Deoptimization for Dynamic Language JITs on Typed, Stack-based Virtual Machines

Madhukar N Kedlaya[†] Behnam Robatmili[⋆] Călin Caşcaval[⋆] Ben Hardekopf[†]

[†]University of California, Santa Barbara [⋆]Qualcomm Research Silicon Valley

{mkedlaya,benh}@cs.ucsb.edu mcjs@qti.qualcomm.com

Abstract

We are interested in implementing dynamic language runtimes on top of language-level virtual machines. Type specialization is a critical optimization for dynamic language runtimes: generic code that handles any type of data is replaced with specialized code for particular types observed during execution. However, types can change, and the runtime must recover whenever unexpected types are encountered. The state-of-the-art recovery mechanism is called *deoptimization*. Deoptimization is a well-known technique for dynamic language runtimes implemented in low-level languages like C. However, no dynamic language runtime implemented on top of a virtual machine such as the Common Language Runtime (CLR) or the Java Virtual Machine (JVM) uses deoptimization, because the implementation thereof used in low-level languages is not possible.

In this paper we propose a novel technique that enables deoptimization for dynamic language runtimes implemented on top of typed, stack-based virtual machines. Our technique does not require any changes to the underlying virtual machine. We implement our proposed technique in a JavaScript language implementation, MCJS, running on top of the Mono runtime (CLR). We evaluate our implementation against the current state-of-the-art recovery mechanism for virtual machine-based runtimes, as implemented both in MCJS and in IronJS. We show that deoptimization provides significant performance benefits, even for runtimes running on top of a virtual machine.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors–Run-time environments, Optimization

VEE '14, March 1–2, 2014, Salt Lake City, Utah, USA.
Copyright © 2014 ACM 978-1-4503-2764-0 /14/03…$15.00.
http://dx.doi.org/10.1145/2576195.2576209

Keywords deoptimization; JavaScript; type specialization; virtual machine; language implementation

1. Introduction

Language-level virtual machines (VMs) provide a number of advantages for application development. These advantages extend to implementing language runtimes on top of existing VMs, which we call *layered architectures*—for example, dynamic language runtimes like Rhino, IronJS, IronRuby, JRuby, IronPython, and Jython, which implement JavaScript, Ruby, and Python runtimes respectively, either on top of the Java Virtual Machine (JVM) or the Common Language Runtime (CLR).

However, VMs can also impose performance penalties that make language implementation unattractive. These penalties include not only VM overheads, but also *opportunity costs* arising from optimizations common to native runtime implementations[1] but difficult or impossible within a VM. Our goal in this work is to alleviate an important opportunity cost for implementing dynamic language runtimes on top of VMs. Specifically, we introduce a novel technique for *deoptimization* on typed, stack-based VMs that enables efficient type specialization, a critical optimization for dynamic language runtimes (explained further in Section 2).

Why Implement Languages on a VM? There are many advantages to using a layered architecture. Layered architectures provide nice program abstractions, free optimizations, and highly-tuned garbage collection, which are all required for a performant engine. Leveraging an existing VM allows the language developers to focus on language-specific optimizations without bothering with machine-specific optimizations that are handled by the existing VM. Layered architectures also offer a good platform for experimenting with new language features and different optimization techniques for language runtimes. Finally, using a layered architecture enables interoperability between different languages implemented on the same runtime.

[1] By which we mean runtimes implemented in a low-level language such as C and compiled to native binaries.

Opportunity Costs. The existing VM often imposes restrictions on the language developer that can prevent important optimizations. For example, a key optimization for dynamic languages is *type specialization*, which uses dynamic profiling to specialize code based on observed type information. Type specialization is unsound and thus requires a recovery mechanism to deal with unexpected types by transferring execution from the type-specialized code to the original unspecialized code. However, the very nature of typed, stack-based VMs such as the JVM or CLR means that the most effective known recovery mechanism, deoptimization, cannot be implemented using any known techniques that are used in native runtimes [12, 15, 20, 22].

Key Insights. We have developed a novel technique for effective deoptimization on typed, stack-based VMs. Our key insight is that we can leverage the VM's existing exception mechanism to perform the deoptimization. Doing so is non-trivial, because exceptions throw away the current runtime stack whereas deoptimization should preserve the stack information from the specialized code in order to re-start execution at the equivalent program point in the unspecialized code. Our technique leverages the code generator's bytecode verifier to track and transfer appropriate values on the runtime stack between the specialized code and the unspecialized code when a deoptimization exception is thrown.

Contributions. Our specific contributions are:

- We describe a novel deoptimization technique to enable type specialization for dynamic language runtimes running on top of a typed, stack-based virtual machine. (Section 3)
- We describe a specific instantiation of this technique for MCJS[2] [21], a JavaScript engine implemented on top of the CLR. (Section 4)
- We evaluate our MCJS implementation and compare against (1) a non-type specializing version of MCJS, (2) a type specializing version of MCJS using an alternate fast-path + slow-path recovery technique, and (3) IronJS[3], a JavaScript engine implemented using the DLR (which performs type specialization using the fast path + slow path technique). We use both standard benchmarks (i.e., Sunspider and V8) and long-running web JavaScript applications, and show that our deoptimization technique significantly outperforms existing type specialization techniques for layered architectures. On an average (geomean) our deoptimization technique is 1.16× and 1.88× faster than MCJS with fast-path + slow-path recovery technique and IronJS respectively. (Section 5)

Before describing our technique, we provide background on type specialization, the two dominant recovery mechanisms used by type specialization, and the challenges they face when implemented on top of a VM (Section 2).

[2] http://www.github.com/mcjs/mcjs.git

2. Type Specialization

In this section we give background on type specialization, the two dominant recovery mechanisms (*fast path* + *slow path* and *deoptimization*) used to implement type specialization, and the challenges faced by these techniques when implemented on top of VMs.

2.1 Type Specialization

Dynamic languages are dynamically typed, i.e., a variable can refer to values of different types at different points during a program's execution. However, dynamic language runtimes implemented in a typed language must declare a single type for each variable in the underlying implementation. Therefore, runtimes must wrap base values (e.g., integers, booleans, strings, etc) inside a wrapper type called a *DValue*, which stands for "dynamic value". Wrapping a base value inside a DValue is called *boxing*, and extracting a base value from a DValue is called *unboxing*.

