Optimizing MATLAB `feval` with Dynamic Techniques

Nurudeen Lameed   Laurie Hendren

Sable Research Group,
School of Computer Science, McGill University,
Montréal, Québec, Canada
{nlamee,hendren}@cs.mcgill.ca, `http://www.sable.mcgill.ca/mclab`

Abstract

MATLAB is a popular dynamic array-based language used by engineers, scientists and students worldwide. The built-in function `feval` is an important MATLAB feature for certain classes of numerical programs and solvers which benefit from having functions as parameters. Programmers may pass a function name or function handle to the solver and then the solver uses `feval` to indirectly call the function. In this paper, we show that there are significant performance overheads for function calls via `feval`, in both MATLAB interpreters and JITs. The paper then proposes, implements and compares two on-the-fly mechanisms for specialization of `feval` calls. The first approach uses on-stack replacement technology, as supported by McVM/McOSR. The second approach specializes calls of functions with `feval` using a combination of runtime input argument types and values. Experimental results on seven numerical solvers show that the techniques provide good performance improvements.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers

Keywords Dynamic Optimization; JIT Compilation; *feval*; LLVM; MATLAB; McJIT

1. Introduction

MATLAB is a dynamic array-based language used by scientists, engineers and students in many disciplines. MATLAB's high-level matrix operators and dynamic typing makes the language suitable for a wide variety of numerical computations. An additional important feature of MATLAB is its support of higher-order functions through the `feval` construct which is widely used in many classes of numerical computations, including fitting functions, estimating Ordinary Differential Equations, machine learning algorithms such as simulated annealing, and general plotting functions. All of these applications share a similar pattern, the main computation function has a function parameter that can accept either a function handle, or a function name as the actual argument. The body of the computation function then repeatedly evaluates the function passed in using `feval`.

DLS '13, October 28, 2013, Indianapolis, Indiana, USA.
Copyright © 2013 ACM 978-1-4503-2433-5/13/10. . . $15.00.
http://dx.doi.org/10.1145/2508168.2508174

Historically, MATLAB has been mainly an interpreted language, with an emphasis on efficient libraries, but no particular focus on efficient execution. More recently, there have been several efforts to provide more efficient execution engines such as Mathworks' proprietary MATLAB JIT Accelerator, first introduced in MATLAB 6.5 [22], and research efforts such as MaJIC [2] and the McLAB group's open source VM/JIT, McVM [5, 16].

Efficient JIT-based execution means that MATLAB can be used for more than just prototyping or calling library routines - it can now be used for a wide variety of user-defined applications. However, there remain many challenges to enable efficient execution, and this paper focuses on the `feval` challenge.

To determine potential overheads of `feval`, we identified a set of seven benchmarks that use algorithms that naturally use `feval`, and performed initial experiments on three interpreters (Octave, Mathworks MATLAB 7 in interpreter mode, and McVM in interpreter mode), plus two JITs (Mathworks MATLAB with the JIT enabled, and McVM with the JIT enabled).[1] These experiments showed, in both the interpreter and JIT situations, that there are significant overheads for calls via `feval`, as compared to direct function calls and inlined function calls.

To reduce the overheads of `feval`, we then designed and implemented two alternative dynamic mechanisms. The first mechanism is the more general of the two approaches. It can handle a wider variety of uses of `feval`, and is based on on-the-fly code generation and on-stack replacement (OSR) techniques implemented in McVM [14]. The OSR-based technique identifies potentially important `feval` calls, and then uses McVM's OSR technology to specialize the `feval` calls to specific direct calls, and to provide correct backup to the general case when the specialized calls do not match the calling context. The second mechanism extends the McVM JIT on-the-fly code specialization mechanism to specialize on the **value** of function parameters in those cases where the parameter is used inside the body of the function as the first argument to `feval`.

The main contributions of this paper are:

Measuring the cost of `feval`: We evaluated the overheads of `feval` and show significant overheads for calls via `feval` for important classes of benchmarks.

OSR-based specialization of `feval`: We developed a technique to detect and instrument important `feval` sites with OSR points, and we designed an OSR-based transformation which can be done at the LLVM IR-level, without requiring access to the generated assembly code. We also designed appropriate JIT-time tests to optimize the guards required to determine if

[1] Octave is an open source interpreter-only implementation which does not have a JIT.

the specialized call could be made or if the general backup path should be taken.

JIT value-based specialization: We designed an extension to the McVM JIT specialization mechanism. Previously specialization was performed based only on the dynamic **types** of function arguments. In the new approach, we also specialize on the **value** of a function argument, for the case where that argument is used as the first argument to a call to `feval` inside the body of the function to be compiled.

Implementation in McVM: We implemented both proposed approaches in McVM. Our implementation is open source.

Experimental Results: We evaluated both approaches, comparing them both to the original `feval` implementation, as well as to hand-specialized versions of the program.

The remainder of the paper is structured as follows. In Section 2, we present a complete MATLAB example, and we show our initial experiments that demonstrate the large overheads for `feval`. In Section 3, we give an overview of McVM and the OSR library used in our implementation. Section 4 gives the background to the standard `feval` implementation in McVM, and then shows how to improve it using OSR. Section 5 provides the details of our second approach based on specializing on the values of key function parameters. Section 6 reports on our experiments. For a set of seven benchmark programs, we first discuss the overheads of `feval`; then we assess the impact of our OSR-based function specialization under three different optimization settings; we conclude this section by comparing the OSR-based specialization with the JIT value-based specialization. We end the paper with a discussion of related work in Section 7 and conclusions in Section 8.

2. Motivation and Problem

In order to provide some intuition about MATLAB and the `feval` challenges, consider the example MATLAB function *newton* in Listing 1. As shown on line 1, the function takes four input arguments, with the first argument *fun* corresponding to either the name of a function or a function handle. Note that MATLAB has no declared types, although the programmer certainly has some expected types in mind, as indicated by the comments on lines 3 to 13. Indeed, not only does the programmer expect the first argument to be a string containing the name of a function, but she also expects the named function to take one input argument and produce two outputs. This is also clear from line 22, where `feval` is used to call the function provided by the argument *fun*. Listing 2 shows the definition of *fx3n*, which is one possible function that could be provided to *newton*.

The MATLAB function `feval` is a built-in function, that is used in MATLAB to indirectly evaluate a function at run time. `feval` is overloaded, with two versions available:

```
[y1, y2, ...] = feval(fhandle, x1, ..., xn)
[y1, y2, ...] = feval(fname, x1, ..., xn)
```

where *fhandle* is a first class type in MATLAB which can be bound to a MATLAB built-in function or a user-defined function using the '@' operator. If the second version is used, then *fname* must be a string containing a single function name and cannot contain a path to a function or a directory.[2]

For our example program in Listing 1, a typical call would be one of the following:

```
newton(@fx3n, 3, 5e−16, 5e−16)
newton('fx3n', 3, 5e−16, 5e−16)
```

[2] See `http://www.mathworks.com/help/matlab/ref/feval.html`.

```
1  function r = newton(fun,x0,xtol, ftol )
2
3  % newton      Newton's method to find a root of the scalar
4  %             equation f(x) = 0
5  % Synopsis:   r = newton(fun,x0, xtol, ftol )
6  % Input:  fun   = ( string ) name of mfile that
7  %                        returns f(x) and f'(x).
8  %         x0    = initial guess
9  %         xtol  = absolute tolerance on x.
10 %                 Smallest: xtol=5*eps
11 %         ftol  = absolute tolerance on f(x).
12 %                 Smallest: ftol=5*eps
13 % Output: r     = the root of the function
14
15   xeps = max(xtol,5*eps);
16   feps = max(ftol,5*eps); % Smallest tols are 5*eps
17   x = x0; k = 0;
18   maxit = 15; % Initial guess, current and max iterations
19   while k <= maxit
20     k = k + 1;
21     % Returns f( x(k−1) ) and f'(x(k−1) )
22     [f,dfdx] = feval(fun,x);
23     dx = f/dfdx;
24     x = x − dx;
25     if ( abs(f) < feps ),  r = x; return; end
26     if ( abs(dx) < xeps ),  r = x; return; end
27   end
28 end
```

Listing 1. Newton's method to find a root of the scalar equation f(x) = 0, adapted from [19, 20]. Function *fx3n* is shown in Listing 2.

```
function [f, dfdx] = fx3n(x)
  % fx3n Evaluate f(x) = x − x^(1/3) − 2 and
  % dfdx for Newton algorithm
  f = x − x.^(1/3) − 2;
  dfdx = 1 −(1/3)*x.^(−2/3);
end
```

Listing 2. Function *fx3n* from [19, 20].

where the first case passes a function handle and the second case passes a string containing the name of the function.

