McGill University
School of Computer Science
Sable Research Group

Optimizing MATLAB `feval` with Dynamic Techniques

Sable Technical Report No. sable-2012-06-rev1

Nurudeen A. Lameed and Laurie Hendren

(Original version, December 17, 2012, Revised version (rev1: March, 2012))

Contents

List of Figures

List of Tables

Abstract

MATLAB is a popular dynamically-typed array-based language. 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 although `feval` provides an acceptable abstraction mechanism for these types of applications, 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 specializes calls of functions with `feval` using a combination of runtime input argument types and values. The second approach uses on-stack replacement technology, as supported by McVM/McOSR. Experimental results on seven numerical solvers show that the techniques provide good performance improvements.

1 Introduction

MATLAB is dynamic array-based language used by scientists and engineers 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`.

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 [1] and the McLAB group's open source VM/JIT, McVM [16, 4].

This paper focuses on determining if `feval` causes significant overheads in both the interpreter and JIT settings, and then proposes two mechanisms to optimize `feval`.

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 mechanisms. The first 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 second mechanism is more general, 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[13]. 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.

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

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.

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.

OSR-based specialization of `feval`: Not all `feval` calls fit the pattern handled by the JIT value-based specialization approach. Thus, we also developed a more general 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 the specialized call could be made or if the general backup path should be taken.

Implementation in McVM/McOSR: 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 provide key background to LLVM, McVM and McVM's OSR support. Section 4 provides the details of our first approach based on specializing on the values of key function parameters. Section 5 gives the key ideas of our second approach, `feval` optimization and how we implemented it in McVM/McOSR. 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 this section we provide some key background on MATLAB and its `feval` function, as well our experimental results which demonstrate the significant overheads of `feval`.

2.1 MATLAB and `feval`

In order to provide some intuition about MATLAB and the `feval` challenges, consider the example MATLAB function *newton* in Figure 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*. Lines 30 to 35 provide the definition of *fx3n*, which is one possible function that could be provided to *newton*.

```
function r = newton(fun,x0,xtol,ftol)

% newton    Newton's method to find a root of the scalar
%           equation f(x) = 0
% Synopsis:   r = newton(fun,x0,xtol,ftol)
% Input:    fun     = (string) name of mfile that
%                     returns f(x) and f'(x).
%           x0      = initial guess
%           xtol    = absolute tolerance on x.
%                     Smallest: xtol=5*eps
%           ftol    = absolute tolerance on f(x).
%                     Smallest: ftol=5*eps
% Output:   r       = the root of the function

  xeps = max(xtol,5*eps);
  feps = max(ftol,5*eps);  % Smallest tols are 5*eps
  x = x0;  k = 0;
  maxit = 15; % Initial guess, current and max iterations
  while k <= maxit
    k = k + 1;
    % Returns f( x(k-1) ) and f'(x(k-1))
    [f,dfdx] = feval(fun,x);
    dx = f/dfdx;
    x = x - dx;
    if ( abs(f) < feps ),  r = x;  return;  end
    if ( abs(dx) < xeps ),  r = x;  return;  end
  end
end

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
```

Figure 1: Newton's method to find a root of the scalar equation f(x) = 0, adapted from [19, 20]

The MATLAB function `feval` is a built-in function, that is used in MATLAB to indirectly evaluate a function at runtime. `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 Figure 1, a typical call would be one of the following:

```
newton(@fx3n, 3, 5e-16, 5e-16)
```

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

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

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.

We conducted our experiments on these systems over a set of MATLAB programs from numerical computing domain. 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 with a static count of the total number of `feval` calls in the program in Table I. The table also shows the number of `feval` calls in a loop in each benchmark.

BM	**Description**	**# `feval`**	**# LP `feval`**
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

Table I: `feval` benchmarks

We conducted all our experimental work on a computer with the following configuration.

Processor: AMD Athlon™ 64 X2 Dual Core Processor 3800+;
RAM: 4GB RAM;
Cache Memory: L1 128KB, L2 512KB;
Operating System: Ubuntu 11:04 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).

In Table II, for each benchmark, we show the execution time for the three systems: Octave, MATLAB and McVM. The column labelled *Interpreter* gives the execution times measured in seconds when the benchmarks were interpreted under the three systems. Similarly, the column labelled *JIT* gives the execution times, also measured in seconds, when the benchmarks were run under MATLAB and McVM. As we mentioned earlier, Octave does not have a JIT compiler. So, we recorded a ***** for each cell under the JIT category