The semantics of dynamic language operations depend heavily on the types involved. For example, the simple expression `a + b` can mean many different things depending on the types of `a` and `b` at the time the expression is evaluated. The runtime must unbox `a` and `b` to determine the types of the wrapped base values, perform the appropriate operation, and then box the result back into a DValue. Thus every expression encountered during execution requires unboxing values, performing a series of branch conditions based on type, performing the desired operation, and finally boxing a value. These operations tend to dominate the execution time of any dynamic language program.

In response, dynamic language implementors have developed an optimization called *type specialization*. During execution the observed types of each variable's values are monitored. The runtime then dynamically generates code that is specialized for the observed types. In the previous example, if `a` and `b` are always observed to hold integer values, then the runtime can generate specialized code that declares them to be `int` types instead of DValues and thus avoid all of the unboxing, branching, and boxing. However, this optimization is unsound—for example, while `a` and `b` have been integers so far, they may hold strings at some point later in the execution. Runtimes that use type specialization must have some sort of *recovery mechanism* that detects unexpected types and falls back to the standard, generic evaluation algorithm. There are two dominant approaches for this recovery mechanism; we describe each below along with their challenges with respect to being implemented on VMs.

2.2 Recovery Option 1: Fast Path + Slow Path

For the *fast path* + *slow path* recovery mechanism, type-specialized code is guarded by a conditional that tests the current types of the specialized variables. If the current types

```
if (GetType(a) == Int && GetType(b) == Int) {
  c = ToDValue(IntAdd(a.ToInt(), b.ToInt()));
}
else { // Slow path
  c = GenericAdd(a, b);
}
// c is of the type DValue here.
```

(a) C-like pseudocode representing the fast path + slow path approach for the statement `c = a + b`

```
if (GetType(a) == Int && GetType(b) == Int) {
  c = IntAdd(a.ToInt(), b.ToInt());
}
else {
  // Jump to deoptimization routine.
}
// c is of the type int here.
```

(b) C-like pseudocode representing the deoptimization approach for the statement `c = a + b`

Figure 1: Two approaches for generating type specialized code for the statement `c = a + b` where `a` and `b` are observed to be integers.

match the expected types then the true branch containing the type-specialized code is taken, otherwise the false branch containing the generic, unspecialized code is taken. In pseudocode, where `variable` is a DValue:

```
if (unbox(variable).type == type T) {
  T variable' = unbox(variable)
  // fast path: specialized code for type T
  // computes the result using variable'
  box(result)
}
else {
  // slow path: unspecialized code computes
  // the result using variable
}
// use result
```

Notice that variables are still unboxed and boxed for the fast path; this is because the type of `result` must be the same regardless of whether the fast path or slow path is taken. However, there may be multiple operations contained in the fast path and so the cost of boxing and unboxing is amortized; in addition, there is no branching on types in the fast path.

Figure 1a gives C-like pseudocode showing how the runtime implements the fast path + slow path operation for a simple binary add operation. Based on the previously observed types of `a` and `b`, say `int` and `int`, the runtime generates code to perform integer addition in the fast path and a generic add operation in the slow path.

Challenges. This technique is the one used in current layered architectures for dynamic languages that perform type specialization, such as IronJS and MCJS. There is no technical difficulty in implementing it, however the constant boxing and unboxing severely limits the benefits of type specialization. Deoptimization is known to out-perform fast path + slow path in native code implementation of dynamic language runtimes; however as we describe below deoptimization is difficult for VMs.

2.3 Recovery Option 2: Deoptimization

For the *deoptimization* recovery mechanism, type-specialized code is again guarded by a conditional that tests the current types of the specialized variables. The key difference is that the fast path and slow path are not contained inside the branches of the condition; instead, the slow path is placed in an entirely separate routine. If the condition fails then control leaves the current, type-specialized routine and jumps to the generic, unspecialized routine, where it resumes execution at the unspecialized program point that is equivalent to the specialized program point where the type mismatch was detected. In pseudocode, where `variable` is a DValue:

```
if (unbox(variable).type == expected type T) {
  T variable' = unbox(variable)
  // fast path: specialized code for type T
  // computes the result using variable'
}
else {
  // jump to equivalent program point in
  // unspecialized code
}
// use result
```

The benefit of this approach is that the remaining code in the routine can assume that the fast path succeeds, and hence we do not need to box the result—we can leave it as whatever type it was specialized to, because if it wasn't supposed to be that type then the code would have jumped completely out of the specialized routine and into the unspecialized routine.

Figure 1b gives C-like pseudocode describing the deoptimization approach for the statement `c = a + b`. Similar to the fast path + slow path approach, the guard condition checks whether the observed types of `a` and `b` are integers. If so, the runtime unboxes the integer values of `a` and `b` and performs the integer addition operation. This constitutes the fast path. A difference here with respect to the fast path + slow path approach is that resultant value is not boxed back into a DValue before assigning it to `c`. Instead, the type of `c` is initialized to be an integer. This prevents further unboxing of `c` when it is used later in the function. The deoptimization code captures the current state of execution of the code and transfers it to either an interpreter or to non-optimized compiled code.

Challenges. Deoptimization has been used in native code implementations of dynamic language runtimes. However, the techniques used there do not translate to typed, stack-based VMs such as the CLR or JVM. Native code uses either *code patching/on-stack replacement* or *long jumps*. In the former strategy, deoptimization is implemented by dynamically replacing the specialized code in the runtime stack with the generic unspecialized code. However, in managed VMs runtime modification of generated functions is not allowed. In the latter strategy, deoptimization is implemented as a long jump to the unspecialized code. However, in managed VMs long jumps are not allowed, for two reasons: first, it disables all optimizations that can be performed within a basic block, and second, these jumps can violate the *Gosling principle* which dictates that stack-based VMs should guarantee the *typestate* at any given program point. Typestate refers to the types of a function's local variables and the types of the values in the operand stack; stack-based VMs enforce the Gosling principle to help ensure correctness and performance. Thus, implementing the deoptimization strategy for type specialization using known techniques is not possible without modifying the underlying VM.

3. Deoptimization on Layered Architectures

In this section we give a high-level overview of our approach to solving the deoptimization problem on layered architectures. We discuss two aspects: (1) how to jump from the specialized code to the correct place in the unspecialized code; and (2) how to transfer the current state from the specialized code to the unspecialized code.

Jump to Unspecialized Code. When specialized code detects a type mismatch, it must jump from the current program point in the specialized code to the equivalent program point in the unspecialized code. As explained in Section 2, we cannot use the standard techniques of code patching or long jump to implement this behavior. Instead, we leverage the underlying VM's exception-handling mechanism. The jump from specialized code is done by throwing a `GuardFailure` exception. The body of every optimized method is wrapped in a try block, and deoptimization for every expression in that body is handled in a common catch block. Figure 3b illustrates the structure of the specialized code that is generated for a specific example.