Clearly algorithms such as *newton* are naturally parameterized over the evaluation function, and MATLAB's `feval` provides a mechanism for this abstraction. However, one might wonder if the use of `feval` causes any significant slow down. To determine this, we studied the cost of `feval` implementations in three implementations of MATLAB: (1) Mathworks' implementation for the MATLAB programming language; (2) Octave, a GNU[3] open-source implementation of the MATLAB language; and (3) McVM, our open source MATLAB framework.

The Mathworks' MATLAB system (called MATLAB in the tables) provides an interpreter for the language and also an accelerator (a JIT compiler). Octave is an interpreter for the MATLAB language. It does not have a JIT compiler. Like Mathworks' MATLAB, McVM has an interpreter and an optimizing JIT compiler. The precise machine and software configurations used for our experiments is given in Figure 1(a). We conducted our experiments on these systems over a set of MATLAB programs from numerical computing domains. These benchmarks include programs for finding the roots of polynomials and to integrate first order ordinary differential equations. All but one (*sim_anl* [4]) of our benchmarks were collected from [20]. We give a short description, together

[3] `http://www.http://www.gnu.org/software/octave/`

[4] `http://www.mathworks.com/matlabcentral/fileexchange`

with a static count of the total number of `feval` calls in the program in Figure 1(a). The table also shows the number of `feval` calls in a loop in each benchmark.

In Figure 1(c) and Figure 1(d), for each benchmark, we show the execution times for the three systems: Octave, MATLAB and McVM. For all our experiments, the execution times do not include the start-up cost of the VM/interpreter. under the JITs, the execution time of a benchmark is the average of 10 runs of the benchmark. In addition, only the execution time of the first run includes the compilation time. By taking the average of the execution times of 10 runs, we spread the compilation cost over the 10 runs. For the interpreters, the execution time is the average of 5 runs.

Figure 1(c) gives the execution times measured in seconds when the benchmarks were interpreted under the three systems. Similarly, Figure 1(d) gives the execution times, also measured in seconds, when the benchmarks were run with MATLAB and McVM JITs enabled. As we mentioned earlier, Octave does not have a JIT compiler. In each table, the column labelled (F) gives the time for the original benchmark, with the `feval` call. The column labelled (D) gives the time when we the `feval` is replaced (by hand) with a direct call to the input function used to run the benchmark, and the (I) column gives the time when the function is inlined (by hand). The rightmost columns give the speedups of the (D) and (F) versions as compared to the original `feval` version.

These results are very interesting because they show that even for the interpreted cases there are substantial overheads for `feval`. When the `feval` is replaced by a direct call the speedups range from 1.05 – 1.23 for Octave, 1.00 – 1.15 for MATLAB, and 1.00 – 1.30 for McVM. When the direct call is inlined the speedups increase even more, ranging from 1.11 – 3.38.

The `feval` overhead for the JIT-based system are proportionally even higher. For the MATLAB JIT replacing the `feval` with a direct call results in speedups of 1.00 – 1.23, and for the McVM JIT the results are 1.31 – 18.88. Inlining the direct call results in large speedups for the MATLAB JIT of 1.11 – 10.65 and for the McVM JIT the results are 1.32 – 19.22.

One might be surprised that the overheads for both `feval` calls and ordinary calls appear to be so high for MATLAB. There are two reasons for this. First, the lookup semantics for function calls in MATLAB are quite complex, and without optimization they require a heavy-weight dynamic lookup based on the current directory, the current path, and the type of the dominant argument. Secondly, the presence of `feval` can disrupt the intra- and inter-procedural analyses needed to accurately approximate dynamic types and array shapes, which is a key factor in generating efficient code.

Focusing on the JIT results, it appears that the McVM JIT can achieve more benefit than the MATLAB JIT by just replacing an `feval` call with a direct call, even without inlining. This is because McVM does on-the-fly interprocedural shape analysis and function specialization, which is enabled as soon as the `feval` is converted to a direct call. Although we do not have access to the implementation of Mathworks' MATLAB JIT, these results would seem to indicate that the MATLAB JIT is not doing a similar interprocedural analysis and that it requires inlining to get a similar benefit.

Since we see potential speedups for all systems, for both interpreters and JITs, there does seem to be an important optimization opportunity for dynamically specializing `feval` calls to direct calls, and then potentially inlining those direct calls. We present our approaches to this problem in Sections 4 and 5.

3. Background

In this section, we provide key background and overview of the infrastructure on which we are building, namely McVM and McOSR.

3.1 McVM and McJIT

The Mathworks implementation of MATLAB is a closed-source proprietary product, so we are not able to experiment its implementation. In contrast, McVM is an open source implementation of a VM with an LLVM-based JIT, which also has support for OSR, and is thus suitable for this research.

McVM composed of a JIT compiler known as McJIT and an interpreter. The JIT compiler can switch to the interpretation mode for the evaluation of some complex expressions, or for functionality unsupported by the JIT. The interactions between these two components is facilitated via a symbol environment. McJIT is built upon the LLVM framework [1, 15], and as such it generates LLVM IR. The LLVM system performs the low-level optimization and code generation to produce target machine code. The techniques presented in this paper operate entirely on the McJIT and LLVM IRs, and do not require any modification of machine code. Thus, the techniques are portable across different target architectures.

3.2 OSR Background

McVM has support of OSR [13, 14] which works completely at the LLVM IR level. The main idea is that LLVM IR instructions can be tagged as interesting, and OSR points can be inserted on any loop that encloses the tagged instructions. Each OSR point is associated with an LLVM-IR transformer, which is applied when the OSR point triggers. The OSR library takes care of saving the appropriate state, and restarting the transformed code at the appropriate location and state. In the next section, we provide the details of how we leverage the OSR machinery to optimize `feval`.

4. OSR-based `feval` Specialization

We have developed our techniques for optimizing `feval` calls in the open source MATLAB JIT, McJIT. Although the details of our approach are specific to McVM/McJIT, the ideas should be applicable to other similar systems. Developing our solutions in an open system allows other researchers to examine our solution, and to build upon it. We start this section with a discussion of the existing approach to implementing `feval`. We then propose a general OSR-based specialization, and show how that can be implemented with McJITs OSR library. In the following section, we develop an alternative specialization approach that does not depend on OSR, but can be easily integrated into systems which perform on-the-fly function specialization based on the values in input arguments.

4.1 Existing McJIT approach for `feval`

When McJIT encounters a MATLAB statement involving a call to `feval`, it generates LLVM code to call to a dynamic dispatcher. For example, for the `feval` statement at line 22 of Listing 1, it generates the code in Listing 3. Let us examine this code snippet. The compiler generates the code to save the arguments to the `feval` call into an array of objects. This is shown in lines 1–5. Then, it generates the call to the dynamic function dispatcher, that is, the call to Interpreter :: callFunction in line 6.

```
1  %argsPtr = call i8* @"ArrayObj::create"(i64 2)
2   call void @"ArrayObj::addObject"(i8* %argsPtr,
3                                     i8* %arg1)
4   call void @"ArrayObj::addObject"(i8* %argsPtr,
5                                     i8* %arg2)
6  %retVal = call i8* @"Interpreter :: callFunction"
7                                    (i8* %funcPtr,
8                                     i8* %argsPtr,
9                                     i64 %nargout)
```

Listing 3. LLVM code generated for an `feval` call.

Processor: Intel®Core^{TM}i7-3930K CPU @3.20GHz
RAM: 16 GB;
Cache Memory: L1 32KB, L2 256KB, L3 12MB;
Operating System: Ubuntu 12.04 x86-64;
LLVM Compiler framework: version 3.0;
McJIT: version 1.1; McOSR: version 1.1;
GNU Octave: 3.0.5;
MATLAB: Version 7.12.0.635 (R2011a) 32-bit (glnx86).

(a) Experimental Setting

BM	Description	#`feval` (total)	#feval[a] (loops)
bisect	Uses bisection to find a root of the scalar equation f(x) = 0	3	1
newton	Newton's method to find a root of the scalar equation f(x) = 0	1	1
odeEuler	Euler's method for integration of a single, first order ODE	1	1
odeMidpt	Midpoint method for integration of a single, first order ODE	2	2
odeRK4	Fourth order Runge-Kutta method for a single, first order ODE	4	4
gaussQuad	Composite Gauss-Legendre quadrature	1	1
sim_anl	Minimizes a function with the method of simulated annealing	2	1

[a]This is the total number of `feval` calls that are located in all the loops of a benchmark.