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

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

	Interpreter					JIT				
	`feval` (F) t(s)	direct(D) t(s)	inlined(I) t(s)	D vs. F % impr	I vs. F % impr	`feval` (F) t(s)	direct(D) t(s)	inlined(I) t(s)	D vs. F % impr	I vs. F % impr
bisect										
Octave	29.31	27.70	19.46	5.50	33.59	*	*	*	*	*
MATLAB	13.67	12.45	7.09	8.91	48.12	6.94	6.24	0.74	10.07	89.31
McVM	13.81	13.69	9.76	0.90	29.36	10.26	7.89	4.63	23.06	54.90
newton										
Octave	29.94	27.90	19.17	6.81	35.97	*	*	*	*	*
MATLAB	15.17	13.79	8.38	9.08	44.71	8.17	7.61	1.60	6.85	80.35
McVM	21.91	21.53	13.94	1.74	36.37	12.45	7.56	2.92	39.28	76.54
odeEuler										
Octave	60.55	56.17	38.85	7.22	35.84	*	*	*	*	
MATLAB	29.45	26.16	15.30	11.17	48.06	7.30	5.96	4.47	18.29	38.74
McVM	31.04	29.01	19.43	6.51	37.40	18.36	3.37	3.23	81.63	82.43
odeMidpt										
Octave	103.35	94.44	55.60	8.63	46.20	*	*	*	*	*
MATLAB	46.87	41.83	19.16	10.75	59.12	8.34	7.32	4.49	12.22	46.22
McVM	48.37	47.98	29.14	0.80	39.76	24.86	2.72	2.66	89.05	89.29
odeRK4										
Octave	186.01	172.16	93.86	7.45	49.54	*	*	*	*	*
MATLAB	84.50	74.06	27.71	12.36	67.21	12.88	11.13	5.35	13.58	58.44
McVM	94.76	92.37	51.09	2.51	46.08	50.01	3.40	3.40	93.00	93.57
gaussQuad										
Octave	39.09	37.09	29.63	5.12	24.19	*	*	*	*	*
MATLAB	34.28	33.56	28.42	2.10	17.10	8.87	8.58	6.23	3.28	29.72
McVM	16.74	15.89	11.94	5.09	28.67	5.62	4.34	4.31	22.69	23.22
sim_anl										
Octave	39.74	38.54	35.39	3.01	10.93	*	*	*	*	*
MATLAB	43.50	43.09	40.51	0.94	6.87	9.58	9.50	9.07	0.83	5.35
McVM	18.15	16.79	14.39	7.49	20.70	15.50	12.12	8.87	21.80	42.79

Table II: `feval` overheads as compared to direct and inlined calls.

for Octave. Under the column labelled *Interpreter*, we show the result of running the original benchmarks (with `feval` calls) under *`feval` (F)*. The column labelled *direct(D)* shows the results of running a version of each benchmark with `feval` call replaced (by hand) with with a direct call to the input function used to run the benchmark. We show the result of running yet another version of each benchmark in which the input function has been hand-inlined under the column labelled *inlined(I)*. We show the results for the cases *`feval` (F)*, *direct(D)*, *inlined(I)* ran when the JIT compiler was enabled under the column *JIT*. Under both the *Interpreter* and *JIT* categories, we show the improvements in percentage (%) of *direct* and *inlined* over the `feval` under *D vs F* and *I vs F* respectively.

There are clear overheads for `feval`, when the `feval` is replaced by a direct call, improvements were from 1% to 12% in the interpreter settings and 7% to 50% in the JIT settings. Further improvements are enabled by inlining the direct calls, ranging from 6% to 67% for the interpreter and 5% to 89% for the JITs.

3 Background

In this section we provide key background and overview 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 with the implementation's code. 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.

The overall structure of McVM is given in Figure 2. It is 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[15, 14], 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.

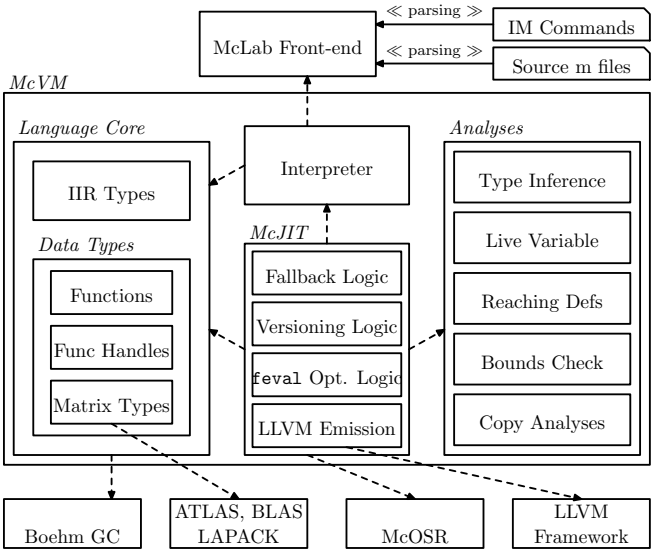

Figure 2: Overview of McVM

As illustrated in Figure 3, McJIT is a specializing JIT. When a function is called (top left of figure), the JIT first looks at the runtime types of the arguments and determines if target code matching those argument types is already available in the code cache, and if so executes it. Otherwise, the JIT looks to see if it already has the McJIT IR, and if not it requests the front-end to parse the matching source code (.m) file and it builds the IR. McJIT then performs its own analysis and transformations, then generates LLVM IR, specialized to the current argument types. LLVM then performs the final low-level optimizations and generates target code, which is stored in the Code Cache and executed.

3.2 `feval` in McVM

When McJIT encounters a MATLAB statement involving a call to `feval`, it generates LLVM code to call to a dynamic dispatcher. For example, when for the `feval` statement at line 22 of Figure 1, it generates the code in Figure 4. 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. And then generates the call to the dynamic function dispatcher, that is, the call to *Interpreter::callFunction* in line 6.

