Trace-based Just-in-Time Type Specialization for Dynamic Languages

Andreas Gal*+, Brendan Eich*, Mike Shaver*, David Anderson*, David Mandelin*,
Mohammad R. Haghighat$, Blake Kaplan*, Graydon Hoare*, Boris Zbarsky*, Jason Orendorff*,
Jesse Ruderman*, Edwin Smith#, Rick Reitmaier#, Michael Bebenita+, Mason Chang+#, Michael Franz+

Mozilla Corporation*
{gal,brendan,shaver,danderson,dmandelin,mrbkap,graydon,bz,jorendorff,jruderman}@mozilla.com

Adobe Corporation#
{edwsmith,rreitmai}@adobe.com

Intel Corporation$
{mohammad.r.haghighat}@intel.com

University of California, Irvine+
{mbebenit,changm,franz}@uci.edu

Abstract

Dynamic languages such as JavaScript are more difficult to compile than statically typed ones. Since no concrete type information is available, traditional compilers need to emit generic code that can handle all possible type combinations at runtime. We present an alternative compilation technique for dynamically-typed languages that identifies frequently executed loop traces at run-time and then generates machine code on the fly that is specialized for the actual dynamic types occurring on each path through the loop. Our method provides cheap inter-procedural type specialization, and an elegant and efficient way of incrementally compiling lazily discovered alternative paths through nested loops. We have implemented a dynamic compiler for JavaScript based on our technique and we have measured speedups of 10x and more for certain benchmark programs.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors — *Incremental compilers, code generation*.

General Terms Design, Experimentation, Measurement, Performance.

Keywords JavaScript, just-in-time compilation, trace trees.

1. Introduction

Dynamic languages such as JavaScript, Python, and Ruby, are popular since they are expressive, accessible to non-experts, and make deployment as easy as distributing a source file. They are used for small scripts as well as for complex applications. JavaScript, for example, is the de facto standard for client-side web programming and is used for the application logic of browser-based productivity applications such as Google Mail, Google Docs and Zimbra Collaboration Suite. In this domain, in order to provide a fluid user experience and enable a new generation of applications, virtual machines must provide a low startup time and high performance.

Compilers for statically typed languages rely on type information to generate efficient machine code. In a dynamically typed programming language such as JavaScript, the types of expressions may vary at runtime. This means that the compiler can no longer easily transform operations into machine instructions that operate on one specific type. Without exact type information, the compiler must emit slower generalized machine code that can deal with all potential type combinations. While compile-time static type inference might be able to gather type information to generate optimized machine code, traditional static analysis is very expensive and hence not well suited for the highly interactive environment of a web browser.

We present a trace-based compilation technique for dynamic languages that reconciles speed of compilation with excellent performance of the generated machine code. Our system uses a mixed-mode execution approach: the system starts running JavaScript in a fast-starting bytecode interpreter. As the program runs, the system identifies *hot* (frequently executed) bytecode sequences, records them, and compiles them to fast native code. We call such a sequence of instructions a *trace*.

Unlike method-based dynamic compilers, our dynamic compiler operates at the granularity of individual loops. This design choice is based on the expectation that programs spend most of their time in hot loops. Even in dynamically typed languages, we expect hot loops to be mostly *type-stable*, meaning that the types of values are invariant. (12) For example, we would expect loop counters that start as integers to remain integers for all iterations. When both of these expectations hold, a trace-based compiler can cover the program execution with a small number of type-specialized, efficiently compiled traces.

Each compiled trace covers one path through the program with one mapping of values to types. When the VM executes a compiled trace, it cannot guarantee that the same path will be followed or that the same types will occur in subsequent loop iterations.

PLDI'09, June 15–20, 2009, Dublin, Ireland.

Hence, recording and compiling a trace *speculates* that the path and typing will be exactly as they were during recording for subsequent iterations of the loop.

Every compiled trace contains all the *guards* (checks) required to validate the speculation. If one of the guards fails (if control flow is different, or a value of a different type is generated), the trace exits. If an exit becomes hot, the VM can record a *branch trace* starting at the exit to cover the new path. In this way, the VM records a *trace tree* covering all the hot paths through the loop.

Nested loops can be difficult to optimize for tracing VMs. In a naïve implementation, inner loops would become hot first, and the VM would start tracing there. When the inner loop exits, the VM would detect that a different branch was taken. The VM would try to record a branch trace, and find that the trace reaches not the inner loop header, but the outer loop header. At this point, the VM could continue tracing until it reaches the inner loop header again, thus tracing the outer loop inside a trace tree for the inner loop. But this requires tracing a copy of the outer loop for every side exit and type combination in the inner loop. In essence, this is a form of unintended tail duplication, which can easily overflow the code cache. Alternatively, the VM could simply stop tracing, and give up on ever tracing outer loops.

