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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

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

Copyright © 2009 ACM 978-1-60558-392-1/09/06. . . $5.00

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.

```
1 for (var i = 2; i < 100; ++i) {
2   if (!primes[i])
3     continue;
4   for (var k = i + i; i < 100; k += i)
5     primes[k] = false;
6 }
```

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

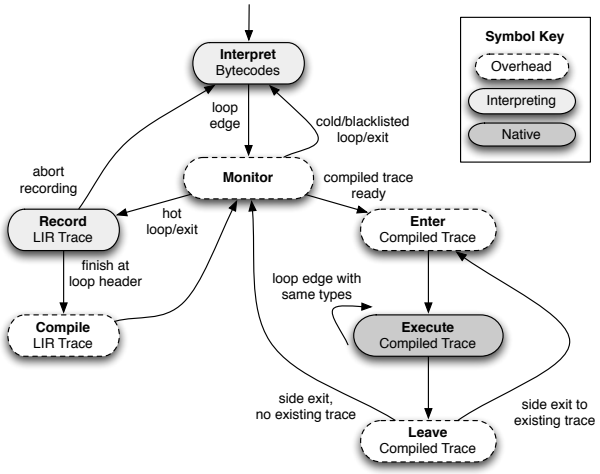


Figure 2. State machine describing the major activities of TraceMonkey and the conditions that cause transitions to a new activity. In the dark box, TM executes JS as compiled traces. In the light gray boxes, TM executes JS in the standard interpreter. White boxes are overhead. Thus, to maximize performance, 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}.

i=4. On this iteration, TraceMonkey calls T_{16}. Because `i=4`, the `if` statement on line 2 is taken. This branch was not taken in the original trace, so this causes T_{16} to fail a guard and take a side exit. The exit is not yet hot, so TraceMonkey returns to the interpreter, which executes the continue statement.

i=5. TraceMonkey calls T_{16}, which in turn calls the nested trace T_{45}. T_{16} loops back to its own header, starting the next iteration without ever returning to the monitor.

i=6. On this iteration, the side exit on line 2 is taken again. This time, the side exit becomes hot, so a trace $T_{23,1}$ is recorded that covers line 3 and returns to the loop header. Thus, the end of $T_{23,1}$ jumps directly to the start of T_{16}. The side exit is patched so that on future iterations, it jumps directly to $T_{23,1}$.

At this point, TraceMonkey has compiled enough traces to cover the entire nested loop structure, so the rest of the program runs entirely as native code.

3. Trace Trees

In this section, we describe traces, trace trees, and how they are formed at run time. Although our techniques apply to any dynamic language interpreter, we will describe them assuming a bytecode interpreter to keep the exposition simple.

3.1 Traces

A *trace* is simply a program path, which may cross function call boundaries. TraceMonkey focuses on *loop traces*, that originate at a loop edge and represent a single iteration through the associated loop.

Similar to an extended basic block, a trace is only entered at the top, but may have many exits. In contrast to an extended basic block, a trace can contain join nodes. Since a trace always only follows one single path through the original program, however, join nodes are not recognizable as such in a trace and have a single predecessor node like regular nodes.

A *typed trace* is a trace annotated with a type for every variable (including temporaries) on the trace. A typed trace also has an entry *type map* giving the required types for variables used on the trace before they are defined. For example, a trace could have a type map `(x: int, b: boolean)`, meaning that the trace may be entered only if the value of the variable `x` is of type `int` and the value of `b` is of type `boolean`. The entry type map is much like the signature of a function.

In this paper, we only discuss typed loop traces, and we will refer to them simply as "traces". The key property of typed loop traces is that they can be compiled to efficient machine code using the same techniques used for typed languages.

In TraceMonkey, traces are recorded in trace-flavored SSA *LIR* (low-level intermediate representation). In trace-flavored SSA (or TSSA), phi nodes appear only at the entry point, which is reached both on entry and via loop edges. The important LIR primitives are constant values, memory loads and stores (by address and offset), integer operators, floating-point operators, function calls, and conditional exits. Type conversions, such as integer to double, are represented by function calls. This makes the LIR used by TraceMonkey independent of the concrete type system and type conversion rules of the source language. The LIR operations are generic enough that the backend compiler is language independent. Figure 3 shows an example LIR trace.