(b) Benchmarks

	Interpreter				
	`feval` (F) t(s)	direct (D) t(s)	inlined (I) t(s)	Speedup F/D	F/I
bisect					
Octave	19.94	17.36	12.85	1.15	1.55
MATLAB	5.43	4.85	2.40	1.12	2.26
McVM	3.60	3.60	2.40	1.00	1.50
newton					
Octave	19.04	16.60	11.02	1.15	1.73
MATLAB	6.23	5.64	3.13	1.10	1.99
McVM	6.20	4.80	3.73	1.30	1.66
odeEuler					
Octave	32.86	28.56	18.41	1.15	1.78
MATLAB	12.63	11.56	6.38	1.09	1.98
McVM	7.05	6.81	4.52	1.03	1.56
odeMidpt					
Octave	54.85	46.65	25.22	1.18	2.17
MATLAB	20.75	18.29	7.76	1.13	2.67
McVM	11.31	11.01	6.61	1.03	1.71
odeRK4					
Octave	101.80	82.74	40.45	1.23	2.52
MATLAB	36.09	31.25	10.68	1.15	3.38
McVM	21.10	19.95	11.33	1.06	1.86
gaussQuad					
Octave	20.12	17.97	14.22	1.12	1.42
MATLAB	13.29	12.90	9.89	1.03	1.34
McVM	3.77	3.71	2.90	1.02	1.30
sim_anl					
Octave	23.81	22.61	20.33	1.05	1.17
MATLAB	16.14	16.15	14.52	1.00	1.11
McVM	4.48	4.45	3.93	1.01	1.14

(b) Interpreter overheads

	JIT				
	`feval` (F) t(s)	direct (D) t(s)	inlined (I) t(s)	Speedup F/D	F/I
bisect					
Octave	*	*	*	*	*
MATLAB	2.99	2.63	0.28	1.14	10.65
McVM	2.38	1.67	1.07	1.41	2.22
newton					
Octave	*	*	*	*	*
MATLAB	3.52	3.20	0.71	1.10	4.98
McVM	2.60	1.40	0.73	1.85	3.56
odeEuler					
Octave	*	*	*	*	*
MATLAB	2.65	2.40	2.11	1.11	1.26
McVM	4.61	0.58	0.73	7.97	6.29
odeMidpt					
Octave	*	*	*	*	*
MATLAB	3.21	2.91	2.17	1.10	1.48
McVM	7.10	0.67	0.65	10.56	10.91
odeRK4					
Octave	*	*	*	*	*
MATLAB	4.07	3.31	2.22	1.23	1.84
McVM	12.79	0.68	0.66	18.88	19.22
gaussQuad					
Octave	*	*	*	*	*
MATLAB	3.92	3.69	2.42	1.06	1.62
McVM	1.27	0.97	0.96	1.31	1.32
sim_anl					
Octave	*	*	*	*	*
MATLAB	3.38	3.31	2.22	1.00	1.11
McVM	3.47	2.51	2.21	1.38	1.57

(c) JIT overheads

Figure 1. Benchmarks and `feval` overheads

When the dispatcher is called at run time, it examines its first argument to determine that this is an `feval` call site. It then calls the library function `feval` passing it its own second argument — the array containing the arguments to the `feval` call. The `feval` library examines its own first argument and resolves the right function to dispatch[5] and then prepares the input arguments needed by this function and calls the function. The result of executing this function is what the dispatcher eventually returns in line 6.

[5]Note that this resolution itself can be quite expensive as the function lookup rules for MATLAB are quite complex and depend on the current directory, the current path, and the run-time type of the dominant argument.

The foregoing procedure can be slow, and furthermore it inhibits function inlining and other flow analyses. However, since the value of the function that `feval` built-in evaluates at run time cannot be determined statically in general, this implementation represents what is typically done to implement the `feval` library function.

Even though the function binding cannot be determined statically, it is often the case that the function binding and the argument types of the function called by `feval` do not change through the whole loop execution, or even through the whole method execution, as is the case for the typical example in Listing 1. For this class of MATLAB programs, we can improve the runtime performance if it is possible to dynamically do on-the-fly code transformation and function specialization and possibly inlining.

Listing 4 shows a code snippet with an `feval` call in a potentially long-running loop. The idea of our OSR-based `feval` specialization is to transform a loop such as that shown in Listing 4 to that in Listing 5 at run time when the value of f is known. The approach transforms the call via the `feval` into a direct call to the compiled function given by the runtime value of f. This is shown in Listing 5 as f_V. To maintain correctness we will need some safety checks (*guards*) that will backup to the general case (i.e., the call via the `feval`) if the current call does not match the last specialized version (the version may be incorrect if the function argument (e.g., f) to `feval` changes, or the types of the other arguments change). Thus, another key challenge is minimizing the overhead for the checks.

```
n = 1000000;
for i=1:n
   ...
   k = feval(f,x);
   ...
end
```

Listing 4. Normal

```
n = 1000000;
for i=1:n
   ...
   if (guard)
      k1 = f_V(x);
   else
      k2 = feval(f,x);
   end
   k = phi(k1, k2);   ...
end
```

Listing 5. Specialized

4.2 Using McJIT OSR

McJIT has support for on-stack-replacement (OSR) [14]. This allows us to develop an OSR-based optimization which will optimize hot loops which contain `feval` calls on the fly.

Our solution strategy has three important steps, the first two steps are done at JIT-compilation time (for example, when function *newton* is first JIT-compiled), whereas the third step happens at runtime (for example, when the while loop inside of *newton* executes). The high-level description in this subsection provides an overview of the approach, and the subsequent subsections provide details of how it is implemented in McJIT/McOSR.

Dispatcher call annotation: During JIT-compilation of a function body, all dispatcher calls that correspond to `feval` calls in a loop must be identified and marked. This is discussed in detail in Section 4.3.

OSR instrumentation: If the first phase identifies some `feval` dispatcher calls, then the closest enclosing loop of each such dispatcher call must be instrumented to include a conditional OSR trigger, usually based on the number of loop iterations. In addition, an OSR point must be inserted, where the OSR point is associated with the `feval` optimizing transformation. We discuss this further in Section 4.4.

Triggering an OSR event at run time: At run time, if an OSR is triggered by a running function, the code transformer attached to that OSR point will be executed. In our approach, this is where the `feval` optimizing transformation is actually performed. This transformation must rewrite the LLVM IR to replace the annotated `feval` call with the appropriate direct (or inlined) call, and it must also insert appropriate guards to ensure that the specialized call is only executed for the correct specialized function and argument types, and it must backup to the general case otherwise. We give a detailed description of the code transformer in Section 4.5.

4.3 Dispatcher Call Site Annotation

To label the statements of interest for the OSR transformation, `feval` call sites are annotated with the OSR ID of their closest enclosing loop. For example, for the `feval` call in Figure 2(a), the following would be generated:

```
%retV = call i8* @"Interpreter :: callFunction"(i8* %funcPtr,
i8* %argsPtr, i64 %nargout), !FI !OSR1
```

where !FI and !OSR1 are the metadata used to annotate the call sites with the call to the dispatcher for an `feval` call. The string !OSR1 indicates that this call site will be considered for an `feval` optimizing transformation if OSR is triggered in the loop identified with OSR ID 1.

We also assign a unique ID to each `feval` call site. This ID is used to index a fixed memory area for caching the types that the arguments to the dispatcher had just before OSR is triggered at run time. To facilitate this process, a store instruction of the following form is generated:

```
store i8* %argsPtr, i8** addrOfCacheSlot, !FI
```

which stores the pointer to the array of objects passed to the dispatcher to a fixed cache slot associated with the current `feval` call. Notice that this instruction is also annotated with the same metadata as the call to the dispatcher.

The metadata !FI encapsulates some JIT-time information about the arguments of the associated `feval` call. It is a 3-tuple. The first operand or field is the unique ID assigned to this `feval` call; the second and the third represent relevant JIT-time facts about the `feval` call site. We defer the discussion on the information collected at the JIT-time to Section 4.6.

The annotations attached to the call to the dispatcher are consumed by the code transformer during an OSR event. We discuss the transformer in more detail in Section 4.5.