We solve the nested loop problem by recording *nested trace trees*. Our system traces the inner loop exactly as the naïve version. The system stops extending the inner tree when it reaches an outer loop, but then it starts a new trace at the outer loop header. When the outer loop reaches the inner loop header, the system tries to call the trace tree for the inner loop. If the call succeeds, the VM records the call to the inner tree as part of the outer trace and finishes the outer trace as normal. In this way, our system can trace any number of loops nested to any depth without causing excessive tail duplication.

These techniques allow a VM to dynamically translate a program to nested, type-specialized trace trees. Because traces can cross function call boundaries, our techniques also achieve the effects of inlining. Because traces have no internal control-flow joins, they can be optimized in linear time by a simple compiler (10). Thus, our tracing VM efficiently performs the same kind of optimizations that would require interprocedural analysis in a static optimization setting. This makes tracing an attractive and effective tool to type specialize even complex function call-rich code.

We implemented these techniques for an existing JavaScript interpreter, SpiderMonkey. We call the resulting tracing VM *TraceMonkey*. TraceMonkey supports all the JavaScript features of SpiderMonkey, with a 2x-20x speedup for traceable programs.

This paper makes the following contributions:

- We explain an algorithm for dynamically forming trace trees to cover a program, representing nested loops as nested trace trees.
- We explain how to speculatively generate efficient type-specialized code for traces from dynamic language programs.
- We validate our tracing techniques in an implementation based on the SpiderMonkey JavaScript interpreter, achieving 2x-20x speedups on many programs.

The remainder of this paper is organized as follows. Section 3 is a general overview of trace tree based compilation we use to capture and compile frequently executed code regions. In Section 4 we describe our approach of covering nested loops using a number of individual trace trees. In Section 5 we describe our trace-compilation based speculative type specialization approach we use to generate efficient machine code from recorded bytecode traces. Our implementation of a dynamic type-specializing compiler for JavaScript is described in Section 6. Related work is discussed in Section 8. In Section 7 we evaluate our dynamic compiler based on a set of industry benchmarks. The paper ends with conclusions in Section 9 and an outlook on future work is presented in Section 10.

```
for (var i = 2; i < 100; ++i) {
  if (!primes[i])
    continue;
  for (var k = i + i; i < 100; k += i)
    primes[k] = false;
}
```

Figure 1. Sample program: sieve of Eratosthenes. `primes` is initialized to an array of 100 `false` values on entry to this code snippet.

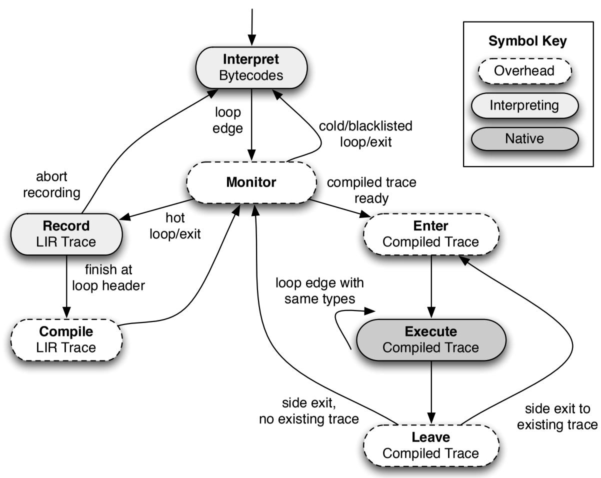

Figure 2. State machine describing the major activities of TraceMonkey and the conditions that cause transitions to a new activity. In the dark box, TM executes JS as compiled traces. In the light gray boxes, TM executes JS in the standard interpreter. White boxes are overhead. Thus, to maximize performance, we need to maximize time spent in the darkest box and minimize time spent in the white boxes. The best case is a loop where the types at the loop edge are the same as the types on entry–then TM can stay in native code until the loop is done.

2. Overview: Example Tracing Run

This section provides an overview of our system by describing how TraceMonkey executes an example program. The example program, shown in Figure 1, computes the first 100 prime numbers with nested loops. The narrative should be read along with Figure 2, which describes the activities TraceMonkey performs and when it transitions between the loops.