Bytecode interpreters typically represent values in a various complex data structures (e.g., hash tables) in a boxed format (i.e., with attached type tag bits). Since a trace is intended to represent efficient code that eliminates all that complexity, our traces operate on unboxed values in simple variables and arrays as much as possible.

A trace records all its intermediate values in a small activation record area. To make variable accesses fast on trace, the trace also imports local and global variables by unboxing them and copying them to its activation record. Thus, the trace can read and write these variables with simple loads and stores from a native activation recording, independently of the boxing mechanism used by the interpreter. When the trace exits, the VM boxes the values from this native storage location and copies them back to the interpreter structures.

For every control-flow branch in the source program, the recorder generates conditional exit LIR instructions. These instructions exit from the trace if required control flow is different from what it was at trace recording, ensuring that the trace instructions are run only if they are supposed to. We call these instructions *guard* instructions.

Most of our traces represent loops and end with the special `loop` LIR instruction. This is just an unconditional branch to the top of the trace. Such traces return only via guards.

Now, we describe the key optimizations that are performed as part of recording LIR. All of these optimizations reduce complex dynamic language constructs to simple typed constructs by specializing for the current trace. Each optimization requires guard instructions to verify their assumptions about the state and exit the trace if necessary.

Type specialization.

All LIR primitives apply to operands of specific types. Thus, LIR traces are necessarily type-specialized, and a compiler can easily produce a translation that requires no type dispatches. A typical bytecode interpreter carries tag bits along with each value, and to perform any operation, must check the tag bits, dynamically dispatch, mask out the tag bits to recover the untagged value, perform the operation, and then reapply tags. LIR omits everything except the operation itself.

A potential problem is that some operations can produce values of unpredictable types. For example, reading a property from an object could yield a value of any type, not necessarily the type observed during recording. The recorder emits guard instructions that conditionally exit if the operation yields a value of a different type from that seen during recording. These guard instructions guarantee that as long as execution is on trace, the types of values match those of the typed trace. When the VM observes a side exit along such a type guard, a new typed trace is recorded originating at the side exit location, capturing the new type of the operation in question.

Representation specialization: objects. In JavaScript, name lookup semantics are complex and potentially expensive because they include features like object inheritance and `eval`. To evaluate an object property read expression like `o.x`, the interpreter must search the property map of `o` and all of its prototypes and parents. Property maps can be implemented with different data structures (e.g., per-object hash tables or shared hash tables), so the search process also must dispatch on the representation of each object found during search. TraceMonkey can simply observe the result of the search process and record the simplest possible LIR to access the property value. For example, the search might finds the value of `o.x` in the prototype of `o`, which uses a shared hash-table representation that places `x` in slot 2 of a property vector. Then the recorded can generate LIR that reads `o.x` with just two or three loads: one to get the prototype, possibly one to get the property value vector, and one more to get slot 2 from the vector. This is a vast simplification and speedup compared to the original interpreter code. Inheritance relationships and object representations can change during execution, so the simplified code requires guard instructions that ensure the object representation is the same. In TraceMonkey, objects' rep-

resentations are assigned an integer key called the *object shape*. Thus, the guard is a simple equality check on the object shape.

Representation specialization: numbers. JavaScript has no integer type, only a Number type that is the set of 64-bit IEEE-754 floating-pointer numbers ("doubles"). But many JavaScript operators, in particular array accesses and bitwise operators, really operate on integers, so they first convert the number to an integer, and then convert any integer result back to a double.[1] Clearly, a JavaScript VM that wants to be fast must find a way to operate on integers directly and avoid these conversions.

In TraceMonkey, we support two representations for numbers: integers and doubles. The interpreter uses integer representations as much as it can, switching for results that can only be represented as doubles. When a trace is started, some values may be imported and represented as integers. Some operations on integers require guards. For example, adding two integers can produce a value too large for the integer representation.

Function inlining. LIR traces can cross function boundaries in either direction, achieving function inlining. Move instructions need to be recorded for function entry and exit to copy arguments in and return values out. These move statements are then optimized away by the compiler using copy propagation. In order to be able to return to the interpreter, the trace must also generate LIR to record that a call frame has been entered and exited. The frame entry and exit LIR saves just enough information to allow the intepreter call stack to be restored later and is much simpler than the interpreter's standard call code. If the function being entered is not constant (which in JavaScript includes any call by function name), the recorder must also emit LIR to guard that the function is the same.