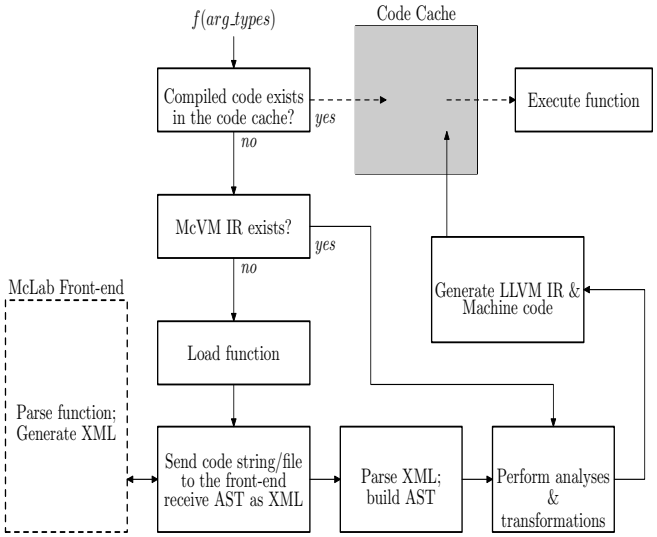

Figure 3: Running a function in McJIT.

```
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)
```

Figure 4: LLVM code generated for a `feval` call

When the dispatcher is called at runtime, it examines its first argument to determine that this 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 determines the right function to dispatch. It 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.

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 runtime cannot be determined statically in general, this implementation represents what is typically done to implement the `feval` library function.

A key point to note is that function binding and the argument types of the function called by `feval` often do not change through the whole loop execution, or even through the whole method execution, as is the case for the typical example in Figure 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.

3.3 OSR Background

McVM has support of OSR[13, 12] 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 Section 5 we provide the details of how we leverage the OSR machinery to optimize `feval`.

4 JIT Value-based Specialization

Our first approach to optimizing `feval` calls is based on the observation that, for some class of MATLAB programs, a function with an `feval` call often accepts as an argument the name or the function handle to a function evaluated by the `feval` call. Further, the call is often executed repeatedly within a long-running loop.

As in most implementations of the MATLAB language, the code generated for an `feval` call by our JIT compiler can be significantly less efficient.

An `feval` call often prevents compiler optimizations because its input function cannot, in general, be determined until the run time. In MATLAB, the value of the input function of an `feval` call — which we shall from now call *`feval` evaluated function* (*fef*) — can be formed dynamically (e.g., a string formed by a concatenation of some run-time values). The value can also come from a data structure (e.g., an array or a struct) or as a return value from a function call.

Our JIT-time code specialization for `feval` replaces calls to a function that has an `feval` call with a call to a special dispatch function. This dispatch function (called the dispatcher for short) evaluates the value of the parameter that corresponds to an *fef*. It then generates a new version of the function with all the `feval` calls replaced with direct calls to the *fef*. This is illustrated in Figure 5.

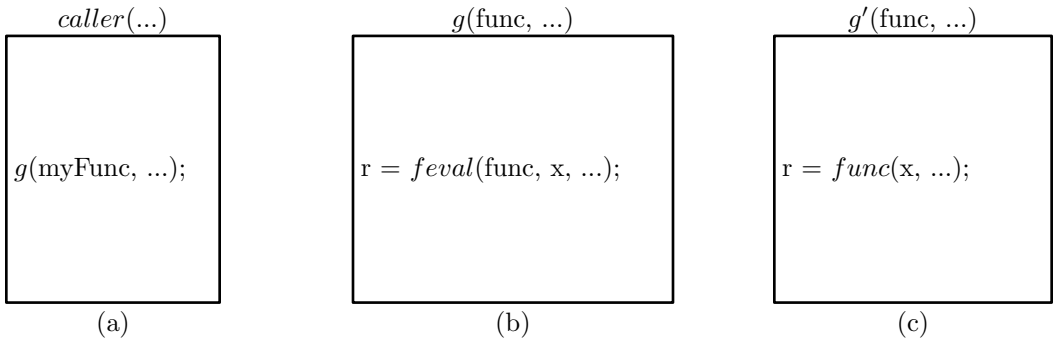

Figure 5: `feval` runtime code specialization.

In Figure 5, function *caller* calls function g. As shown in (b), function g has an `feval` call that evaluates one of its parameters, namely *func*. Function *caller* calls g with an argument, *myFunc*, which references a function (e.g., a function handle or a function name). This is the function that the `feval` call in g will evaluate.

However in Figure 5(c), a new version of function g named g' is created and all the `feval` calls that evaluate *func* have been replaced with direct calls to function *func*.

In the next section, we describe in detail the implementation of this approach, and in Section 5, we de-

scribe a more powerful approach that is capable of optimizing `feval` calls within a loop where the *fef* of an `feval` is not required to be a parameter of the `feval` call's enclosing function.

4.1 JIT Code Specialization for `feval`

During the parsing of the XML string for a compilation unit (i.e., a list of MATLAB functions in a MATLAB mfile (Figure 3)), McJIT analyzes all the functions in the compilation unit and annotates those with an `feval` call, whose *fef*, that is, the first parameter, is a read-only parameter of the enclosing function.

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) that corresponds to the called function at the call site;

(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.

$$descriptor = < entry\_address, argument\_types, counter, feval\_versions >$$

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*.

4.1.1 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 this call site. From the code cache descriptor, it compares the current value of the counter 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.

if, however, the counter is below the threshold, 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 the function at the address returned by the look-up. Otherwise, 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.