4.4 OSR Instrumentation

At JIT compilation time for a function, if a loop contains an `feval` call, the loop must be instrumented with a test that determines whether a loop counter has reached a given threshold. This is the OSR condition. We experimented with a threshold value set at 2. So, at run time, after the execution of the second iteration of the loop, the OSR condition will be satisfied. The conditional execution of the OSR point is achieved by generating the following LLVM conditional instruction at end of the loop header.

```
br i1 %osrCond, label %OSR, label %LB
```

This instruction inspects the OSR condition (**%osrCond**) and branches to the basic block named **%OSR** (which triggers the OSR) if the test is successful. Otherwise, it branches to **%LB** where the body of the loop will be executed as normal.

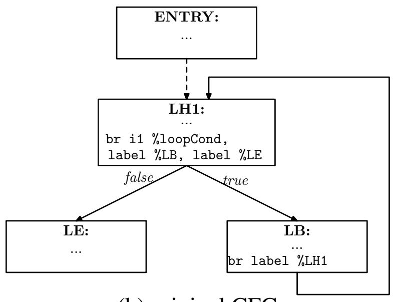
For our `feval` optimization, we use a closest-enclosing-loop strategy for the placement of an OSR point. The McOSR library requires that each OSR point is associated with a code transformer - it is this transformer that will execute when the OSR triggers. Thus, our `feval` optimizing transformation logic is implemented by the code transformer that we attach to the inserted OSR point.

Figure 2(a) shows a code snippet from our running example, and in Figure 2(b), we show in a simplified form the corresponding control flow graph (CFG) in LLVM IR. *LH1* is the loop header block and terminates with a conditional branch instruction. The basic block branches to the loop body at *LB* or the loop exit block at *LE* depending on the loop exit condition (*%loopCond*).

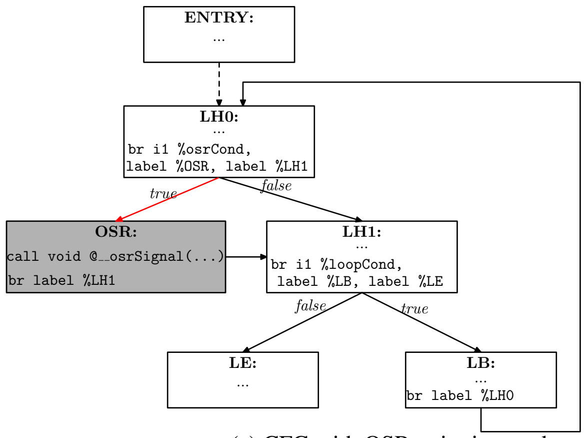
The CFG shown in Figure 2(b) is transformed into that shown in Figure 2(c) after inserting an OSR point. As can be observed from the figure, the loop header block now contains the instruction to compute the OSR triggering condition (**%osrCond**) and terminates with a conditional branch instruction as discussed earlier.

```
  ...
 while k <= maxit
    k = k + 1;
    [f, dfdx] = feval(fun,x);
     ...
 end
end
```

(a) source `while` loop



(b) original CFG



(c) CFG with OSR point inserted

Figure 2. Inserting OSR points into CFG

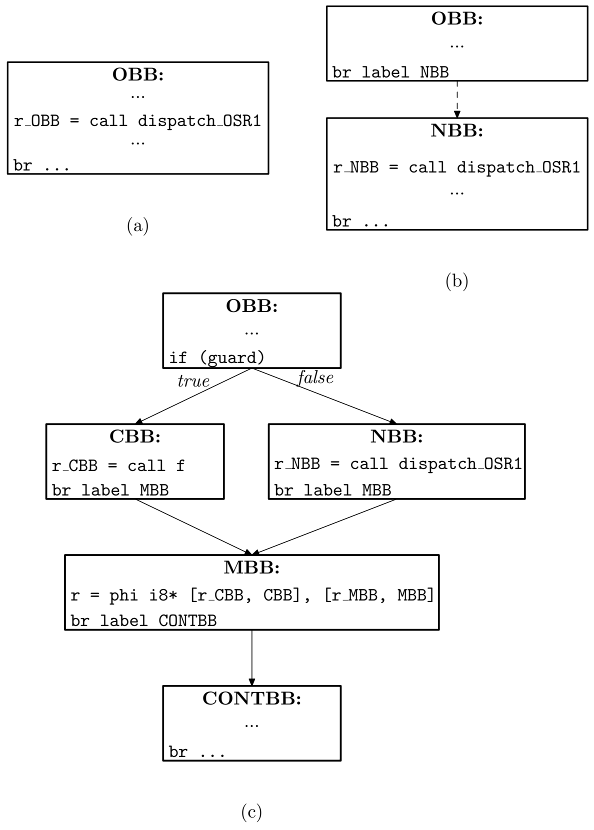


Figure 3. Actions of the code transformer. Basic block *OBB* in (a) is split into two. The result of the splitting process is shown in (b). In (c), *NBB* is split into *NBB* and *CONTBB*; two new basic blocks have been inserted into the CFG: *CBB* contains a call to the compiled function (f), and *MBB* merges the results from the call in *CBB* and the original call to the dispatcher in *NBB*.

4.5 OSR Triggering and Runtime Transformation

At the heart of our implementation is the code transformer that is attached to an OSR point. When an OSR is triggered at run time, the OSR runtime system passes control to the code transformer. This is where our `feval` optimizing transformation is performed.

The code transformer first traverses its input function (i.e, the LLVM IR of the running function) and collects all the calls to the dispatcher that are associated with an *feval* call site in the source program. It identifies these call sites using the OSR label attached to such instructions at their creation times. It also finds and removes all the store instructions that were inserted to cache the last-known types for the arguments to the dispatcher.

The transformer then processes the call instructions as follows. For each dispatcher call, the transformer retrieves the pointer to the array of pointers to the last arguments passed to the dispatcher. Using this pointer, the transformer determines the function being dispatched — the *fef*. However, if the cache slot is unset, the code transformer continues with the next annotated dispatcher call.

Having determined the *fef* at this call site, the transformer begins a series of transformations at the basic block containing the current call. We illustrate these transformations in Figure 3.

Figure 3(a) shows a basic block (*OBB*) with a call to the dispatcher, represented with *dispatcher_OSR1*. The call is annotated with OSR label *OSR1*.

The transformer first splits the original basic block (*OBB* in Figure 3(a)) to obtain the basic blocks shown in Figure 3(b). In Figure 3(b), the call to the dispatcher in *OBB* has been moved into the beginning of a new basic block named *NBB*. Later, the transformer calls the compiler to generate specialized code for the *fef* that corresponds to the current set of argument types. Let us call such a newly compiled function f. Note that the code transformer may choose to inline f if it considers it as a good inlining candidate. It can also perform further optimizations on the calling function. After the compilation, the transformer creates a new basic block and generates the instructions to call f. This new block is shown in Figure 3(c) as *CBB*.

Now, we have two alternative paths to evaluating function f: (1) via a direct call in *CBB* and (2) via the call to the dispatcher in *NBB*. To link *CBB*, the code transformer splits *NBB* (of Figure 3(b)) after the call to the dispatcher to obtain a new basic block *CONTBB*. This is the continuation block for both *NBB* and *CBB*. Because the code in *OBB* (Figure 3(b)) is always executed before the call to the dispatcher in the original *OBB* (Figure 3(a)), it must follow that the current *OBB* dominates both *CBB* and *NBB*. Thus, the code transformer terminates *OBB* with a runtime *guard*. We discuss the *guard* in the next section. The transformer also creates a new basic block named *MBB*. As shown in Figure 3(c), *MBB* merges the results from *CBB* and *NBB* via a *phi* instruction generated by the code transformer. *MBB* terminates with a branch to the continuation block, *CONTBB*, as shown in Figure 3(c).

The transformer essentially implements our OSR-based `feval` optimization. To some degree, the runtime performance depends on the cost of evaluating the *guard* that determines the execution path taken at run time. We now discuss the functions of the *guard*.

4.6 Runtime Guards

The code transformer generates a runtime guard (shown in Figure 3(c)) that will determine the path taken by the program at run time. It chooses from among several guards depending on the quality of the metadata it retrieved from the call instruction that calls the dispatcher. In Section 4.3, we mentioned that we collect a variety of JIT compilation-time facts on `feval` call sites in the *!FI* metadata. The second component of the metadata is an unsigned integer that encodes three bits of information, corresponding to the following queries.