Guards and side exits. Each optimization described above requires one or more guards to verify the assumptions made in doing the optimization. A guard is just a group of LIR instructions that performs a test and conditional exit. The exit branches to a *side exit*, a small off-trace piece of LIR that returns a pointer to a structure that describes the reason for the exit along with the interpreter PC at the exit point and any other data needed to restore the interpreter's state structures.

Aborts. Some constructs are difficult to record in LIR traces. For example, `eval` or calls to external functions can change the program state in unpredictable ways, making it difficult for the tracer to know the current type map in order to continue tracing. A tracing implementation can also have any number of other limitations, e.g.,a small-memory device may limit the length of traces. When any situation occurs that prevents the implementation from continuing trace recording, the implementation *aborts* trace recording and returns to the trace monitor.

3.2 Trace Trees

Especially simple loops, namely those where control flow, value types, value representations, and inlined functions are all invariant, can be represented by a single trace. But most loops have at least some variation, and so the program will take side exits from the main trace. When a side exit becomes hot, TraceMonkey starts a new *branch trace* from that point and patches the side exit to jump directly to that trace. In this way, a single trace expands on demand to a single-entry, multiple-exit *trace tree*.

This section explains how trace trees are formed during execution. The goal is to form trace trees during execution that cover all the hot paths of the program.

[1] Arrays are actually worse than this: if the index value is a number, it must be converted from a double to a string for the property access operator, and then to an integer internally to the array implementation.

Starting a tree. Tree trees always start at loop headers, because they are a natural place to look for hot paths. In TraceMonkey, loop headers are easy to detect–the bytecode compiler ensures that a bytecode is a loop header iff it is the target of a backward branch. TraceMonkey starts a tree when a given loop header has been executed a certain number of times (2 in the current implementation). Starting a tree just means starting recording a trace for the current point and type map and marking the trace as the root of a tree. Each tree is associated with a loop header and type map, so there may be several trees for a given loop header.

Closing the loop. Trace recording can end in several ways.

Ideally, the trace reaches the loop header where it started with the same type map as on entry. This is called a *type-stable* loop iteration. In this case, the end of the trace can jump right to the beginning, as all the value representations are exactly as needed to enter the trace. The jump can even skip the usual code that would copy out the state at the end of the trace and copy it back in to the trace activation record to enter a trace.

In certain cases the trace might reach the loop header with a different type map. This scenario is sometime observed for the first iteration of a loop. Some variables inside the loop might initially be *undefined*, before they are set to a concrete type during the first loop iteration. When recording such an iteration, the recorder cannot link the trace back to its own loop header since it is *type-unstable*. Instead, the iteration is terminated with a side exit that will always fail and return to the interpreter. At the same time a new trace is recorded with the new type map. Every time an additional type-unstable trace is added to a region, its exit type map is compared to the entry map of all existing traces in case they complement each other. With this approach we are able to cover type-unstable loop iterations as long they eventually form a stable equilibrium.

Finally, the trace might exit the loop before reaching the loop header, for example because execution reaches a `break` or `return` statement. In this case, the VM simply ends the trace with an exit to the trace monitor.

As mentioned previously, we may speculatively chose to represent certain Number-typed values as integers on trace. We do so when we observe that Number-typed variables contain an integer value at trace entry. If during trace recording the variable is unexpectedly assigned a non-integer value, we have to widen the type of the variable to a double. As a result, the recorded trace becomes inherently type-unstable since it starts with an integer value but ends with a double value. This represents a mis-speculation, since at trace entry we specialized the Number-typed value to an integer, assuming that at the loop edge we would again find an integer value in the variable, allowing us to close the loop. To avoid future speculative failures involving this variable, and to obtain a type-stable trace we note the fact that the variable in question as been observed to sometimes hold non-integer values in an advisory data structure which we call the "oracle".

When compiling loops, we consult the oracle before specializing values to integers. Speculation towards integers is performed only if no adverse information is known to the oracle about that particular variable. Whenever we accidentally compile a loop that is type-unstable due to mis-speculation of a Number-typed variable, we immediately trigger the recording of a new trace, which based on the now updated oracle information will start with a double value and thus become type stable.