After the compilation of a new version, the dispatcher enters an entry, that is, a pair comprising of the AST corresponding to the compiled code and the entry point address of the compiled code, into a map in the code cache descriptor of the base IR so that if the function is called again with the *fef* value, the dispatcher can retrieve and execute the correct code. Finally, the dispatcher updates the counter associated with the cache slot descriptor and executes the function.

Even though the base AST and new versions of the AST have identical 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 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 corresponding to an *fef* via a wrapper. A wrapper is a short function that is composed of a call instruction and the instructions that convert the return values to their expected types.

A code cache miss causes a compilation of a new version. For this action, the counter associated with the code cache descriptor is incremented. The counter is reset to zero (0) at every code cache hit. As mentioned earlier, if the counter exceeds a given threshold, the dispatcher stops compiling a new version and always executes the original code generated for the base AST of the called function. This is useful in cases where multiple 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. In the next section, we discuss a more powerful approach that uses OSR to support an on-the-fly specialization of `feval` calls, and in Section 6.3, we compare the performance of this approach with the other approach, which we shall now describe.

5 OSR-based `feval` Transformation

In Section 2, we discussed the cost of `feval` in MATLAB programs and the challenges to an efficient implementation of `feval` in a MATLAB JIT compiler. We begin this section with a short discussion of the objectives for our approach to optimize `feval`, and then we highlight the major steps in our approach to on-the-fly specialization using OSR.

5.1 `feval` Optimization Goals and Strategy

In Figure 4 we illustrated the code currently generated for a call to `feval`. Line 6 contains the key problem, which is an indirect call to the interpreter *callFunction* method is required in order to dispatch to the correct function.

The aim of our approach is to replace the call to the dispatcher with a direct call to the function given as the first argument to the `feval` call while maintaining the correctness of the code. To maintain correctness we will need some safety checks that will backup to the general case if the current call does not match the last specialized version. Thus, another key challenge is minimizing the overhead for the check.

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 run-time (for example, when the while loop inside of *newton* executes).

Dispatcher call annotation: During JIT-compilation of a function body, all dispatcher calls that correspond to `feval` calls must be identified and marked. This is discussed in detail in Section 5.2.

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 5.3.

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 5.4.

5.2 Dispatcher call site annotation

As mentioned in the introduction to this section, we have added a pass to the McJIT compiler to identify all the calls to the dispatcher that correspond to an `feval` call. Theses call sites are annotated with the OSR ID of their closest enclosing loop. For example, for the `feval` call in Figure 6, 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 5.5.

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 5.4.

5.3 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 %osrTriggeringCond, label %OSRBB, label %BodyBB
```

This instruction inspects the OSR condition (*%osrTriggeringCond*) and branches to the basic block named %OSRBB (which triggers the OSR) if the test is successful. Otherwise, it branches to *%BodyBB* 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. Our code transformer has the following signature:

```
void transformFeval(llvm::Function* F, osr::OSRLabel L);
```

where F is the LLVM IR of the function that has triggered an OSR event, and L is the OSR label of the loop where an OSR has been triggered. We discuss in detail the logic of the code transformer in Section 5.4.

Figure 6 shows a code snippet from our running example, and in in Figure 7, 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 *LBB* or the loop exit block at *LE* depending on the loop exit condition (*%loopExitCond*).

The CFG shown in Figure 7 is transformed into that shown in Figure 8 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 (*%osrTriggeringCond*) and terminates with a conditional branch instruction as discussed earlier.