1. Is the first argument to an `feval` call a read-only variable in the function? We shall denote this query with *ROQ*.
2. Is the first argument a loop constant variable? We shall use *LCQ* to denote this query.
3. Do all the arguments to the `feval` call have a fixed runtime type? We shall denote this with *FTQ*.

The first two pieces of information are computed at JIT compilation time using standard flow analyses. The third is computed using McJIT's type inference [5], which starts with the actual runtime types for all arguments to the function and infers a set possible types for each variable at every program point. Therefore at the call to an `feval`, the type-inference can determine the set of possible types for all the arguments to the `feval` call. If only one type exists in the type set for each argument, then *FTQ* is true.

The combination of these queries guides the choice of the guards generated by the transformer. If *ROQ* is true, we can move the part of the computation of the guard (to determine whether or not the runtime value of this argument corresponds to the function that will be called at *CBB* shown in Figure 3(c)) to the function's entry block.

If *LCQ* is true, we can compute the guard outside the loop and use the result to determine the path taken by the program after *OBB*. If *FTQ* is true, it means that all the arguments are monomorphic and we can eliminate the check that determines whether the type of any argument changes at run time. We discuss this further below.

Let

f**:** denote the first argument to an `feval` call;

P**:** denote the set of the remaining arguments $p_2, p_3, ..., p_n$ to the `feval` call;

`lastValue`(f): denote the cached value of f;

`newValue`(f): denote the current value of f;

`lastType`(p): denote the cached type of variable p;

`newType`(p): denote the current type of variable p.

FEB: be the entry basic block of a function containing an `feval` call; and

LEB: be the entry basic block of a loop with an `feval` call.

We enumerate in Table 1, the different possible guards (based on the three queries) that the code transformer can generate together with the optimal point to compute a guard.

To simplify the table, we define

$$f\_cond = \texttt{lastValue}(f) == \texttt{newValue}(f)$$
$$a\_cond = \forall(p \in P), \texttt{lastType}(p) == \texttt{newType}(p)$$

and write *f_cond* (*FEB*) if *f_cond* should be computed at the entry basic block of the function containing a corresponding `feval` call.

#	*ROQ*	*LCQ*	*FTQ*	**Guard**	**Compute Point**
1	T	T	T	*f_cond*	*f_cond* (*FEB*)
2	T	T	F	*f_cond* ∧ *a_cond*	*f_cond* (*FEB*); *a_cond* (*OBB*)
3	T	F	T	*	*
4	T	F	F	*	*
5	F	T	T	*f_cond*	*f_cond* (*LEB*)
6	F	T	F	*f_cond* ∧ *a_cond*	*f_cond* (*LEB*); *a_cond* (*OBB*)
7	F	F	T	*f_cond*	*f_cond* (*OBB*);
8	F	F	F	*f_cond* ∧ *a_cond*	*f_cond* (*OBB*); *a_cond* (*OBB*)

Table 1. Guard truth table (a "*" denotes an impossible result).

Let us examine Table 1. In the first case (i.e., table row 1), *ROQ*, *LCQ*, and *FTQ* are true, in this case, only *f_cond* should be computed and can be done at *FEB*, that is, the calling function's entry basic block. *FTQ* is true. Thus, we know that the runtime type of each argument at the `feval` call site is fixed so, there is no need to include *a_cond* in the *guard* that is evaluated at *OBB*.

In Case 2 (i.e., table row 2), the required guard that the code transformer must generate is: $guard = f\_cond \ \wedge \ a\_cond$. This is because the type of each argument to f may change at run time. Further, if after transforming the code, the value of f changes (i.e., in a subsequent call of the function with the `feval` call), the backup path must be taken. The *f_cond* component of the guard can be evaluated at the function's entry basic block because f is read-only in the calling function. It must be a parameter of the function. However, because the types of the arguments may change before the `feval` call site, the second element of the guard, *a_cond*, must be evaluated just before the use of the guard in basic block *OBB*.

Cases 3 and 4 represent impossible cases because it cannot be that f is a read-only variable in the calling function and at the same time not be a loop constant in that function.

In Case 5, only *f_cond* should be computed and can be done at *LEB*.

Case 6 is similar to Case 2 except that *ROQ* is false, meaning that f is not a read-only variable but it is a loop constant. For this reason, like Case 2, the required guard is $guard = f\_cond \wedge a\_cond$. Unlike Case 2, however, the optimal point to compute *f_cond* is *LEB*. The second element (*a_cond*) must still be computed at *OBB*.

In Case 7, we know that the arguments have constant types at the `feval` call site. But we also know that f is neither a read-only nor a loop constant. So, the required guard is to evaluate only *f_cond* at *OBB* before the use of the guard.

Case 8 requires that both *f_cond* and *a_cond* be computed at *OBB* before the use of the guard in the block. This is because f is neither a read-only nor a loop constant variable. Further, the types of the arguments may change at run time as indicated by the value of *FTQ* in row 8 of Table 1. Observe that this is the most expensive guard computation the code transformer can generate.

The least expensive guard is in Case 1. This is the ideal case. In the worst case (Case 8), the code transformer inserts a relatively expensive guard at the end of *OBB* that tests whether the current runtime value of *fef* (of an `feval` call) corresponds to the compiled function and that the remaining arguments have stable types. This may have an impact on performance, although we believe this seldom happens within the class of the applications that we have considered.

4.7 Resuming Execution after an OSR is Triggered

You will note that we have only focused on defining the OSR points and the transformation that occurs when an OSR triggers, but have not defined how the newly transformed code is executed and how the state is restored or how control flow is correctly resumed. These important details are handled automatically by McOSR [14].

5. JIT Value-based Specialization

In the previous section we presented an OSR-based approach to specializing an `feval`. This works by intercepting a running function when it evaluates an `feval` in a loop, specializing the code for the `feval` call, inserting the correct guards, and then restarting the execution with the specialized code. For example, in our motivating example, *newton* from Listing 1, the execution of the loop would be intercepted, the `feval` specialized to a direct call, and the execution resumed.

This method is general in that the value of the first argument to `feval` (the *fef*) can be defined anywhere, it could be an argument of the enclosing function, it could have been stored in a data structure, or it could have been computed.

The technique presented in this section concentrates on an alternative approach for the most common case, when the *fef* is a parameter of the enclosing function, as is the case in *newton*. McJIT already has a mechanism for compiling specialized versions of functions based on the run-time **type** of the arguments [5]. We wanted to go even further, and specialize code based on the run-time **value** of arguments that correspond to an *fef*.

Unlike the OSR-approach which intercepted the function containing the `feval` while it was running, the JIT-specialization approach specializes the function before it is called. For example, a call of the form *newton('fn3n', 3, 5e-16, 5e-16)* would first check to see if a version of *newton* matching the **value** 'fn3n' for the first arguments and matching the **types** of the other arguments had already been compiled. If so, it would call the previously compiled version, and if not it would compile a specialized version, and call this new version.

To further illustrate this idea, consider Listing 6 and Listing 7. In Listing 6, function h calls function g passing it an argument defined in function h as a reference to function *myFunc*. Function g accepts a function as an argument and contains an `feval` call that evaluates the argument at some value x.

```
1  function h()
2    f = @myFunc; ...
3    g(f);
4  end
5
6  function g(f)
7    n = 1000000;
8    for i=1:n
9      ...
10     k = feval(f,x);
11     ...
12   end
13 end
```

Listing 6. Normal

```
1  function h()
2    f = @myFunc; ...
3    dispatch(g,f);
4  end
5
6  function g_myFunc(f)
7    n = 1000000;
8    for i=1:n
9      ...
10     k = myFunc(x);
11     ...
12   end
13 end
```

Listing 7. Specialized

The main idea of the JIT value-based `feval` specialization is to replace a call to a function containing an `feval` call in a long-running loop with a call to a special dispatch function. When the dispatch function (called the dispatcher for short) is called at run time, it will evaluate the value of the parameter that corresponds to an *fef*. If not already compiled, it will then generate a new version with all the `feval` calls replaced with direct calls to the *fef*.

Thus, the call of *dispatch* in line 3 of Listing 7, will cause the dispatcher to generate a new version of g shown as *g_myFunc*. Notice that the call to `feval` in g has been replaced with a direct call to *myFunc* in *g_myFunc*.