Extending a tree. Side exits lead to different paths through the loop, or paths with different types or representations. Thus, to completely cover the loop, the VM must record traces starting at all side exits. These traces are recorded much like root traces: there is a counter for each side exit, and when the counter reaches a hotness threshold, recording starts. Recording stops exactly as for the root trace, using the loop header of the root trace as the target to reach.

Our implementation does not extend at all side exits. It extends only if the side exit is for a control-flow branch, and only if the side exit does not leave the loop. In particular we do not want to extend a trace tree along a path that leads to an outer loop, because we want to cover such paths in an outer tree through tree *nesting*.

3.3 Blacklisting

Sometimes, a program follows a path that cannot be compiled into a trace, usually because of limitations in the implementation. TraceMonkey does not currently support recording throwing and catching of arbitrary exceptions. This design trade off was chosen, because exceptions are usually rare in JavaScript. However, if a program opts to use exceptions intensively, we would suddenly incur a punishing runtime overhead if we repeatedly try to record a trace for this path and repeatedly fail to do so, since we abort tracing every time we observe an exception being thrown.

As a result, if a hot loop contains traces that always fail, the VM could potentially run much more slowly than the base interpreter: the VM repeatedly spends time trying to record traces, but is never able to run any. To avoid this problem, whenever the VM is about to start tracing, it must try to predict whether it will finish the trace.

Our prediction algorithm is based on *blacklisting* traces that have been tried and failed. When the VM fails to finish a trace starting at a given point, the VM records that a failure has occurred. The VM also sets a counter so that it will not try to record a trace starting at that point until it is passed a few more times (32 in our implementation). This *backoff* counter gives temporary conditions that prevent tracing a chance to end. For example, a loop may behave differently during startup than during its steady-state execution. After a given number of failures (2 in our implementation), the VM marks the fragment as blacklisted, which means the VM will never again start recording at that point.

After implementing this basic strategy, we observed that for small loops that get blacklisted, the system can spend a noticeable amount of time just finding the loop fragment and determining that it has been blacklisted. We now avoid that problem by patching the bytecode. We define an extra no-op bytecode that indicates a loop header. The VM calls into the trace monitor every time the interpreter executes a loop header no-op. To blacklist a fragment, we simply replace the loop header no-op with a regular no-op. Thus, the interpreter will never again even call into the trace monitor.

There is a related problem we have not yet solved, which occurs when a loop meets all of these conditions:

- The VM can form at least one root trace for the loop.
- There is at least one hot side exit for which the VM cannot complete a trace.
- The loop body is short.

In this case, the VM will repeatedly pass the loop header, search for a trace, find it, execute it, and fall back to the interpreter. With a short loop body, the overhead of finding and calling the trace is high, and causes performance to be even slower than the basic interpreter. So far, in this situation we have improved the implementation so that the VM can complete the branch trace. But it is hard to guarantee that this situation will never happen. As future work, this situation could be avoided by detecting and blacklisting loops for which the average trace call executes few bytecodes before returning to the interpreter.

4. Nested Trace Tree Formation

Figure 7 shows basic trace tree compilation (11) applied to a nested loop where the inner loop contains two paths. Usually, the inner loop (with header at i_2) becomes hot first, and a trace tree is rooted at that point. For example, the first recorded trace may be a cycle

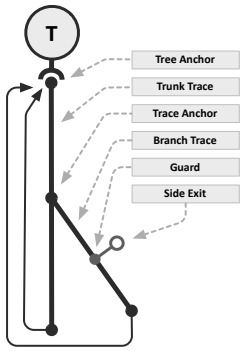


Figure 5. A tree with two traces, a trunk trace and one branch trace. The trunk trace contains a guard to which a branch trace was attached. The branch trace contain a guard that may fail and trigger a side exit. Both the trunk and the branch trace loop back to the tree anchor, which is the beginning of the trace tree.