```
1     ...
2     while k ≤ maxit
3         k = k + 1;
4         [f, dfdx] = feval(fun,x);
5          ...
6     end
7  end
```

Figure 6: `while` loop extracted from (Figure 1)

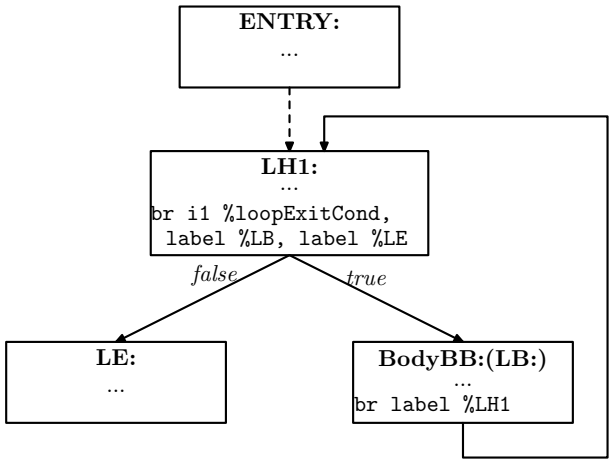

Figure 7: A CFG for the MATLAB `while` loop in Figure 6.

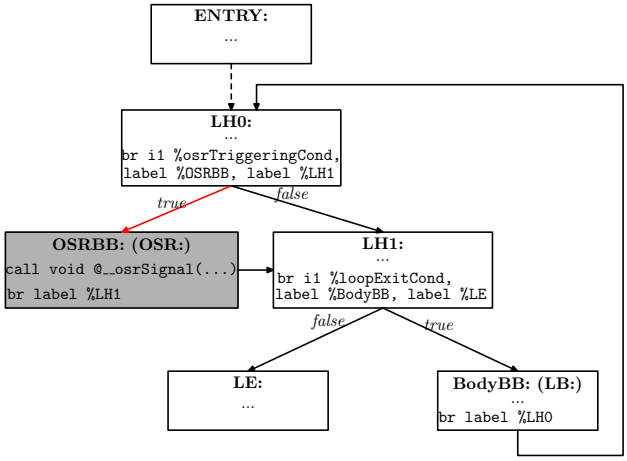

Figure 8: The CFG of a loop with an OSR point.

5.4 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.

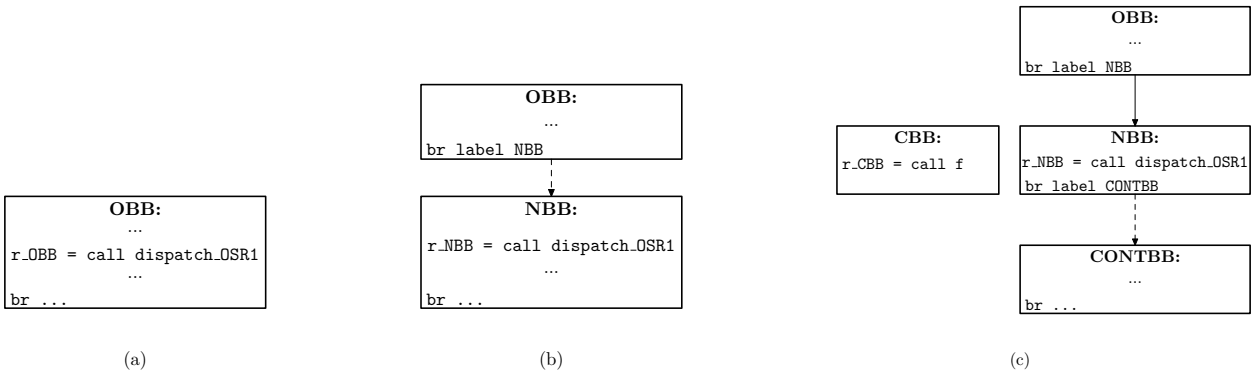

Figure 9: Actions of the code transformer. Basic block *OBB* in (a) is split into two. The result of the splitting process is shown in (b). In (c), *NBB* is split into *NBB* and *CONTBB*. A new unlinked basic block named *CBB* is also generated. *CBB* contains a call to the new compiled function (f).

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. The transformer can identify these call sites using the OSR label attached to such instructions at their creation time. The transformer also identifies 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 extracts the cache slot ID of the current call dispatcher. It then uses the cache slot ID as an index into the cache to retrieve the pointer to the array of objects containing the last arguments passed to the dispatcher. Using this pointer, the code transformer determines the function being dispatched — the *fef* — at this call site. However, if the cache slot is unset, the processing of the current call is aborted and the code transformer continues with the next call.

Having determined precisely the function passed to `feval` at this call site, the transformer begins a series of transformations at the basic block containing the current call. We illustrate the actions of the code transformer in Figure 9 and Figure 10.

Figure 9(a) shows a basic block (*OBB*) with a call to the dispatcher, represented with *dispatcher_OSR1*. As shown in the figure, the call to the dispatcher is annotated with OSR label *OSR1*.

The transformer first splits the original basic block (*OBB* in Figure 9(a)) to obtain the basic blocks shown in Figure 9(b). In Figure 9(b), the call to the dispatcher in *OBB* has been moved into the beginning of a new basic block named *NBB*.

Later, the transformer forms a string from the types determined for the last arguments passed to the dispatcher. This string forms a key into the code cache. Recall that McJIT caches code based on the types of the arguments passed to a function at a call site. The code transformer inspects the code cache using this key. If no matching compiled code is found, the code transformer calls the compiler to compile the function. 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 and performs further optimizations on the calling function as well.

After the compilation, the transformer creates a new basic block and creates the instructions to call the compiled function (f). This new block is shown in Figure 9(c) as *CBB*. To terminate *CBB*, the code transformer must first determine the continuation block. Of course, after the call to f in *CBB* returns, the execution must continue with the code after the call to the dispatcher in the original block (*OBB* in Figure 9(a)).

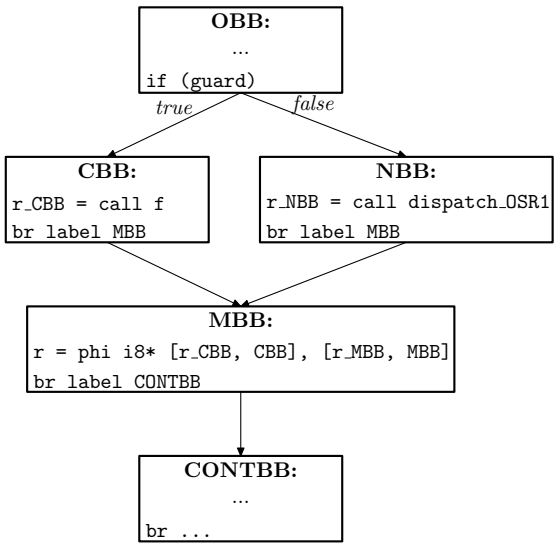

Figure 10: Actions of the code Transformer. Two new basic blocks have been inserted into the CFG: *CBB* contains a call to the compiled function (f), and *MBB* merges the results from the call in *CBB* and the original call to the dispatcher in *NBB*.

Thus, the code transformer splits *NBB* after the call to the dispatcher to obtain a new basic block *CONTBB*. This is the continuation block for *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*. Because the code in the current *OBB* (Figure 9(c)) is always executed before the call to the dispatcher in the original *OBB* (Figure 9(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 10, *MBB* merges the results from *CBB* and *NBB* via a *phi* instruction generated by the code transformer. *MBB* then terminates with a branch to the continuation block, *CONTBB* as shown in Figure 10.

The code transformer essentially implements our OSR-based `feval` optimization. The runtime performance is to a certain degree 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*.

5.5 Runtime guards

The code transformer generates a runtime guard (shown in Figure 10) 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 5.2, we mentioned that we collect a variety of JIT compilation-time facts on `feval` call sites in the *!FI* metadata. The second parameter of the metadata is an unsigned integer. It 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?
2. Is the first argument a loop constant variable?
3. Is there a possibility that any of the arguments to the `feval` call can have multiple types at run time?

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 [4], 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 more than one type exists in the type set for any argument, then the third query is true.

The combination of these queries guides the choice of the guards generated by the transformer. If query (1) is true, we can move the part of the computation of the guard (to determine whether or not the the run-time value of this argument corresponds to the function that will be called at *CBB* shown in Figure 10) to the function's entry block.

If query (2) 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 query (3) is false, it means that all the arguments are monomorphic and we can completely 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`: denote a function that returns the cached value of f;