TraceMonkey always begins executing a program in the bytecode interpreter. Every loop back edge is a potential trace point. When the interpreter crosses a loop edge, TraceMonkey invokes the *trace monitor*, which may decide to record or execute a native trace. At the start of execution, there are no compiled traces yet, so the trace monitor counts the number of times each loop back edge is executed until a loop becomes *hot*, currently after 2 crossings. Note that the way our loops are compiled, the loop edge is crossed before entering the loop, so the second crossing occurs immediately after the first iteration.

Here is the sequence of events broken down by outer loop iteration:

```
v0 := ld state[748]     // load primes from the trace activation record
      st sp[0], v0      // store primes to interpreter stack
v1 := ld state[764]     // load k from the trace activation record
v2 := i2f(v1)           // convert k from int to double
      st sp[8], v1      // store k to interpreter stack
      st sp[16], 0      // store false to interpreter stack
v3 := ld v0[4]          // load class word for primes
v4 := and v3, -4        // mask out object class tag for primes
v5 := eq v4, Array      // test whether primes is an array
      xf v5             // side exit if v5 is false
v6 := js_Array_set(v0, v2, false)  // call function to set array element
v7 := eq v6, 0          // test return value from call
      xt v7             // side exit if js_Array_set returns false.
```

Figure 3. LIR snippet for sample program. This is the LIR recorded for line 5 of the sample program in Figure 1. The LIR encodes the semantics in SSA form using temporary variables. The LIR also encodes all the stores that the interpreter would do to its data stack. Sometimes these stores can be optimized away as the stack locations are live only on exits to the interpreter. Finally, the LIR records guards and side exits to verify the assumptions made in this recording: that `primes` is an array and that the call to set its element succeeds.

```
mov edx, ebx(748)       // load primes from the trace activation record
mov edi(0), edx         // (*) store primes to interpreter stack
mov esi, ebx(764)       // load k from the trace activation record
mov edi(8), esi         // (*) store k to interpreter stack
mov edi(16), 0          // (*) store false to interpreter stack
mov eax, edx(4)         // (*) load object class word for primes
and eax, -4             // (*) mask out object class tag for primes
cmp eax, Array          // (*) test whether primes is an array
jne side_exit_1         // (*) side exit if primes is not an array
sub esp, 8              // bump stack for call alignment convention
push false              // push last argument for call
push esi                // push first argument for call
call js_Array_set       // call function to set array element
add esp, 8              // clean up extra stack space
mov ecx, ebx            // (*) created by register allocator
test eax, eax           // (*) test return value of js_Array_set
je side_exit_2          // (*) side exit if call failed
...
side_exit_1:
mov ecx, ebp(-4)        // restore ecx
mov esp, ebp            // restore esp
jmp epilog              // jump to ret statement
```

Figure 4. x86 snippet for sample program. This is the x86 code compiled from the LIR snippet in Figure 3. Most LIR instructions compile to a single x86 instruction. Instructions marked with `(*)` would be omitted by an idealized compiler that knew that none of the side exits would ever be taken. The 17 instructions generated by the compiler compare favorably with the 100+ instructions that the interpreter would execute for the same code snippet, including 4 indirect jumps.

i=2. This is the first iteration of the outer loop. The loop on lines 4-5 becomes hot on its second iteration, so TraceMonkey enters recording mode on line 4. In recording mode, TraceMonkey records the code along the trace in a low-level compiler intermediate representation we call *LIR*. The LIR trace encodes all the operations performed and the types of all operands. The LIR trace also encodes *guards*, which are checks that verify that the control flow and types are identical to those observed during trace recording. Thus, on later executions, if and only if all guards are passed, the trace has the required program semantics.

TraceMonkey stops recording when execution returns to the loop header or exits the loop. In this case, execution returns to the loop header on line 4.

After recording is finished, TraceMonkey compiles the trace to native code using the recorded type information for optimization. The result is a native code fragment that can be entered if the interpreter PC and the types of values match those observed when trace recording was started. The first trace in our example, T_{45}, covers lines 4 and 5. This trace can be entered if the PC is at line 4, `i` and `k` are integers, and `primes` is an object. After compiling T_{45}, TraceMonkey returns to the interpreter and loops back to line 1.

i=3. Now the loop header at line 1 has become hot, so TraceMonkey starts recording. When recording reaches line 4, TraceMonkey observes that it has reached an inner loop header that already has a compiled trace, so TraceMonkey attempts to nest the inner loop inside the current trace. The first step is to call the inner trace as a subroutine. This executes the loop on line 4 to completion and then returns to the recorder. TraceMonkey verifies that the call was successful and then records the call to the inner trace as part of the current trace. Recording continues until execution reaches line 1, and at which point TraceMonkey finishes and compiles a trace for the outer loop, T_{16}.