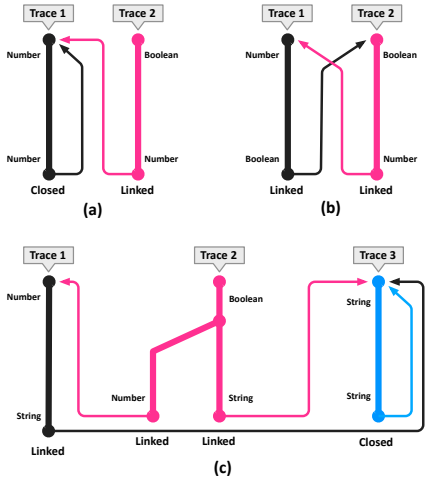


Figure 6. We handle type-unstable loops by allowing traces to compile that cannot loop back to themselves due to a type mismatch. As such traces accumulate, we attempt to connect their loop edges to form groups of trace trees that can execute without having to side-exit to the interpreter to cover odd type cases. This is particularly important for nested trace trees where an outer tree tries to call an inner tree (or in this case a forest of inner trees), since inner loops frequently have initially undefined values which change type to a concrete value after the first iteration.

through the inner loop, $\{i_2, i_3, i_5, \alpha\}$. The α symbol is used to indicate that the trace loops back the tree anchor.

When execution leaves the inner loop, the basic design has two choices. First, the system can stop tracing and give up on compiling the outer loop, clearly an undesirable solution. The other choice is to continue tracing, compiling traces for the outer loop inside the inner loop's trace tree.

For example, the program might exit at i_5 and record a branch trace that incorporates the outer loop: $\{i_5, i_7, i_1, i_6, i_7, i_1, \alpha\}$. Later, the program might take the other branch at i_2 and then exit, recording another branch trace incorporating the outer loop: $\{i_2, i_4, i_5, i_7, i_1, i_6, i_7, i_1, \alpha\}$. Thus, the outer loop is recorded and compiled twice, and both copies must be retained in the trace cache.

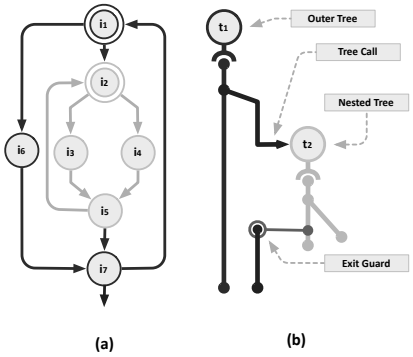


Figure 7. Control flow graph of a nested loop with an if statement inside the inner most loop (a). An inner tree captures the inner loop, and is nested inside an outer tree which "calls" the inner tree. The inner tree returns to the outer tree once it exits along its loop condition guard (b).

In general, if loops are nested to depth k, and each loop has n paths (on geometric average), this naïve strategy yields $O(n^k)$ traces, which can easily fill the trace cache.

In order to execute programs with nested loops efficiently, a tracing system needs a technique for covering the nested loops with native code without exponential trace duplication.

4.1 Nesting Algorithm

The key insight is that if each loop is represented by its own trace tree, the code for each loop can be contained only in its own tree, and outer loop paths will not be duplicated. Another key fact is that we are not tracing arbitrary bytecodes that might have irreduceable control flow graphs, but rather bytecodes produced by a compiler for a language with structured control flow. Thus, given two loop edges, the system can easily determine whether they are nested and which is the inner loop. Using this knowledge, the system can compile inner and outer loops separately, and make the outer loop's traces *call* the inner loop's trace tree.

The algorithm for building nested trace trees is as follows. We start tracing at loop headers exactly as in the basic tracing system. When we exit a loop (detected by comparing the interpreter PC with the range given by the loop edge), we stop the trace. The key step of the algorithm occurs when we are recording a trace for loop L_R (R for loop being recorded) and we reach the header of a different loop L_O (O for other loop). Note that L_O must be an inner loop of L_R because we stop the trace when we exit a loop.

- If L_O has a type-matching compiled trace tree, we call L_O as a nested trace tree. If the call succeeds, then we record the call in the trace for L_R. On future executions, the trace for L_R will call the inner trace directly.
- If L_O does not have a type-matching compiled trace tree yet, we have to obtain it before we are able to proceed. In order to do this, we simply abort recording the first trace. The trace monitor will see the inner loop header, and will immediately start recording the inner loop. [2]

If all the loops in a nest are type-stable, then loop nesting creates no duplication. Otherwise, if loops are nested to a depth k, and each loop is entered with m different type maps (on geometric average), then we compile $O(m^k)$ copies of the innermost loop. As long as m is close to 1, the resulting trace trees will be tractable.

[2] Instead of aborting the outer recording, we could principally merely suspend the recording, but that would require the implementation to be able to record several traces simultaneously, complicating the implementation, while saving only a few iterations in the interpreter.