`newValue`: denote a function that returns the current value of f;

`lastType`: denote a function that returns the cached type of a variable such as p_2, p_3, ..., p_n; and

`newType`: be a function that returns the current type of a variable.

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

Define

$$\begin{aligned} \textit{func\_cond} &= \texttt{lastValue}(f) == \texttt{newValue}(f) \\ \textit{arg\_cond} &= \forall (p \in P), \texttt{lastType}(p) == \texttt{newType}(p) \end{aligned}$$

#	Query(1)	Query(2)	Query(3)	Guard	Compute Point
1	true	true	true	*func_cond* $\wedge$ *arg_cond*	*func_cond*, entry block; *arg_cond* at *OBB*.
2	true	true	false	*func_cond*	*func_cond*, entry block.
3	true	false	true	*	*
4	true	false	false	*	*
5	false	true	true	*func_cond* $\wedge$ *arg_cond*	*func_cond* at loop entry block; *arg_cond* at *OBB*.
6	false	true	false	*func_cond*	*func_cond*, loop entry block.
7	false	false	true	*func_cond* $\wedge$ *arg_cond*	*func_cond* at *OBB*; *arg_cond* at *OBB*.
8	false	false	false	*func_cond*	*func_cond* at *OBB*.

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

Let us examine Table III. In the first case (i.e., table row 1), the results from the three queries are true, in this case, the required guard that the code transformer must generate is

guard = func_cond $\wedge$ *arg_cond*

this is because the type of each argument to f may change at run time. Furthermore, 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 *func_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 component of the guard, *arg_cond*, must be evaluated just before the use of the guard in basic block *OBB*.

In case 2 (i.e., table row 2), only *func_cond* should be computed and can be done at the calling function's entry basic block. Query(3) is false. Thus, we know that the runtime type of each argument at the `feval` call site is fixed so, there is no need to include the test *arg_cond* in the *guard* at *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.

Case 5 is similar to Case 1 except that Query(1) is false, meaning that f is not a read-only variable but it is a loop constant. For this reason, like Case 1, the required guard is
guard = *func_cond* $\wedge$ *arg_cond*.

However, unlike Case 1, the optimal point to compute *func_cond* is at the loop entry basic block (also referred to as the loop initialization basic block). The second component (*arg_cond*) must still be computed at *OBB*.

In Case 6, only *func_cond* should be computed and this can be done in the loop entry basic block.

Case 7 requires that both *func_cond* and *arg_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. And the types of the arguments may change at run time as indicated by the value of Query(3) in row 7 of Table III. Observe that this is the most expensive guard computation the code transformer can generate.

The last case is less expensive than Case 7 because in this case, 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 *func_cond* at *OBB* before the use of the guard.

The least expensive guard is in Case 2. This our ideal case. In the worst case (Case 7), 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.

5.6 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 the McOSR library[13].

6 Experimental Results

In Section 5, we presented a general OSR-based technique which allows a JIT compiler to generate better code on-the-fly for dispatching `feval` calls. Furthermore, in Section 4, we presented a relatively light-weight approach to optimizing `feval` calls. Here, we first discuss in detail the cost of `feval` calls in MATLAB programs. Then, we present the results of the experiments that we conducted to assess the effectiveness of our two specialization approaches for `feval` calls. Later, we compare the performance of the

two approaches on the benchmarks described in Section 2.

6.1 Cost of `feval`

In Section 2, we summarised the results of the experiment conducted to evaluate the cost of `feval` in MATLAB programs. Now, we discuss the results in detail. From Table II, we can observe that function evaluation via `feval` incur overheads. When we compared the `feval` and *direct*, we found performance improvements from 0.8% to 12.36%. This was consistent across the three systems.

When we compared `feval` and *inlined* versions under interpretation. Here we found a significant difference for each benchmark. In fact, we found performance improvement from 6.87% (*sim_anl*, with the MATLAB interpreter) to 67.21% (*odeRK4*, also with the MATLAB interpreter). When we computed a similar statistic for Octave alone, we found performance improvement from 10.93% (*sim_anl*) to 49.54% (*odeRK4*). And for McVM we found 20.71%(*sim_anl*) to 46.08% (*odeRK4*). The three different systems (Octave, MATLAB and McVM) gave the lowest improvement when running *sim_anl* and the highest improvement when running *odeRK4*.

We compared `feval` and *direct* under the JIT category. We found performance improvement from 0.83% (*sim_anl*, under the MATLAB JIT) to 92.99% (*odeRK4*, under the McVM JIT). The comparable statistic for `feval` against *inlined* was performance improvement from 5.34% (*odeRK4*, under the MATLAB JIT) to 93.57% (*odeRK4*, under the McVM JIT).