To enable this new form of specialization we added a pass to McJIT so that when it builds the AST for a program, it analyzes all the functions in the compilation unit and annotates those with an `feval` call, whose *fef* is a read-only parameter of the enclosing function. We then changed how McJIT compiles calls to such annotated functions. Normally, after McJIT has compiled the right version of a function at a call site, it inserts the corresponding LLVM call instruction into the current basic block. However, to support the runtime code specialization for `feval`, we modified McJIT so that it does not insert the call instruction but, instead, generates a new instruction of the form

```
call void @''JITExt::dispatchFunction''(i8* %baseIRPtr,
                                        i8* %fefValue,
                                        i8* %inArgsPtr,
                                        i8* %retValsPtr,
                                        i32 %csID)
```

that calls the dispatcher. The dispatcher, that is, function *JITExt::dispatchFunction*, accepts five arguments:

(1) the first is the pointer to the base IR (i.e., the original version of the IR (AST)) that corresponds to the called function;

(2) the second is a pointer to the argument that corresponds to the *fef* (i.e., the first parameter) of a marked `feval` call in the called function;

(3) the third is a pointer to a structure containing the input arguments to the called function;

(4) the fourth is a pointer to a structure containing the return values;

(5) the last argument is an integer that denotes the index of a cache slot where a pointer to the descriptor of the AST can be located.

Each AST representing a function with an `feval` call has one or more code cache descriptors. A code cache descriptor contains information related to the code of the AST that corresponds to the types of the arguments passed to the function at a call site.

A function that is called with different argument types at different call sites has a code cache descriptor for each call site. A code cache descriptor is a four-tuple.

$$\begin{aligned} \textit{descriptor} \;=\; & < \textit{entry\_address}, \textit{argument\_types}, \\ & \quad \textit{counter}, \textit{feval\_versions} > \end{aligned}$$

where *entry_address* is the address of the entry to the compiled code corresponding to the AST of the called function. We shall denote the called function at a call site with f. Field *argument_types* denotes the types of the arguments at the call site. Due to McJIT's code specialization on argument types at call sites, the set of types for the arguments at a call site is immutable. Field *counter* denotes a compilation counter that counts the number of versions that are generated at different consecutive executions of the call to the dispatcher instruction. Field *feval_versions* is a map containing (*AST* , *entry_address*) pairs. The first member of the pair is the IR corresponding to the value of the parameter used as the first argument to some `feval` calls in f. The second member of the pair is the address of the entry point to the compiled code of f that corresponds to an *fef*.

5.1 Functions of the Dispatcher

At run time, the dispatcher first uses a combination of its first parameter (i.e., the AST) and its last parameter (i.e., the cache slot index) to retrieve the code cache descriptor that matches the argument types at the current call site. This is shown in line 1 of Algorithm 1. Then, in line 2, the dispatcher performs a look-up using its second parameter to determine whether a corresponding code version had been generated.

If the look-up is successful, the dispatcher executes (in line 13 of Algorithm 1) the function at the address returned by the look-up.

Otherwise, the dispatcher compares the current value of the counter in the code cache descriptor with a given *threshold*. If the counter has exceeded the threshold, the dispatcher executes the initial code generated for the AST at this call site. This is shown in line 15 of Algorithm 1. If the counter is below the threshold, however, the dispatcher clones the original AST and replaces all the marked `feval` calls with direct calls to the evaluated function given as its second parameter. After, the dispatcher retrieves the types attached to this call site and calls the compiler to compile and generate the correct code matching the argument types at this call site. These actions are performed in lines 3 – 11 of Algorithm 1.

```
input  : baseIR, fef, inArgPtr, outArgPtr, cacheSlot
output: void
1  ci ← getCodeCacheInfo (baseIR, cacheSlot);
2  entryPoint ← lookupFunction (ci, fef);
3  if entryPoint == NULL AND ci.counter <=
   THRESHOLD then
4      newIR ← clone (baseIR);
5      replaceFevalCalls (newIR, fef);
6      llvmIR ← compileFunction (newIR,
       ci.argTypesStr);
7      entryPoint ← compCallWrapper (llvmIR,
       newIR, ci.argTypesStr);
8      // insert an entry for a new version into the cache;
9      putNewVersion (ci, getFunction (fef),
       entryPoint);
10     ci.counter ← ci.counter + 1;
11 end
12 if entryPoint ≠ NULL then
13     call entryPoint (inArgsPtr, outArgsPtr);
14 else
15     call ci.entryPoint(inArgsPtr, outArgsPtr);
16 end
```

Algorithm 1: dispatch function

After the compilation of a new version, the dispatcher inserts an entry — that is, a pair comprising of the AST corresponding to the current value of the *fef* and the entry point address of the compiled code — into a map in the code cache descriptor of the base IR. This action is performed by the call of function *putNewVersion* in line 9 of Algorithm 1. The dispatcher does this so that if the function is called again with the same *fef* value, it can retrieve and execute the correct code. Finally, the dispatcher updates the counter associated with the cache slot descriptor.

Although the base AST and new versions of the AST have the same number of input and output parameters, the types of the values returned by the compiled code that corresponds to a given *fef* may be different. This presents a problem in that the rest of the code of the calling function was generated using the information obtained from the base AST. We resolved this problem by generating a wrapper (line 7 of Algorithm 1) that converts from the types returned by a new version to the types used in generating the code for the original version. Because of this problem, we always call the code that matches an *fef* via a wrapper.[6] A wrapper is a short function. It is composed of a call instruction and the instructions that convert the return values to their expected types.

A code cache look-up miss causes a compilation of a new version if the value of the counter in the code cache descriptor has not exceeded the threshold. After the counter has exceeded the given threshold, the dispatcher stops compiling new versions. Thus, for a new *fef* value, the dispatcher then always executes the original code generated for the base AST of the called function. This scheme can prevent excessive compilation actions in cases where too many different functions are being called. However, this rarely happens in practice. So, we expect only a reasonable number of new versions to be generated.

Again, we stress that this approach only works in cases where the *fef* of an `feval` call in the called function is a read-only function parameter. This covers most of the programs under study.

5.2 General Dispatcher

We could also extend Algorithm 1 to cover more cases of JIT value-based specialization. We could generalize the input value *fef* from Algorithm 1 to be a general value V, and could generalize line 5 to apply an arbitrary transformer, and line 9 to store a general key.

As an example of an application of the general dispatcher, consider the MATLAB `eval` (many dynamic languages have a similar feature as well). The MATLAB `eval` built-in evaluates MATLAB code given as its input string expression. Like the `feval` specialization, in some cases, we can also specialize a function with an `eval` call whose input string is a parameter of the function by developing a suitable *IR transformer* for the specialization.

Another example is the specialization of a function with a parameter that is an array. We can specialize the function using the properties of the array, such as array bounds, to generate more efficient code for loops that operate on such arrays in the function.

6. Experimental Results

In Section 2, we demonstrated that `feval` resulted in significant overheads, and that replacing an `feval` by a direct call resulted in substantial speedups, which could be further increased by inlining the direct call. In this section we examine the performance improvements achieved through our two dynamic techniques: the OSR-based specialization presented in Section 4, and the dynamic value-based function specialization presented in Section 5. We examine both the benefits and limitations of each approach, and we compare their performance with the upper bound speedups provided under the hand-coded direct call and inlined versions.

6.1 OSR-based approach

In Table 2, the column labelled **Baseline** shows the results of executing the benchmarks with McVM JIT in the normal mode. The columns labelled **OSR-based Optimization** give the execution times for three variations of the OSR approach. *Opt0* gives the results when the benchmarks were run with our basic OSR-based `feval` optimization enabled. We also experimented with two further improvements. The column labelled *Opt1* shows the benchmarks with the OSR-based `feval` optimization plus a dynamic function inlining optimization that is performed when the OSR point triggers. *Opt2* is a further improvement where we first apply the dynamic inlining, and then apply a further optimization of the symbol table environment, which is sometimes enabled by the inlining. We describe this optimization in more detail in our discussion of the performance of this optimization.

[6] Instead of using a wrapper, our future implementations will use a specialized compiler that directly performs the type conversion in the generated specialized version.