The catch block must then transfer control to the unspecialized code, specifically the point equivalent to where the exception was thrown in the specialized version. To achieve this, we assume that the dynamic language runtime implements something like a subroutine-threaded interpreter [5]. A subroutine-threaded interpreter implements each operation of the program (e.g., reading a value of a variable, or performing binary addition) as a separate, unspecialized subroutine implemented in the underlying VM bytecode; each subroutine returns a pointer to the next subroutine that should be executed, and so interpretation consists of a series of subroutine calls with each call returning the address of the next subroutine to call.

Assuming the interpreter is subroutine-threaded, each language expression has an unspecialized implementation in the form of a subroutine with a known address. At each deoptimization guard, a pointer to the appropriate expression's subroutine is hardcoded into the thrown exception's value. The catch block then calls the appropriate subroutine to transfer control to the unspecialized code. We illustrate this process with an example in Section 4.

State Transfer. It is not sufficient to simply transfer control from the specialized code to the unspecialized code; we must also transfer the current state of the program, i.e., the values of the local variables on the runtime stack *and* the values on the operand stack used to store intermediate values during expression evaluation. Transferring the local variables is straightforward: we insert code immediately before the `GuardFailure` exception to read the values of each local variable and store them in a separate data structure shared by both specialized and unspecialized code. We describe such a data structure in Section 4.

The tricky part of state transfer is the operand stack. This stack is cleared whenever an exception is thrown, and its values are not stored in local or temporary variables. For example, suppose while evaluating the expression `a + b + c` that there is a deoptimization guard around `c` that throws an exception. The value of `a + b` resides (only) in the operand stack, and must be transferred to the unspecialized code that will evaluate `c` before the operand stack is cleared by the thrown exception. What makes this process tricky is that the number and types of values on the operand stack vary across deoptimization points; therefore we must have access to the stack size and type information at each deoptimization point in order to correctly transfer state. Unfortunately, managed VMs do not provide the ability to reflect on the operand stack during runtime.

We solve this problem by using compile-time[3] validation of the generated intermediate representation. To achieve this, the code generator is combined with a bytecode verifier which verifies the generated code line-by-line during code generation (as opposed to the normal order, which completely generates the code and then validates it). The benefit of this approach is that, in order to verify type safety, the code verifier maintains a shadow stack of value types present in the operand stack at any program point. The code generator can take advantage of this information during code generation, whereas it could not do this if the validator waited until after generation is complete.

[3] Throughout this paper, "compile" refers to generation of the typed bytecode of the underlying VM from the dynamic language being implemented on that VM. This should not be confused with the native code generation that happens at the VM level.

```
function foo(a)
{
  var b = 10;
  return a + b + global;
}
```

Figure 2: Running example in JavaScript.

This approach has two benefits beyond enabling correct state transfer. First, it enables runtime validation of the VM intermediate bytecode generated by the dynamic language runtime, which aids the language implementor in detecting compiler errors early rather than waiting until the code is actually run and the underlying VM gives an "Invalid IR" message. Secondly, there are certain unusual circumstances where the values on the operand stack cannot be transferred correctly to the unspecialized code, and hence deoptimization is not feasible (this is discussed further in Section 4.3). The code verifier will detect such circumstances and mark the code as un-optimizable.

4. Deoptimization for MCJS

This section concretely explains the algorithm for deoptimization that we have implemented in MCJS, a JavaScript engine implemented on top of the Common Language Runtime (CLR). MCJS performs type feedback based type inference to generate type specialized code. The type inference algorithm implemented in MCJS is described in the paper by Kedlaya et al [18]. The explanation in this section uses a running example given in Figure 2: a JavaScript function `foo` that takes an argument `a` which the example assumes is always an integer value.

The function `foo` is initially interpreted by the MCJS runtime. When `foo` becomes warm, it is compiled by the fast compiler into CIL[4] bytecode. This fast compilation also: (1) uses the code verifier to detect the types of values present on the operand stack for each potential deoptimization point, and determines for each point if deoptimization is feasible;[5] and (2) instruments the code to collect type profiling information. Finally, if `foo` becomes hot then it is re-compiled by the optimizing compiler into (1) a type-specialized CIL bytecode version based on the collected profile information; and (2) an unspecialized subroutine-threaded version used by the deoptimizer to recover from unexpected types.

The remaining subsections expand on the optimizing compiler pass: we explain first the subroutine-threaded code generator and then the specialized code generator that handles deoptimization.

4.1 Subroutine-Threaded Interpreter

When a hot function is compiled, the optimizing compiler first generates subroutine-threaded code for that function before generating type-specialized code. The order is important, because the specialized code needs to have pointers to the appropriate subroutines for each potential deoptimization point. Table 1 shows the subroutines that are generated for the example function in Figure 2. The only possible place for deoptimization (assuming `a` is always an integer) is if the type of `global` changes during some subsequent execution of `foo`. Thus, subroutine 5 is the subroutine that the runtime will jump to if deoptimization occurs. Since the subroutine-threaded interpreter executes a sequence of subroutines for each operation in the function, it is important to maintain an explicit stack that mimics the operand stack across the subroutines. MCJS implements this operand stack in the `callFrame` data structure described in Figure 4. The operand stack generated by subroutine 4 needs to be reconstructed by the deoptimizer before jumping into subroutine 5. The method to do so is explained below.

Index	Subroutine Name	Expression	Operand Stack
0	WriteIndentifier	b = 10	[]
1	ReadIdentifier	a	[a]
2	ReadIdentifier	b	[a, b]
3	AddExpression	a+b	[a+b]
4	ReadIdentifier	global	[a+b, global]
5	AddExpression	a+b+global	[a+b+global]
6	Return	return	[]

Table 1: Subroutines generated for a subroutine-threaded interpreter corresponding to the example in Figure 2. Subroutine 5 is the unspecialized code where control is transferred by the deoptimizer if `global` contains an unexpected type during the specialized code evaluation.

4.2 Specialized Code Generator

The generated type-specialized code contains deoptimization hooks at each potential deoptimization point. These hooks are filled in with the addresses of the appropriate subroutines generated as per the above description. In the example, the deoptimization code in the guard around `global` is compiled with a pointer to subroutine 5. Figure 3a shows the CIL code that is generated for the expression `a+b+global`.

It remains to explain how a deoptimization point transfers control to the unspecialized code subroutine while maintaining the current program state. We first explain how control is transferred from the specialized code into the unspecialized code, and then we explain how program state is transferred along with the control.