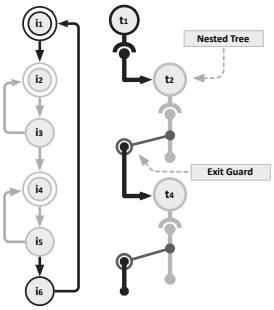


Figure 8. Control flow graph of a loop with two nested loops (left) and its nested trace tree configuration (right). The outer tree calls the two inner nested trace trees and places guards at their side exit locations.

An important detail is that the call to the inner trace tree must act like a function call site: it must return to the same point every time. The goal of nesting is to make inner and outer loops independent; thus when the inner tree is called, it must exit to the same point in the outer tree every time with the same type map. Because we cannot actually guarantee this property, we must guard on it after the call, and side exit if the property does not hold. A common reason for the inner tree not to return to the same point would be if the inner tree took a new side exit for which it had never compiled a trace. At this point, the interpreter PC is in the inner tree, so we cannot continue recording or executing the outer tree. If this happens during recording, we abort the outer trace, to give the inner tree a chance to finish growing. A future execution of the outer tree would then be able to properly finish and record a call to the inner tree. If an inner tree side exit happens during execution of a compiled trace for the outer tree, we simply exit the outer trace and start recording a new branch in the inner tree.

4.2 Blacklisting with Nesting

The blacklisting algorithm needs modification to work well with nesting. The problem is that outer loop traces often abort during startup (because the inner tree is not available or takes a side exit), which would lead to their being quickly blacklisted by the basic algorithm.

The key observation is that when an outer trace aborts because the inner tree is not ready, this is probably a temporary condition. Thus, we should not count such aborts toward blacklisting as long as we are able to build up more traces for the inner tree.

In our implementation, when an outer tree aborts on the inner tree, we increment the outer tree's blacklist counter as usual and back off on compiling it. When the inner tree finishes a trace, we decrement the blacklist counter on the outer loop, "forgiving" the outer loop for aborting previously. We also undo the backoff so that the outer tree can start immediately trying to compile the next time we reach it.

5. Trace Tree Optimization

This section explains how a recorded trace is translated to an optimized machine code trace. The trace compilation subsystem, NANOJIT, is separate from the VM and can be used for other applications.

5.1 Optimizations

Because traces are in SSA form and have no join points or ϕ-nodes, certain optimizations are easy to implement. In order to get good startup performance, the optimizations must run quickly, so we chose a small set of optimizations. We implemented the optimizations as pipelined filters so that they can be turned on and off independently, and yet all run in just two loop passes over the trace: one forward and one backward.

Every time the trace recorder emits a LIR instruction, the instruction is immediately passed to the first filter in the forward pipeline. Thus, forward filter optimizations are performed as the trace is recorded. Each filter may pass each instruction to the next filter unchanged, write a different instruction to the next filter, or write no instruction at all. For example, the constant folding filter can replace a multiply instruction like $v_{13} := mul3, 1000$ with a constant instruction $v_{13} = 3000$.

We currently apply four forward filters:

- On ISAs without floating-point instructions, a soft-float filter converts floating-point LIR instructions to sequences of integer instructions.
- CSE (constant subexpression elimination),
- expression simplification, including constant folding and a few algebraic identities (e.g., $a - a = 0$), and
- source language semantic-specific expression simplification, primarily algebraic identities that allow DOUBLE to be replaced with INT. For example, LIR that converts an INT to a DOUBLE and then back again would be removed by this filter.

When trace recording is completed, nanojit runs the backward optimization filters. These are used for optimizations that require backward program analysis. When running the backward filters, nanojit reads one LIR instruction at a time, and the reads are passed through the pipeline.

We currently apply three backward filters:

- Dead data-stack store elimination. The LIR trace encodes many stores to locations in the interpreter stack. But these values are never read back before exiting the trace (by the interpreter or another trace). Thus, stores to the stack that are overwritten before the next exit are dead. Stores to locations that are off the top of the interpreter stack at future exits are also dead.
- Dead call-stack store elimination. This is the same optimization as above, except applied to the interpreter's call stack used for function call inlining.
- Dead code elimination. This eliminates any operation that stores to a value that is never used.