As shown in Table I, the *odeRK4* benchmark has four `feval` calls in its only loop. It is therefore not surprising that both the MATLAB JIT and McVM JIT recorded significant improvement when the `feval` calls were eliminated in the *inlined* version of the benchmark. Although the *sim_anl* benchmark has only one `feval` call in a long-running loop, the evaluated function was largely interpreted by our McVM JIT as it computes a complicated expression. The relatively low improvement recorded by both the MATLAB and McVM JITs under this benchmark may be due to the complicated nature of other computations performed by the benchmark. Although our `feval` optimization was still able to improve the performance of this benchmark. We return to this discussion in Section 6.2.

These statistics are interesting. They revealed to us that the direct and indirect cost of an `feval` call in a long-running loop can be significant. Thus, calling `feval` in a long-running loop presents an optimization opportunity, which we decided to explore further and develop an approach to optimizing an `feval` call in long-running MATLAB loops. We discuss the impact of our OSR-based `feval` optimization in the next section.

6.2 OSR-based `feval` optimization

In Table IV, the column labelled *Normal* shows the results of executing the benchmarks with McVM JIT in normal mode. Column *Opt-0* shows similar results when the benchmarks were run with our basic OSR-based `feval` optimization enabled. The column labelled *Opt-1* shows the result of running the benchmarks with the OSR-based `feval` optimization plus inlining optimization that is performed on a suitable compiled function by the code transformer. We show similar results when we raise the optimization level and include our symbol environment optimization on the larger scope enabled by inlining under column *Opt-2*. In the last part of the table, under the column labelled *Improvement*, we show the percentage improvement recorded at the three OSR-based `feval` optimization levels over McVM in the normal mode.

From our results, we found that our `feval` optimization was effective. Our `feval` optimization consistently outperformed the McVM JIT. We recorded the highest improvement of 50.45% running *newton* at the *Opt-2* optimization level (i.e., `feval` optimization plus inlining and interpreter-interaction simplifi-

	Normal	**`feval` Optimization**					
	t(s)	t(s)			Improvement (%)		
	Normal(N)	Opt-0	Opt-1	Opt-2	Opt-0 vs N	Opt-1 vs N	Opt-2 vs N
bisect	10.26	9.09	9.16	9.02	11.34	10.67	12.11
Newton	12.45	8.87	8.87	6.17	28.79	28.74	50.45
odeEuler	18.36	13.80	13.20	13.36	24.82	28.09	27.22
odeMidPoint	24.86	16.48	16.19	17.14	33.71	34.85	31.03
odeRK4	50.01	31.65	31.13	30.78	36.72	37.76	38.44
gaussQuad	5.62	4.02	4.29	4.15	28.46	23.67	26.16
sim_anl	15.50	12.59	12.85	10.87	18.80	17.07	29.85

Table IV: The McVM JIT vs the `feval` optimizing McVM JIT.

cation enabled by inlining). It does seem that the benchmark does not benefit from inlining alone as the result under *Opt-1* (i.e., `feval` optimization plus inlining) suggests. Only the *ode* benchmarks: *odeEuler*, *odeMidpt* and *odeRK4* show some improvements at *Opt-1*. The `feval` calls in these benchmarks evaluated the same function. And if the LLVM code of the inlined function is composed mainly of interactions with the interpreter, as it is the case with the *newton* benchmark, it may not lead to performance improvement because interpretation dominates. However, this class of code presents further optimization opportunity: after inlining, the interaction with the interpreter may be simplified. This can lead to a significant performance improvement. Our result for the *newton* benchmark supports this.

We recorded the lowest improvement of 10.67% with *bisect* running at the *Opt-1* optimization level. The `feval` calls in this benchmark evaluated the same function as the *newton* benchmark. Like *newton*, *bisect* performed better under *Opt-2*. Again, the simplification of the interpreter-JIT compiler interaction code benefits this benchmark and others as shown in the table.

The McVM JIT plus our OSR-based `feval` optimizing transformation outperformed the standard McVM JIT in all the benchmarks. Another important question is to see how it compares to the hand-coded direct versions and the hand-inlined versions. Our OSR-based version outperformed the hand-coded *direct* version shown Table II under the standard McVM JIT in three of the benchmarks: *newton*, *gaussQuad* and *sim_anl*. Further, it outperformed the standard McVM JIT even under the *inlined* version in the case of *gaussQuad*.

What then makes our optimization effective? Our results suggest that for our implementation, the inlining optimization is not enough. However, as our interaction simplification optimization shows, inlining is a big enabler of other optimizations. The interaction simplification code was particularly effective for most of the benchmarks.

Thus, converting an indirect call to a direct call can reveal good optimization opportunities that may be exploited for performance improvement. We conclude that our OSR-based `feval` optimizing transformation technique is effective and practical. We will continue to improve our optimizer and we believe that our technique can be used to improve performance in similar JIT compilers.

6.3 A Comparison of the OSR and JIT value-based-specialization approaches

In this section, we evaluate and compare the performance of our OSR-based approach against the JIT value-based-specialization approach.

In Figure 11, we show the execution times of the benchmarks under the two approaches. It is clear from the figure that the JIT value-based approach significantly outperforms the OSR-based approach in five of the seven benchmarks. On our benchmark set, the JIT value-based approach is about 1 – 8 times faster than