Control Transfer. The jump to the deoptimization code is implemented using the exception handling feature of the CLR. Each specialized method is wrapped in a try-catch block. Before a `GuardFailure` exception is thrown at a deoptimization point, the runtime updates the profiler with the new type that was observed, in order to improve the profiler's type information. The operand stack is then captured

[4] Common Intermediate Language, a typed bytecode IR used by the CLR.

[5] This is discussed further in Section 4.3.

```
...
0055   ldloc a
0056   ldloc b
0057   call Int32 Binary.Add:Run (Int32, Int32)

... ;  TYPE CHECK
... ;  Load the global variable
0071   dup
0072   call int DValue:get_ValueType()
007b   ldc.i4 9 ; 9 = observed type = Int32
0080   beq fast ; jump to fast path

... ;  DEOPTIMIZATION CODE
... ;  Update the profiler with observed type.
... ;  Transfer the operand stack to the
... ;  callFrame->stack data-structure.
... ;  Explained in Table 2.
00d4   ldc.i4 5 ; 5 is the index of the
                ; subroutine to jump into.
00db   throw GuardFailedException(Int32)

... ;  FAST PATH
fast   call Int32 DValue:AsInt32() ; Unboxing
00e5   call Int32 Binary.Add:Run(Int32, Int32)
... ;  Set the return value in the callFrame
00f4   ret

... ; CATCH BLOCK
... ; Store the current values of the local
... ; variables into the callFrame->symbols array.
... ; BlackList this function.
... ; Load the callFrame object that contains
... ; the updated stack and symbols.
... ; Load the subroutine index obtained from
... ; the exception value.
0147   ldc.i4 subroutineIndex
014c   call Void STInterp(Int32, CallFrame)
```

(a) CIL code generated by the type-specializing code generator for the expression `a + b + global`.

```
void __foo(CallFrame *callFrame)
{
 int a, b;

 try {
  b = 10;
  a = callFrame->argument[0].ToInt();

  int _temp0 = a + b;
  DValue _temp1 = callFrame->getGlobal("global");

  /* TYPE CHECK */
  if (_temp1.type != Int) { // Int is the profiled type
    /* DEOPTIMIZATION CODE */
    /* Update the profiler with newly observed type */
    UpdateProfiler(global, Int);
    /* Capture the current values of _temp* */
    callFrame->stack.Enqueue(_temp1); // Enqueue(DValue);
    callFrame->stack.Enqueue(_temp0); // Enqueue(int);
    /* 5 is the pointer to the subroutine */
    throw new GuardFailureException(5);
  }
  else { // FAST PATH
    callFrame->retVal = DValue(_temp0 + _temp1.AsInt32());
    return;
  }
 }
 catch (GuardFailureException e) {
  /* Update the callFrame->symbols array with the
     current values of local variables */
  callFrame->symbols[symbolsIndex++] = DValue(a);
  callFrame->symbols[symbolsIndex]   = DValue(b);
  BlackList(this); // BlackList this function code.
  STInterp(e.subRoutineIndex, callFrame);
 }
}
```

(b) C-like psuedocode that describes the generated CIL for the JavaScript code in Figure 2. The values pushed onto the stack are made explicit using _temp variables.

Figure 3: The code generated for the JavaScript example in Figure 2.

at the point the exception is thrown. The function locals, in contrast, are captured inside the catch block; this is because the operand stack is specific to a particular deoptimization point while the locals are common across all deoptimization points in the function. Capturing the values of the local variables in a single place avoids code duplication and reduces code bloat.

Once inside the catch block and with all local variables captured, the runtime must clean up and then transfer control to the appropriate subroutine. First, the runtime calls the `Blacklist` function which deletes the specialized code that had to be deoptimized and updates the function metadata with this information; this prevents the function from entering a cycle of specialization followed by deoptimization over and over again. Secondly, the runtime calls the appropriate subroutine whose pointer was passed inside the `GuardFailure` exception, passing it the updated `callFrame` data structure as explained below.

State Transfer. In MCJS, the `callFrame` data structure tracks the state of execution for the current function. It also holds a link to the scoping structure used to resolve the scope of the variables used in the function. Figure 4 shows the definition of `callFrame`. MCJS uses the `callFrame` object to transfer program state from the specialized code to the unspecialized subroutine-threaded code. The two relevant fields are `symbols`, which holds the values of the function's local variables at the deoptimization point, and `stack`, which holds the operand stack at the deoptimization point.

```
struct CallFrame {
  // Arguments passed to the function
  DValueArray arguments;
  // Return value of the function
  DValue retVal;

  // Fields and functions to track the scope
  // and other bookkeeping.
  Scope currentScope;
  Scope parentScope;

  // Fields below are only used by the
  // subroutine-threaded interpreter.
  // symbols array is used to store the values of
  // local variables at the deoptimization point.
  DValueArray symbols;

  // stack array is used to capture the state of
  // operand stack at the deoptimization point.
  DValueArray stack;
}
```

Figure 4: The `callFrame` data structure which tracks the state of execution for the current function.

The `symbols` field is computed inside the specialized function's catch block, as explained above. This is straightforward for the MCJS implementation because the runtime maintains a list of local symbols; the catch block merely iterates over this list and copies the values into the `callFrame.symbols` field.

The `stack` field must be computed separately for each deoptimization point. For each point, the type and number of values that need to be pushed onto the stack are different. The code generator used to generate the specialized CIL code uses the bytecode verifier to track this information. The verifier is reponsible for inferring and checking type information, which means that it already needs to know the required information. We simply piggyback on the verifier to determine what code to emit for enqueueing the operand stack values at each deoptimization point. The verifier maintains a data structure called the TypeStack which holds the types of values inside the operand stack at each program point. At each deoptimization point, we record the current TypeStack and emit code to enqueue the operand stack values onto `callFrame.stack`. Each value is wrapped inside a DValue before being enqueued. Because in CIL value types are not subtypes of the Object type, the runtime cannot use a generic `Enqueue(Object)` method to enqueue the values which is why we need the verifier's TypeStack information.

Table 2 shows how the state transfer code is generated for the example in Figure 2. Maintaining a TypeStack during code generation helps to determine which variation of `Enqueue` has to be called to enqueue the value in the top of the operand stack to `callFrame.stack`. In the example, while enqueuing `global` from the operand stack, the top of the TypeStack is referred for the appropriate type. Since the type of `global` is DValue, the CIL code to call `Enqueue(DValue)` is emitted by the code generator. Similarly, a call to `Enqueue(Int32)` is emitted to capture the value of `a + b` from the operand stack.

