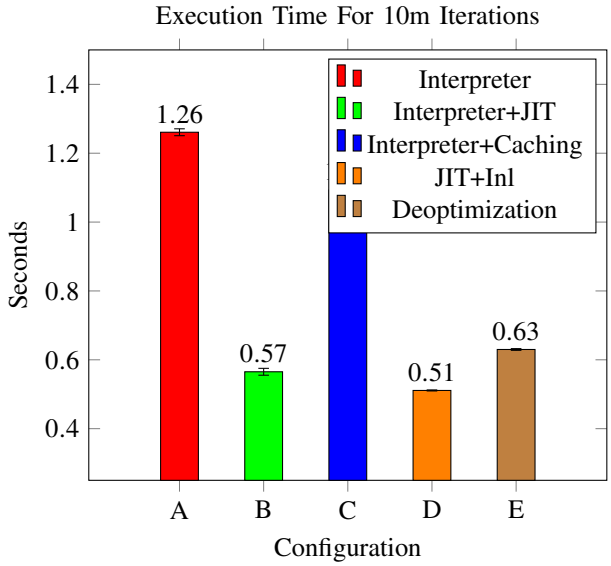
than the others, and perhaps a more complete implementation would lead to statistically significant speed gains.

C. Increasing Number of Class Methods

The previous experiments were all done with a class hierarchy in which members had a single method. This is not only unrealistic, but favors the non-caching, standard language implementation without any profiling or instrumentation, as seen above.

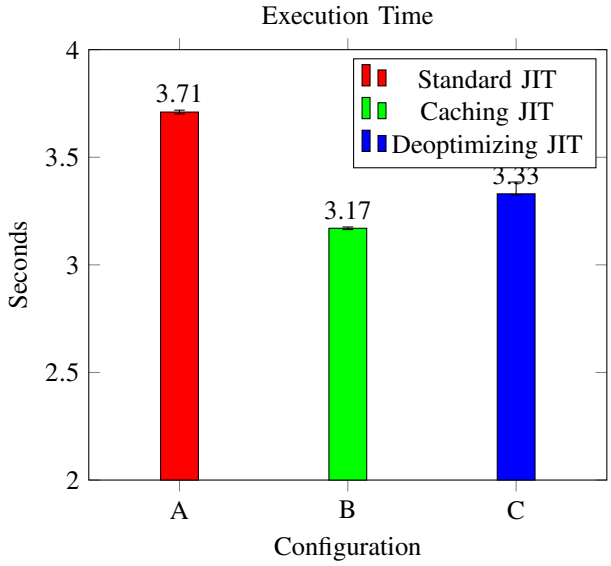
To look at an extreme first, the program is modified to include an extra 200 member functions in the parent class. Figure 5 shows the 100k iterations scenario again. The non-caching configurations are now noticably slower than the cahing ones, which have almost no performance penalty.

Fig. 5. Running time for different JIT'ing configurations.



Here too one can watch the performance gap between caching JIT and deoptimizing+caching JIT shrink as the number of iterations is increased. Unfortunately the results between cachingng JIT and deoptimizing JIT (Figure 7, for 10 million iterations) are not statistically significant.

Fig. 6. Running time for different JIT'ing configurations with 200 class methods and 1m iterations. Non-compiling configurations have been left out.



The same game can be played with 20, rather than 200 class methods: this is a much more realistic scenario, though languages like Java with large class hierarchies may exceed that number easily. Only the 10 million iteration data is plotted here to reduce clutter. See Figure 8.

D. More than 2 Deoptimizations

The scenarios tested so far are as ideal as they can be for the deoptimizing implementation. A slightly different example

Fig. 7. Running time for different JIT'ing configurations with 200 class methods and 10m iterations. Non-compiling configurations have been left out.

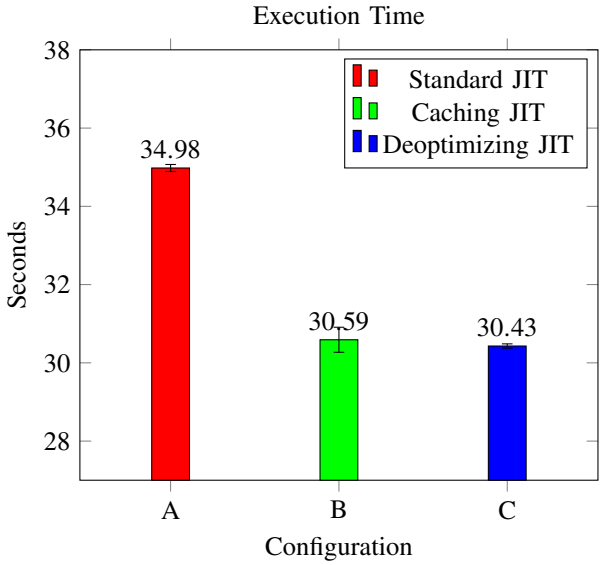
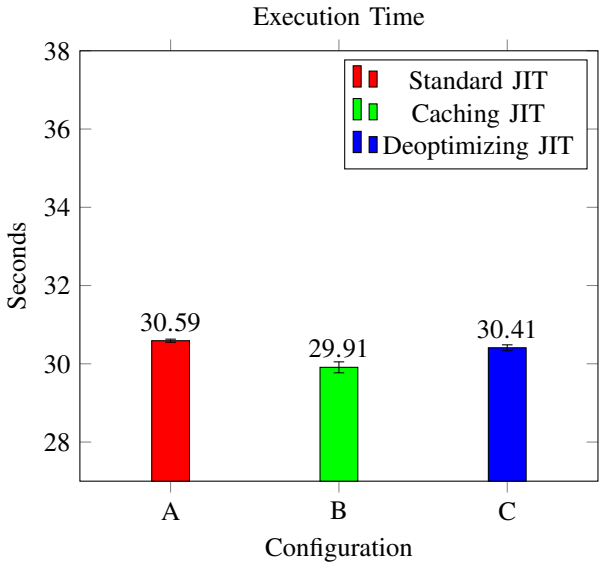


Fig. 8. Running time for different JIT'ing configurations with 20 class methods and 10m iterations. Non-compiling configurations have been left out.



is taken here, with classes that overload the + operator in different ways. 5 different classes with 20 methods each are run through a *runner* which executes the + operator – this runner is recompiled 5 different times accordingly. Each class type is run through the *runner* 200,000 times.