	Baseline	OSR-based Optimization						Hand-coded	
	t(s)	t(s)			Speedup			Speedup	
Benchmark	Baseline(F)	Opt0	Opt1	Opt2	F/Opt0	F/Opt1	F/Opt2	F/D	F/I
bisect	2.38	1.93	1.92	1.93	1.23	1.24	1.23	1.41	2.22
newton	2.60	2.23	2.23	1.55	1.17	1.17	1.68	1.85	3.56
odeEuler	4.61	2.71	2.82	2.64	1.71	1.63	1.75	7.97	6.29
odeMidpt	7.10	4.22	4.18	4.15	1.68	1.70	1.71	10.56	10.91
odeRK4	12.79	7.35	7.46	7.36	1.74	1.72	1.74	18.88	19.22
gaussQuad	1.27	1.03	1.04	1.05	1.23	1.22	1.21	1.31	1.32
sim	3.47	3.40	3.36	2.98	1.02	1.03	1.16	1.38	1.57
Geometric Mean					1.37	1.36	1.47	3.58	4.16

Table 2. Overall results for OSR-based optimization in McVM JIT

From the results, we found that our `feval` optimization was effective. McJIT with the `feval` optimization consistently outperforms the standard McVM JIT on our benchmark set. The geometric mean of speedups at *Opt0* is 1.37. The dynamic inlining optimization enabled by *Opt1* does not improve performance on its own, but in combination with the subsequent symbol table optimization enabled for *Opt2*, there is an improvement, with a geometric mean speedup of 1.47.

At optimization level 2 (*Opt2*), we recorded the highest performance improvements with the *newton* and *sim* benchmarks. In McVM, the interaction between the compiled code and the interpreter is often facilitated through a symbol look-up environment. A symbol environment is a table that associates a value to a symbol. It is used to bind a value to a variable, and to look-up the value of a variable at run time. When needed, McJIT inserts the instructions to set up a symbol look-up environment for a function at the function's prologue. This can be a major source of overhead. After dynamic inlining, we perform an optimization that eliminates redundant set-up code. We found that this optimization was particularly effective in two of the benchmarks: *newton* and *sim*, which contained significant redundant setup code after inlining.

Although speedups of 1.47 are good, it is also important to examine if our dynamic optimization is approaching the upper bound speedups that we measured by hand-coding the direct call and hand-lining that call. The last two columns show the speedups we had measured for the hand-coded versions, and we see that the geometric mean speedups were 3.58 for the direct call and 4.16 for the inlined call. Thus, there is still a significant gap between what the dynamic technique achieves and the upper bound.

To see why this is the case, we examined the kinds of the runtime guards and the LLVM code generated for our benchmarks. We show the kinds for each benchmark in Table 3, with column *# feval (in loop)* showing the number of `feval` calls in the loops of a benchmark. We show the kinds of the runtime guards generated for the `feval` calls in a benchmark under column *Types of Guards*.

We can see from Table 3 that a somewhat expensive guard — one that checks the value of the *fef* passed in at the *entry* basic block and the types of *all* the arguments to an `feval` call in a loop — is generated for each `feval` call in the *ode* benchmarks. This is the case because the type inference engine infers that the type of at least one of the arguments is variable or *unknown*. This can be a source of runtime overhead. In addition, because the type-inference infers that the type of an argument to the target function of each `feval` call in the *ode* benchmarks is variable, the LLVM code generated for the *ode* benchmarks is less efficient. This is the main reason for the relatively lower performance recorded for the OSR-based version running the actual *ode* benchmarks.

Benchmark	# `feval` (in loop)	Types of Guards
bisect	1	Case 1[a]
newton	1	Case 2[b]
odeEuler	1	Case 2
odeMidpt	2	Case 2
odeRK4	4	Case 2
gaussQuad	1	Case 1
sim_anl	1	Case 1

[a] According to Table 1, Case 1 means that only the value of the *fef* is checked at the function's entry basic block. The types of the arguments to the `feval` call are stable.

[b] According to Table 1, Case 2 means that the value of the *fef* is checked at the function's entry; while the types of all the arguments are checked in the loop containing the `feval` call.

Table 3. Types of the runtime guards used by each benchmark.

6.2 JIT value-based-specialization approach

The OSR-based approach is general-purpose, and can operate on any `feval` within a loop. However, our results show that there is still a gap between the performance of the OSR-approach and the upper bound. The value-specialization approach applies to a common case where the *fef* of the `feval` call is a read-only parameter of the enclosing function. In these cases the value-specialization can generate a completely specialized version of the function, without the need for run-time guards, and in which the JIT-time type and shape analysis can operate more accurately.

In Table 4, we show the results of the value-based specialization in a context where we can compare it to both the hand-coded, and OSR-based results. The column labelled **VB-specialization** gives the time and the speedup relative to the baseline. We note that this gives excellent results, with speedups approaching the hand-coded upper bound for all the benchmarks. The value-based results gave a geometric mean speedup of 3.22, which is substantially better than the 1.37 for the OSR-based approach, and almost as good as the upper bound of 3.58.

Under the JIT value-based specialization approach, the specialized versions of the functions with `feval` calls may no longer contain `feval` calls. Thus, allowing McJIT to generate much more efficient code. The *odeRK4* benchmark has four `feval` calls within a long-running loop. These calls are replaced with direct calls in the specialized version generated at run time. Because the `feval` target function (*fef*) is now known, the type inference engine can analyze the function more precisely, and McJIT can then generate more efficient code for both the target and the calling functions.

Benchmark	**Baseline** t(s)	**OSR-based (OPT0)** t(s)	speedup	**VB-Specialization** t(s)	speedup	**Hand-coded (D)** t(s)	speedup
bisect	2.38	1.93	1.23	1.66	1.43	1.68	1.42
newton	2.60	2.23	1.16	1.61	1.61	1.40	1.85
odeEuler	4.61	2.70	1.71	0.67	6.86	0.58	7.97
odeMidpt	7.10	4.22	1.68	0.83	8.53	0.67	10.56
odeRK4	12.79	7.35	1.74	0.89	14.30	0.68	18.88
gaussQuad	1.27	1.03	1.23	0.90	1.41	0.97	1.31
sim	3.47	3.40	1.02	2.60	1.33	2.51	1.38
Geometric Mean			1.37		3.22		3.58

Table 4. Comparing Value-based specialization to OSR-based and hand-coded

6.3 Comparing the approaches

To understand in more detail why the value-based approach provides better performance, we need to examine the quality of the LLVM code generated for each benchmark, and the sources of overheads under the two approaches.

Under the OSR-based approach, McJIT generates less efficient code. This is so because McJIT generates a call to the interpreter for an `feval` call after *boxing* the arguments to make them more generic. In addition, because the called function (*fef*) at the call site is unknown during the compilation time, the type inference engine is unable to infer precise types for the values returned by the `feval` call, thus forcing the compiler to generate more generic instructions that are suitable for handling different types. This is a major source of inefficiency in the OSR-based approach.

Runtime guard computation can be expensive. The OSR-based approach generates runtime guards, which, as discussed in Section 4.6, depend on whether or not the arguments to an `feval` call have a fixed type. As mentioned in Section 6.1, for the three *ode* benchmarks, the type inference engine infers that the types to all the `feval` calls are variable, forcing the code transformer to generate an expensive guard for each `feval` call specialization.

We examined *odeRK4*. The code snippet for the only loop of the benchmark is shown in Listing 8.

```
for j=2:n
    k1 = feval( diffeq ,  t(j−1),       y(j−1)          );
    k2 = feval( diffeq ,  t(j−1)+h2, y(j−1)+h2∗k1 );
    k3 = feval( diffeq ,  t(j−1)+h2, y(j−1)+h2∗k2 );
    k4 = feval( diffeq ,  t(j−1)+h,   y(j−1)+h∗k3   );
    y(j) = y(j−1) + h6∗(k1+k4) + h3∗(k2+k3);
end
```

Listing 8. The *odeRK4* benchmark (from [19, 20]).

In the first `feval` call (line 2), the type inference engine infers that t(j−1) is a scalar floating point value. It, however, infers that y(j−1) can either be a scalar floating point value or a scalar complex value. In all the remaining three `feval` calls (lines 3 – 5), the type inference engine infers that the second parameter is a floating point value, but infers *unknown* for the third parameter.

Thus, in specializing the four `feval` calls in *odeRK4*, the code transformer inserts an expensive guard for each call specialization. The guards generated correspond to Row 2 of Table 1, that is, *f_cond* is evaluated at the function's entry basic block and *a_cond* is evaluated in the loop.