Instruction	Operand Stack	TypeStack
`;before state transfer`	`global` / `a + b`	`DValue` / `Int32`
`LdLoc callFrame`	`callFrame` / `global` / `a + b`	`DValue` / `Int32`
`LdFld stack`	`stack` / `global` / `a + b`	`DValue` / `Int32`
`Call stack.Enqueue(DValue)`	`a + b`	`Int32`
`LdLoc callFrame`	`callFrame` / `a + b`	`Int32`
`LdFld stack`	`stack` / `a + b`	`Int32`
`Call stack.Enqueue(Int32)`		

Table 2: The different steps taken when popping values from the operand stack.

4.3 Limitations

Our deoptimization technique assumes that all values present on the operand stack at a deoptimization point are subtypes of DValue. If so, then all of the values are easily convertable to value types used in the JavaScript runtime. However, there are rare cases where this assumption is not true. Some optimizations, such as polymorphic inline caches, store the `map` or `class` of an object in the operand stack of the CLR. If a deoptimization is triggered at this point, state transfer is not possible because `map` cannot be converted to a DValue and stored in `callFrame.stack`.

Fortunately, it is easy to detect this ahead of time during code generation. During the fast compilation phase which translates warm functions to CIL bytecode and instruments the code with type profiling hooks, the types of the values in the operand stack are tracked by the code verifier as previously described. For every deoptimization point, the type stack is checked to see whether it contains values that cannot be converted to DValue. If so, then the function is

Benchmark	Type
breakout.js	Game
chopper.js	Game, Animation
colorfulPointer.js	Utility, Animation
conways.js	Animation, Algorithm
flyingWindows.js	Animation, Utility
loadingSpinner.js	Utility
sierpinskiGasket.js	Algorithm
analogClock.js	Utility
halloweenAnim.js	Animation
growingGrass.js	Animation
kaboom.js	Game
mandelbrot.js	Animation, Algorithm
plasma.js	Animation
primesAnim,js	Algorithm
springPond.js	Algorithm, Animation
tetris.js	Game
waveGraph.js	Algorithm, Utility

Table 3: Table describing JS1k web applications used as benchmarks.

marked as non-optimizable. The profile hooks are removed and the function is compiled directly to CIL without any type feedback-based type specialization. Our evaluation shows that this circumstance rarely happens.

5. Evaluation

We evaluate our deoptimization technique on MCJS using the standard JavaScript benchmark suites Sunspider [1] and V8 [2] [6]. Because the Sunspider benchmarks run for a short duration of time (average of 180ms), each benchmark was wrapped in a 20× loop. We also evaluate our technique on real-world long-running web applications from the JS1k [4] website. Due to the unstable nature of IronJS, we selected only the benchmarks that IronJS was able to execute without any problem. The JS1k benchmarks are described in the Table 3.

Experimental Setup. We perform our experiments on a machine with two 6-core 1.9 GHz Intel Xeon processors with 32GB of RAM, running the Ubuntu 12.04.3 Linux OS and Mono v3.2.3. We used the latest version of IronJS, v0.2.1.0 from its Github repository [3].

Calculating Speedups. To calculate execution times, each of the benchmarks is run eleven times and the average execution time of the last ten executions is recorded.

Configurations. Speedup numbers were collected for the following five configurations.

- MCJS without type feedback-based type specialization (the base configuration against which results for other configurations are normalized).
- MCJS with type specialization using the standard fast path + slow path recovery mechanism (MCJS_FS).
- MCJS with type specialization using the deoptimization recovery mechanism, i.e., our technique (MCJS_D).
- MCJS with optimal type specialization (MCJS_OPT) as described below.
- IronJS in its default configuration.

[6] MCJS and IronJS do not implement typed arrays. Therefore, we not evaluate our implementation on Octane benchmarks.

The optimal type specialization configuration means that code is type-specialized but there is no deoptimization or any other recovery mechanism; this is unsound, but provides a maximal speedup due to type specialization against which we can compare our technique and the cost of deoptimization. IronJS is implemented on top of DLR [10] which mimics the fast path + slow path approach to optimizing type specializable code, hence we use it to show that MCJS is not a strawman JavaScript implementation.

5.1 Speedups

Figure 5 shows the speedups achieved by the type specializing configurations with respect to the MCJS base configuration for the Sunspider benchmark suite. The approaches without a local slow path (i.e., MCJS_D and MCJS_OPT) perform significantly better than the fast path + slow path approaches implemented in MCJS and IronJS. The MCJS_OPT configuration does not emit any deoptimization code and the runtime exits when any deoptimization should occur, which is why the 3d-cube.js benchmark sees a speedup of 0× for the MCJS_OPT configuration.

On an average (geomean) MCJS_D, MCJS_FS, and IronJS are 1.5×, 1.31×, and 0.77× faster than the base configuration, respectively. On comparing the execution times of MCJS_D against MCJS_FS and IronJS, we see an average speedup (geomean) of 1.14× and 1.97× respectively. An important observation is that for a few of the benchmarks like access-fannkuch.js, access-nbody.js, access-nsieve.js, bitops-bitwise-and.js, etc, the runtime does not benefit from type feedback-based type specialization. This is because these benchmarks are relatively small and execute for a very short period of time (average of 237.2ms). For these benchmarks profiling overhead is not amortized over time.

Figure 6 shows the speedups achieved by the type specializing configurations with respect to the MCJS base configuration for the V8 benchmark suite. We selected the benchmarks for which IronJS executed without crashing. Following a similar trend as the Sunspider benchmarks, MCJS_D, MCJS_FS, and IronJS are 2.13×, 1.74×, and 1.21× faster than the MCJS base configuration. On comparing the execution times of MCJS_D against MCJS_FS and IronJS, we see an average (geomean) speedup of 1.22× and 1.75× respectively. Excluding regexp.js (for which MCJS spends most of the time executing the inefficient regexp library code) and splay.js (which is a benchmark designed for stressing the garbage collection of the engine rather than the runtime