In this case, the standard language implementation comes out fastest, while the deoptimizing JIT is slowest by a small margin, as expected.

E. When To Deoptimize

Using the simple replacement strategy described in Section II-C there is one parameter to be tuned: the number of consecutive different types that need to be encountered before the JIT'ed segment is recompiled using new type assumptions.

Fig. 9. Running time for different JIT'ing when executing 200k a function for iterations of each of 5 different types.

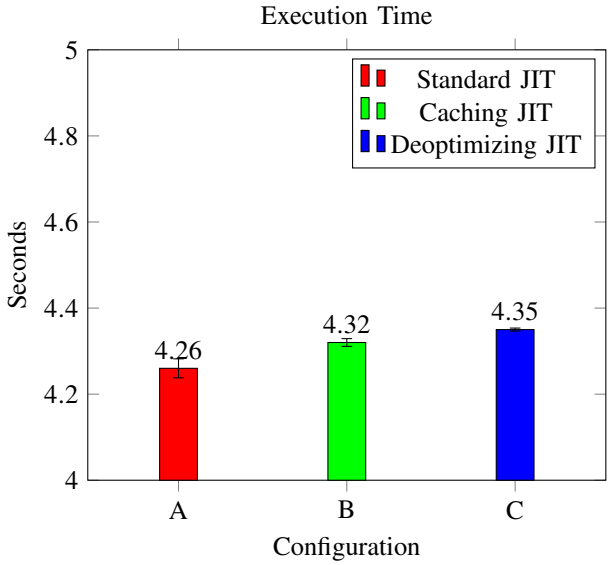
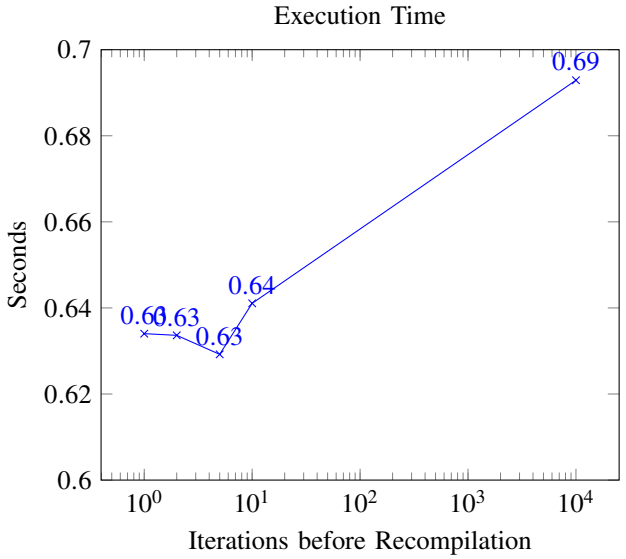


Fig. 10. Running time for 200k iterations, 1 class methods with a recompile threshold of 1, 2, 5, 10, and 10000. Plotted on semilog to include the datapoint at 10,000 iterations



Interestingly, there is very little variation performance as the threshold is varied. The best setting seems to be between 5 and 10, as empirically suggested by Figure 9. However, unless the threshold is set extremely high there is hardly a any performance difference. At 10,000 side exits the there is a difference of 0.05s, an approximately 8% performance penalty for spending 10% of computation traversing the slow path.

This plot reveals that the side exit, interpreter reconstruction, resumption, and return back to the JIT is extremely fast. Given that the fast path of the inline-caching function is extremely short, the big cost to be paid using deoptimization is the time to execute a recompilation via LLVM.

V. Conclusion

This report discusses the inclusion of deoptimization into the MysoreScript language. The evaluation focuses on comparing inline caching implementations along with the baseline, non-caching interpreter and JIT.

One strong conclusion is that inline caching is worth it if there are more than a few class methods. However, the hardware is able to execute the $O(n)$ method lookup very efficiently, making the optimizations perform worse with few methods in a class.

Another is that the time required to recompile a method or closure is relatively high (around 0.15 seconds), meaning it's not worthwhile if frequent recompilations are required. Even just 5 recompilations over a a 4 second runtime place the deoptimizing JIT configuration slower than the alternatives.

The implementation of deoptimization is a little bit incomplete, in that is can be extended to include live variable analysis, reducing the runtime cost of taking the slow path. However, the data also supports that side exit is already very fast – it is possible another compile time analysis would be detrimental to performance until the cost is amortized over many slow path exits.

A. Future Work

The most urgent addition to the code base is the ability to have more than one stackmap record per function, which is a rather crippling restriction. Being able to have a fast path with many inline cache hits would make the deoptimizing variant presented much faster in the common case.

A more difficult improvement would be to lift the restriction of only allowing side exits at statement boundaries: this would require a method for returning intermediate results from the compiled execution into non-existent values in the interpreter.

References

[1] John Aycock. "A brief history of just-in-time". In: *ACM Computing Surveys (CSUR)* 35.2 (2003), pp. 97–113.

[2] John McCarthy. "Recursive functions of symbolic expressions and their computation by machine, Part I". In: *Communications of the ACM* 3.4 (1960), pp. 184–195.