The JIT value-based approach is less affected by the foregoing issues. If all the `feval` calls in a function have the same *fef* and the *fef* is a read-only parameter of the function, then the specialized code generated to match the *fef* at run time will not contain any `feval` call implementation. Each `feval` call in the AST of the function would have been replaced with a direct call to the *fef*. This allows the type inference engine to analyze the called function, which, in turn, allows McJIT to further specialize the call site and generate efficient code. The `feval` calls in all the benchmarks have their *fef*s passed in as a parameter, thus contributing to the generation of the more efficient code for the specialized versions.

It is, however, true that the JIT value-based approach incurs some runtime overheads, including that of the code cache lookup. But this is small given the expected gains. Further, unlike the OSR-based approach that is limited to specialization of `feval` calls within a long-running loop, the JIT value-based approach can specialize a function with an `feval` call that occurs anywhere within the body of the function.

6.4 Summary

We conclude that although the JIT value-based approach is less powerful than the OSR-based approach, it is more effective on our benchmark set. The JIT approach only works where the *fef* is passed as a read-only parameter to a function. The OSR-based approach works in all cases but incurs much larger runtime overhead. It is possible to combine the two approaches in a JIT compiler by first analyzing a function with an `feval` call to determine whether a call of the function can benefit from the JIT value-based specialization approach. With speedups of up to 14 times faster, it would seem that such techniques are well worth incorporating into JIT compilers for MATLAB and other dynamic languages which have compute-intensive solvers which are abstracted over the computation function (*fef*).

7. Related Work

Historically, function dispatch in dynamic languages was implemented with a dispatch look-up table. This was found to be slow. More efficient approaches have emerged; they often employ a variety of caching techniques to speed up table look up. Smalltalk-80 [8, 12] uses a global cache to improve look up performance.

Our OSR-based approach is more related to the inline caching [6] approach used in another Smalltalk implementation. Interestingly, the implementation was based on several studies of Smalltalk programs that revealed that 95% of the time, the type of a Smalltalk message receiver is constant [6, 23, 24]. Our approaches to `feval` optimization are also based on the observation that `feval` calls in most MATLAB loops have unchanging first argument.

The inline caching technique used in the Smalltalk compiler involves caching the address of a looked-up method at the call site by modifying the compiled target code on-the-fly — by overwriting the call instruction. This allows the method to be called directly in a subsequent execution, avoiding the need for a look up. It also involves generating additional code (often called prologue) in the method that tests that the receiver type is correct before executing the body of the method. However, if the test does not succeed, it calls the look-up code.

Hölzle et al. extended the inline caching technique to handle polymorphic call sites by including more than one cached look-up result per call site. This technique is known as polymorphic inline caching (PIC) [9]. The PIC approach caches all the receiver types at a call site in a *stub* that is generated on-the-fly and rebinds the call to the stub routine.

In contrast to these approaches, our implementation is done completely at the LLVM-IR level, and not at target code level. Without on-stack replacement support [3, 7, 10, 13, 14, 18, 21], it is hard to cache previous function look-up result "inline"(i.e., at the call site). We also do not need additional code in the called function. We insert runtime guards so that execution can continue with the original call to the dispatcher if the guard fails. Also our backup path obviates the need to cache look-up results in a stub as in PIC case used in the implementations of SELF [4, 11].

Although multi-paradigm programming languages such as Python, JavaScript, and functional languages, including Lisp, Haskell, Scheme support higher-order functions, the function arguments are directly evaluated at run time and often lead to runtime code generation that is typically supported by polymorphic type inference, and sometimes, binding time analysis [17]. The MATLAB `feval` is an overloaded built-in that accepts a function name as a string or function handle and indirectly evaluates, at run time, the function argument. Our approaches are supported by a type-inference analysis, although it is explicit that the `feval` built-in evaluates functions only. Our approaches are aimed at improving JIT compiled code, and facilitating efficient compilation of the MATLAB `feval`, which can be extended to handle similar features in other dynamic languages, where it would have otherwise appeared impossible.

8. Conclusion

We proposed a general on-the-fly mechanism for specializing `feval` calls in hot loops using the OSR mechanism available in McVM, an open source research virtual machine for MATLAB. We demonstrated good performance improvements using the approach.

We introduced an effective JIT value-based specialization technique for optimizing `feval` calls, whose first argument is a function parameter. We showed how the JIT value-based `feval` specialization can be extended to handle more cases of JIT value-based specialization in a MATLAB JIT compiler. The approach can also be used for JIT value-based specialization in other similar dynamic languages. Indeed, the OSR-based approach can be so extended.

We collected a set of seven typical benchmarks that use `feval`, and demonstrated that our specialization approaches provide significant speedups over the base `feval` implementation for this benchmark set. In some cases the performance is near to the optimal performance of a hand-inlined function, but in other cases a gap remains. We would like to continue to develop new optimizations to further close that gap, and to apply the same sort of transformations to other dynamic features in MATLAB.

A somewhat surprising discovery in this work was the complex interplay between the JIT-time interprocedural type analysis and the on-the-fly transformations. The JIT value-based specialization can replace `feval` calls with direct calls in a function body, before doing the type analysis of that function body, thus leading to much better specialized code (because the interprocedural analysis can handle the direct calls much more precisely). On the other hand, this specialization can only happen at the function level, and only when the `feval` target function corresponds to a read-only parameter. The OSR-based method is more general, and can be applied at the level of loops, but suffers from less precise type information. It would be interesting to look at future work that combine the strengths of both approaches.

References

[1] LLVM. http://www.llvm.org/.

[2] G. Almási and D. Padua. MaJIC: Compiling MATLAB for Speed and Responsiveness. In *In PLDI '02*, PLDI '02, pages 294–303, New York, USA, 2002. ACM.

[3] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes Research Virtual Machine Project: Building an Open-Source Research Community. *IBM Syst. J.*, 44(2):399–417, 2005.

[4] C. Chambers and D. Ungar. Making Pure Object-Oriented Languages Practical. In *OOPSLA '91*, pages 1–15, 1991.

[5] M. Chevalier-Boisvert, L. Hendren, and C. Verbrugge. Optimizing MATLAB through Just-In-Time Specialization. In *CC '10*, pages 46–65, 2010.

[6] L. P. Deutsch and A. M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *In POPL '84*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM.

[7] S. J. Fink and F. Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-stack Replacement. In *CGO '03*, pages 241–252, 2003.

[8] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 2 edition, 1985.

[9] U. Hölzle, C. Chambers, and D. Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *In ECOOP '91*, ECOOP '91, pages 21–38, London, UK, UK, 1991. Springer-Verlag.

[10] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *PLDI '92*, pages 32–43, 1992.

[11] U. Hölzle and D. Ungar. A Third-Generation SELF Implementation: Reconciling Responsiveness with Performance. In *OOPSLA '94*, pages 229–243, 1994.

[12] G. Krasner. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, 1983.

[13] N. Lameed. McOSR: A Tool for Supporting On-Stack Replacement (OSR) in LLVM, 2012. http://www.sable.mcgill.ca/mclab/mcosr/.

[14] N. Lameed and L. Hendren. A Modular Approach to On-Stack Replacement in LLVM. In *VEE '13*, pages 143–154, 2013.

[15] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04*, pages 75–86, 2004.

[16] McLAB. The McVM Virtual Machine and its JIT Compiler, 2012. http://www.sable.mcgill.ca/mclab/mcvm_mcjit.html.

[17] H. R. Nielson and F. Nielson. Using Transformations in the Implementation of Higher-Order Functions. *Journal of Functional Programming*, 1:459–494, 1991.

[18] M. Paleczny, C. Vick, and C. Click. The Java HotSpot Server Compiler. In *JVM'01*, pages 1–12, 2001.

[19] G. Recktenwald. *Numerical Methods with MATLAB: Implementations and Applications*. Prentice Hall, 2000.

[20] G. Recktenwald. Numerical Methods with MATLAB: Implementations and Applications (Source Code Distribution), 2000. http://web.cecs.pdx.edu/~gerry/nmm/mfiles.

[21] S. Soman and C. Krintz. Efficient and General On-Stack Replacement for Aggressive Program Specialization. In *Software Engineering Research and Practice*, pages 925–932, 2006.

[22] The Mathworks. Technology Backgrounder: Accelerating MATLAB, September 2002. http://www.mathworks.com/company/newsletters/digest/sept02/accel_matlab.pdf.

[23] D. Ungar and D. Patterson. What Price Smalltalk? *Computer*, 20(1):67–74, Jan. 1987.

[24] D. M. Ungar. *The Design and Evaluation of a High Performance Smalltalk System.* MIT Press, Cambridge, MA, USA, 1987.