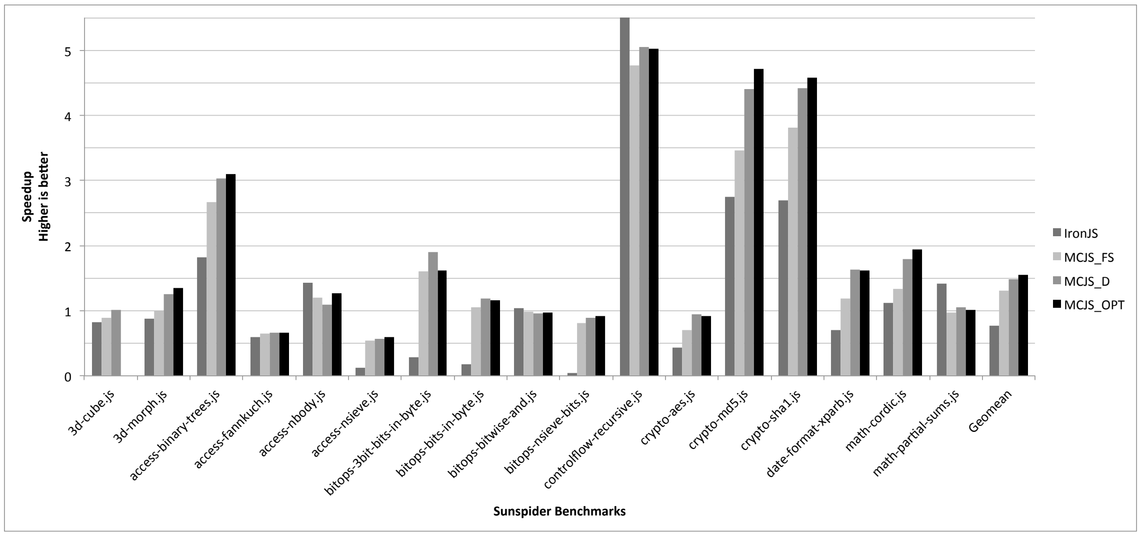

Figure 5: This figure shows the speedup numbers for various configurations of MCJS and IronJS for the Sunspider benchmark suite. FS = fast path + slow path, D = deoptimization, OPT = optimal.

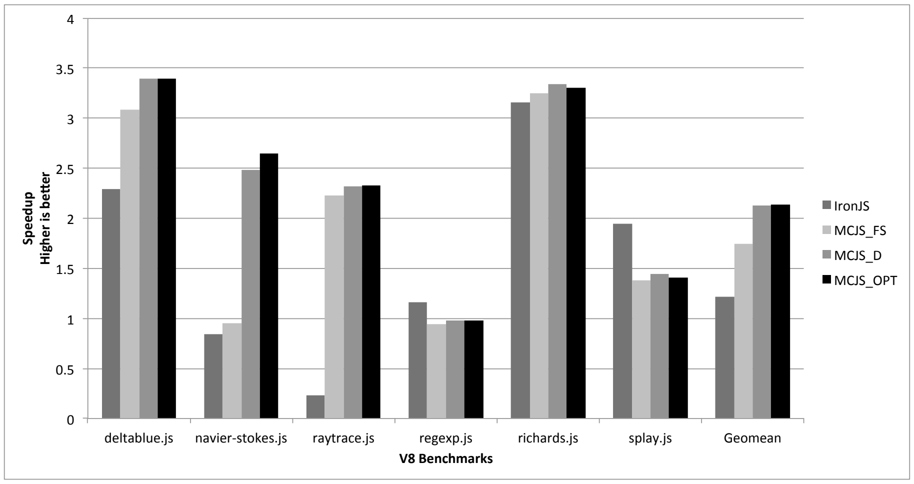

Figure 6: This figure shows the speedup numbers for various configurations of MCJS and IronJS for the V8 benchmark suite. FS = fast path + slow path, D = deoptimization, OPT = optimal.

performance), MCJS_D consistently performs better than all other configurations.

The JS1k benchmarks represent a diverse set of applications including games, utilities, algorithms, and animations. We manually modified the JavaScript code to eliminate or substitute code that interacted with the browser DOM. We substituted `setTimeOut` and `setInterval` functions with JavaScript functions that execute the passed-in function in a loop for a considerable number of times. For the benchmarks that require user interactions like mouse clicks, the user events were simulated using a fixed set of event objects embedded in the code. These applications run for a relatively long duration with the average execution time for the base configuration being 10.66 seconds.

Figure 7 shows the speedups achieved by the type specializing configurations with respect to the MCJS base configuration for the JS1k web application benchmark suite. As expected, MCJS_D, MCJS_FS and IronJS are 1.76×, 1.5×, and 0.94× times faster than the MCJS base configuration. Similar to the other benchmark suites, on comparing the execution times of MCJS_D with MCJS_FS and IronJS, we see an average speedup of 1.18× and 1.87× respectively.

Speedup vs. V8: MCJS achieves on an average about 75% of the V8 engine performance on the Sunspider benchmarks. The speedup is significantly lower for few of the

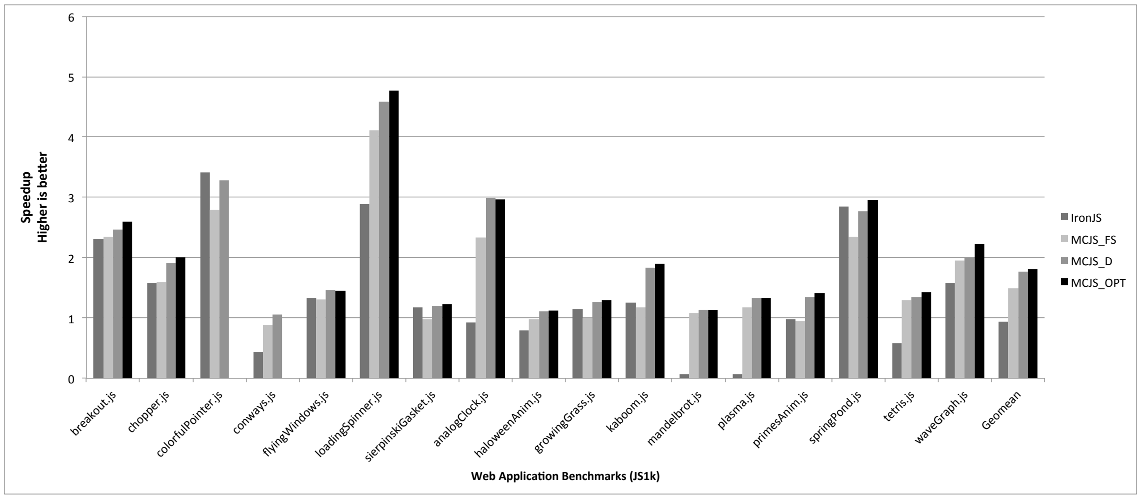

Figure 7: This figure shows the speedup numbers for various configurations of MCJS and IronJS for the web application benchmark suite. FS = fast path + slow path, D = deoptimization, OPT = optimal.

benchmarks in V8 benchmark suite. This is mainly because the regexp and string library implementations of MCJS (which are based on CLR's implementations) are very slow. Those affects dominate performance for those benchmarks, rather than anything due to recovery. However, this is not a fair comparison because V8 implements both the recovery mechanisms as part of its compilers along with many other optimizations, making it very difficult to tease out and isolate the effect of each of the recovery mechanisms.