the OSR approach. The average speed up is 2.3. Why does the JIT value-based approach perform better?

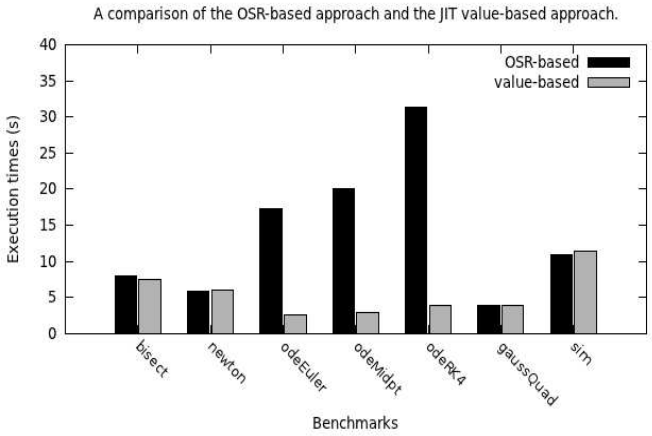

Figure 11: OSR vs JIT.

To understand why the JIT 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 the `feval` call to make them more generic (the return values are *unboxed* when the call returns). 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 approach.

In fact, runtime guards computation can be expensive. The OSR approach generates runtime guards, which, as discussed in Section 5.5, depend on whether or not the arguments to an `feval` call have a fixed type. In the three *ode* benchmarks, the type inference engine infers that the types to the first `feval` call in the three benchmarks are variable, forcing the code transformer to generate an expensive guard in this case. For the remaining `feval` calls (*odeMidpt* has 2; and *odeRK4* has 4 (Table I)), the type inference engine infers that the arguments have a fixed type and generates a much less expensive guard.

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 as a parameter, thus contributing to the generation of 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 look-up. But this is small given the expected gains.

We conclude that, although, the JIT value-based approach is less powerful than the OSR-based approach, it is much more effective on our benchmark set. The JIT approach only works where the *fef* is passed as a read-only parameter to a function. It does not work if the *fef* is a local variable in the function with the `feval` call. The OSR approach works in all cases but incurs much more 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.

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 [7, 11] uses a global cache to improve look up performance.

Our OSR-based approach is more related to the inline caching [5] approach used in another Smalltalk implementation. Interestingly, the Smalltalk 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 [5, 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) [8]. 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 [9, 18, 6, 2, 21, 13], 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 [3, 10].

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 runtime 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 runtime, 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.

To the best of our knowledge, we are not aware of any work on optimization technique for `feval` in a JIT compiler for MATLAB.

8 Conclusions and Future Work

MATLAB programmers often use `feval` to implement a wide variety of numeric solvers. `feval` provides a mechanism to pass function names or function handles as parameters. This use of `feval` is a very

reasonable way to implement general-purpose solvers, but in this paper we showed that `feval` incurs a significant performance overhead, both on interpreted systems and in existing JIT compilers.

We introduced an effective JIT value-based specialization technique for optimizing `feval` calls, whose first argument is a function parameter. We also proposed a more 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 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] G. Almási and D. Padua. MaJIC: Compiling MATLAB for Speed and Responsiveness. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 294–303, New York, USA, 2002. ACM.

[2] 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.

[3] C. Chambers and D. Ungar. Making Pure Object-Oriented Languages Practical. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 1–15, New York, USA, 1991. ACM.

[4] M. Chevalier-Boisvert, L. Hendren, and C. Verbrugge. Optimizing MATLAB through Just-In-Time Specialization. In *International Conference on Compiler Construction*, pages 46–65, March 2010.

[5] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM.

[6] S. J. Fink and F. Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-stack Replacement. In *Proceedings of the International Symposium on Code generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '03, pages 241–252, Washington, DC, USA, 2003. IEEE Computer Society.

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

[8] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 21–38, London, UK, UK, 1991. Springer-Verlag.

[9] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 32–43, New York, NY, USA, 1992. ACM.

[10] U. Hölzle and D. Ungar. A third-generation self implementation: Reconciling responsiveness with performance.

In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, OOPSLA '94, pages 229–243, New York, NY, USA, 1994. ACM.

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

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

[13] N. Lameed and L. Hendren. A modular approach to on-stack replacement in LLVM. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual Execution Enviornments*, VEE '13, pages 143–154, 2013.

[14] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–86, Washington, DC, USA, 2004. IEEE Computer Society.

[15] llvm.org. LLVM, 2012. `http://www.llvm.org/`.

[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 *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 1–12, Berkeley, CA, USA, 2001. USENIX Association.

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

[20] G. Rectenwald. 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.