5.2 Effect of Deoptimization

Deoptimization is a rare occurrence and it is observed only during the execution of the 3d-cube.js, colorfulPointer.js, and conways.js benchmarks. The speedup numbers for these benchmarks indicate that the overhead of the actual deoptimization process is negligible.

There are two ways of measuring the effect of the deoptimization code. First, we compare the speedup achieved by the MCJS_D and MCJS_OPT configurations. Figures 5, 6, and 7 indicate that the runtime overhead of the deoptimization code is negligible.

Secondly, we compare the size of the extra code that is generated to achieve deoptimization for each of the benchmarks. Figure 8 shows the comparison on code size of MCJS_D and MCJS_FS with respect to MCJS_OPT. Though the amount of code that is generated in MCJS_D is approx. 30% higher compared to the MCJS_OPT configuration, the impact on performance is negligible. This is because most of the extra code that is generated is to enable deoptimization. This deoptimization code is rarely ever executed for most of the benchmarks.

Another important metric used while comparing two implementations is the memory consumption. The amount of data captured in the stackFrame data-structure is very minimal; the operand stack and values associated with local variables are usually a few bytes in size. Therefore, the stackFrame data-structure has little to no impact on memory when we consider a managed runtime system.

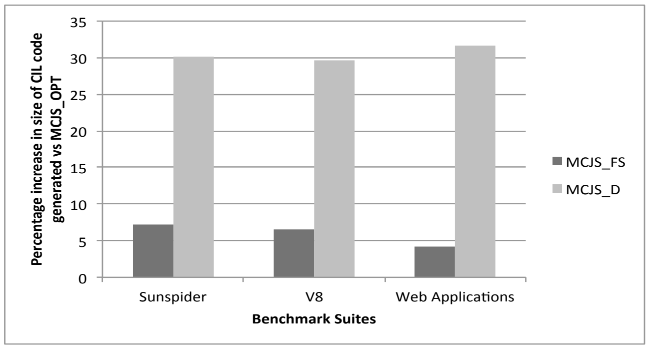

Figure 8: This figure shows the percentage increase in CIL code generated for MCJS_FS and MCJS_D in comparison to MCJS_OPT. FS = Fast + Slow Path, D = Fast Path with Deoptimization, and OPT = Fast Path with No Deoptimization.

5.3 Boxing/Unboxing

The amount of boxing and unboxing of DValues performed during the execution of the benchmarks is a major cause of overhead for the MCJS_FS configuration. Figure 9 shows the percentage increase in boxing and unboxing performed in MCJS_FS configuration when compared to the MCJS_D configuration for each of the benchmark suites. As expected, MCJS_FS performs more boxing and unboxing of values when compared to MCJS_D across all benchmark suites.

An important observation is that the percentage of boxing for web applications is significantly higher compared to other benchmarks. This is because the number of variables

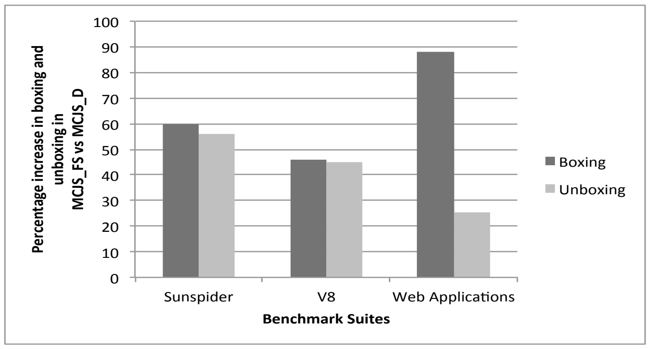

Figure 9: This figure shows the percentage increase in boxing and unboxing in MCJS_FS in comparison to MCJS_D. FS = fast path + slow path and D = deoptimization.

that are typed in the MCJS_D configuration is significantly higher compared to the number of local variables which are typed in the MCJS_FS configuration. This means that for the MCJS_FS configuration almost all the values need to be boxed before assigning them to the variable.

5.4 Non-optimizable Code

In some benchmarks, the deoptimization approach is not possible because some values present in the operand stack cannot be converted to DValues, as explained in Section 4.3. Among all benchmarks from the various benchmark suites that were executed, the runtime was not able to generate deoptimization code for only 35 out of 448 functions that were classified as hot. This shows that the deoptimization approach is viable for type specialization on top of VMs.

6. Related Work

Both the fast path + slow path and the deoptimization approaches for type specialization have been used in various dynamic language runtimes implemented natively (rather than on top of a VM). The baseline compilers of popular JavaScript engines V8 [9, 24] and SpiderMonkey [23] use the fast path + slow path approach for initial compilation of JavaScript functions to native code. Once a function becomes hot, the optimizing compilers for both of these engines generate type-specialized code with deoptimization hooks. If the types used for specialization change during execution, the runtime performs deoptimization by initiating long jumps to deoptimization routines in the compiled machine code. Language runtimes written on top of typed, stack-based runtimes cannot implement such a deoptimization technique because of the typed nature of the IR and the runtime type safety guarantees enforced by the VM.

TraceMonkey [13], PyPy [6], and LuaJIT [19] are popular tracing JIT compilers. Deoptimization is a common approach to use in runtimes with trace-based compilers. These traces span across function boundaries and are compiled to native code with deoptimization hooks. Implementing such a trace-based compiler on top of a VM is very complicated, especially from the perspective of deoptimization.

Brunthaler et al [7, 8] describe a purely interpretative optimization technique called *Quickening* implemented in CPython runtime. Quickening involves rewriting generic instructions to optimized alternatives based on the runtime information. This is analogous to the fast-path + slow-path approach of optimization. Quickening with deoptimization can be an alternative to the existing approach of optimization.

Hackett et al [14] describe an approach of combining type inference with type feedback to generate type specialized code. This approach uses recompilation approach instead of classic deoptimization technique to bail out whenever the type related assumptions do not hold anymore in the compiled code. Their approach tracks the type of values held by a variable or object field, and recompile all the type specialized code to generic version when the new types are observed.

Dynamic Language Runtime (DLR) [10] based language implementations like IronJS [3], IronRuby, and IronPython compile the program written in the dynamic language into DLR's *ExpressionTrees*. DLR performs the optimizations and native code generation required for the runtime. DLR employs polymorphic inline caches to specialize any operation observed during execution, which is analogous to the fast path + slow path approach of type specialization. As observed in Section 5, such an optimization does not always result in good performance when compared to the deoptimization approach.

Ishizaki et al [17] implement a dynamically-typed language runtime by modifying a statically-typed language compiler. Their approach to type specialization modifies the compiler to generate fast path + slow path code for arithmetic, logical, and comparison operators. Similarly to the MCJS original fast path + slow path approach, their approach also has to deal with incessant boxing and unboxing of values.

Duboscq et al [11] describe a way of inserting and coalescing deoptimization points in the IR of the Graal VM. This technique is orthogonal to and complementary to our own. In our approach, the deoptimization points are determined while generating the subroutine threaded code for the interpreter. Our implementation can benefit from the techniques like coalescing and movement of deoptimziation points described in their paper.

On-stack replacement (OSR) is a deoptimization / reoptimization strategy that has been explored and implemented in language runtimes to enable speculative optimizations [12, 15, 20, 22] and to enable debugging of optimized code [16]. Hölzle et al [16] implement deoptimization for the SELF programing language for debugging optimized code. The main focus of this work is to maintain the mapping from optimized compiled code to source code. As the authors have complete control over the underlying VM, such deoptimization is relatively easy to implement when compared to our implementation which does not modify the underlying VM. Fink et al [12] describe an on-stack replacement strategy for deoptimization implemented in JikesRVM. As described in the paper, capturing the state of the execution is straightforward given the access to the JVM scope descriptor object of the executing code. Our implementation is not straightforward due to the fact that part of the state that needs to be transferred resides in the underlying operand stack which is not easily accessible by any code currently executing in the VM. Soman et al [22] present a new general-purpose OSR technique on JikesRVM which is decoupled from the optimization performed by the runtime. Similar to this approach our deoptimization technique is also general-purpose. Applying the current deoptimization technique to other optimizations would involve minor modifications to the subroutine threaded interpreter to indicate the expected points of deoptimization specific to that optimization.

7. Conclusion

Deoptimization is a recovery mechanism which allows the runtime to bail out of type specialized code when type assumptions are violated, capture the state of current execution and continue execution form an equivalent point in a unspecialized code. This paper proposes a novel deoptimization based type-specialized code generation for a dynamic language runtime implemented on top of a typed, stack-based virtual machine. Our approach does not require any modification to the underlying virtual machine. Our implementation uses the exception handling feature offered by the underlying VM to perform deoptimization. Just using exception handling feature to jump into unspecialized code is not enough because throwing an exception clears the operand stack of the VM. The operand stack is an important part of state that needs to be transferred during deoptimization. We leverage the shadow type stack maintained by the bytecode verifier, which verifies the validity of the code generated during its generation, to safely transfer the values in the operand stack to the unspecialized code.

We implement our proposed technique in MCJS, a research JavaScript engine running on top of the Mono runtime. We evaluate our implementation against the fast path + slow path approach implemented in MCJS and IronJS. Our results show that deoptimization approach is on an average (geomean) $1.16\times$ and $1.88\times$ faster than fast path + slow path approach implemented in MCJS and IronJS respectively on Sunspider, V8 and web application benchmark suites.

Our implementation is generic and can be extended to enable other optimizations like function inlining. A few minor modifications to the existing approach are required to implement it in a sound manner. Currently, the location of deoptimization is determined by the placement of type checking guards. This needs to be extended to incorporate possible places where function inlining is possible in the code.

Acknowledgments

We would like to thank the the anonymous reviewers for their valuable input. The work described in this paper was significantly supported by Qualcomm Research.

References

[1] Sunspider benchmark suite. http://www.webkit.org/perf/sunspider/sunspider.html.

[2] V8 benchmark suite. http://v8.googlecode.com/svn/data/benchmarks/v7/README.txt.

[3] Ironjs javascript engine. https://github.com/fholm/IronJS, 2013.

[4] Js1k. http://js1k.com, 2013.

[5] M. Berndl, B. Vitale, M. Zaleski, and A. D. Brown. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *Proceedings of the international symposium on Code generation and optimization*, 2005.

[6] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, 2009.

[7] S. Brunthaler. Efficient interpretation using quickening. In *Proceedings of the 6th Symposium on Dynamic Languages*, 2010.

[8] S. Brunthaler. Inline caching meets quickening. In *Proceedings of the 24th European Conference on Object-oriented Programming*, 2010.

[9] Crankshaft compiler. V8 engine. http://www.jayconrod.com/posts/54/a-tour-of-v8-crankshaft-the-optimizing-compiler, 2013.

[10] DLR. Microsoft Dynamic Language Runtime. http://dlr.codeplex.com/, 2013.

[11] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the sixth ACM workshop on Virtual machines and intermediate languages*, 2013.

[12] 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*, 2003.

[13] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, 2009.

[14] B. Hackett and S.-y. Guo. Fast and precise hybrid type inference for javascript. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012.

[15] 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*, 1994.

[16] 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*, 1992.

[17] K. Ishizaki, T. Ogasawara, J. Castanos, P. Nagpurkar, D. Edelsohn, and T. Nakatani. Adding dynamically-typed language support to a statically-typed language compiler: performance evaluation, analysis, and tradeoffs. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2012.

[18] M. N. Kedlaya, J. Roesch, B. Robatmili, M. Reshadi, and B. Hardekopf. Improved type specialization for dynamic scripting languages. In *Proceedings of the 9th Symposium on Dynamic Languages*, 2013.

[19] LuaJIT. Lua Just-In-Time compiler. http://luajit.org/, 2013.

[20] 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*, 2001.

[21] B. Robatmili, C. Cascaval, M. Reshadi, M. N. Kedlaya, S. Fowler, M. Weber, and B. Hardekopf. Muscalietjs: Rethinking layered dynamic web runtimes. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2014.

[22] S. Soman and C. Krintz. Efficient and general on-stack replacement for aggressive program specialization. In *Proceedings of the 2006 International Conference on Programming Languages and Compilers*, 2006.

[23] SpiderMonkey. SpiderMonkey JavaScript Engine. http://www.mozilla.org/js/spidermonkey/, 2013.

[24] V8. Google Inc. V8 JavaScript virtual machine. https://code.google.com/p/v8, 2013.