After a LIR instruction is successfully read ("pulled") from the backward filter pipeline, nanojit's code generator emits native machine instruction(s) for it.

5.2 Register Allocation

We use a simple greedy register allocator that makes a single backward pass over the trace (it is integrated with the code generator). By the time the allocator has reached an instruction like $v_3 = add\ v_1, v_2$, it has already assigned a register to v_3. If v_1 and v_2 have not yet been assigned registers, the allocator assigns a free register to each. If there are no free registers, a value is selected for spilling. We use a class heuristic that selects the "oldest" register-carried value (6).

The heuristic considers the set R of values v in registers immediately after the current instruction for spilling. Let v_m be the last instruction before the current where each v is referred to. Then the heuristic selects v with minimum v_m. The motivation is that this frees up a register for as long as possible given a single spill.

If we need to spill a value v_s at this point, we generate the restore code just after the code for the current instruction. The corresponding spill code is generated just after the last point where v_s was used. The register that was assigned to v_s is marked free for the preceding code, because that register can now be used freely without affecting the following code

Tag	JS Type	Description
xx1	number	31-bit integer representation
000	object	pointer to JSObject handle
010	number	pointer to double handle
100	string	pointer to JSString handle
110	boolean	enumeration for null, undefined, true, false
	null, or	
	undefined	

Figure 9. Tagged values in the SpiderMonkey JS interpreter. Testing tags, unboxing (extracting the untagged value) and boxing (creating tagged values) are significant costs. Avoiding these costs is a key benefit of tracing.

6. Implementation

To demonstrate the effectiveness of our approach, we have implemented a trace-based dynamic compiler for the SpiderMonkey JavaScript Virtual Machine (4). SpiderMonkey is the JavaScript VM embedded in Mozilla's Firefox open-source web browser (2), which is used by more than 200 million users world-wide. The core of SpiderMonkey is a bytecode interpreter implemented in C++.

In SpiderMonkey, all JavaScript values are represented by the type `jsval`. A `jsval` is machine word in which up to the 3 of the least significant bits are a type tag, and the remaining bits are data. See Figure 6 for details. All pointers contained in `jsvals` point to GC-controlled blocks aligned on 8-byte boundaries.

JavaScript *object* values are mappings of string-valued property names to arbitrary values. They are represented in one of two ways in SpiderMonkey. Most objects are represented by a shared structural description, called the *object shape*, that maps property names to array indexes using a hash table. The object stores a pointer to the shape and the array of its own property values. Objects with large, unique sets of property names store their properties directly in a hash table.

The garbage collector is an exact, non-generational, stop-the-world mark-and-sweep collector.

In the rest of this section we discuss key areas of the TraceMonkey implementation.

6.1 Calling Compiled Traces

Compiled traces are stored in a *trace cache*, indexed by intepreter PC and type map. Traces are compiled so that they may be called as functions using standard native calling conventions (e.g., `FASTCALL` on x86).

The interpreter must hit a loop edge and enter the monitor in order to call a native trace for the first time. The monitor computes the current type map, checks the trace cache for a trace for the current PC and type map, and if it finds one, executes the trace.

To execute a trace, the monitor must build a trace activation record containing imported local and global variables, temporary stack space, and space for arguments to native calls. The local and global values are then copied from the interpreter state to the trace activation record. Then, the trace is called like a normal C function pointer.

When a trace call returns, the monitor restores the interpreter state. First, the monitor checks the reason for the trace exit and applies blacklisting if needed. Then, it pops or synthesizes interpreter JavaScript call stack frames as needed. Finally, it copies the imported variables back from the trace activation record to the interpreter state.

At least in the current implementation, these steps have a non-negligible runtime cost, so minimizing the number of interpreter-to-trace and trace-to-interpreter transitions is essential for performance. (see also Section 3.3). Our experiments (see Figure 12) show that for programs we can trace well such transitions happen infrequently and hence do not contribute significantly to total runtime. In a few programs, where the system is prevented from recording branch traces for hot side exits by aborts, this cost can rise to up to 10% of total execution time.

6.2 Trace Stitching

Transitions from a trace to a branch trace at a side exit avoid the costs of calling traces from the monitor, in a feature called *trace stitching*. At a side exit, the exiting trace only needs to write live register-carried values back to its trace activation record. In our implementation, identical type maps yield identical activation record layouts, so the trace activation record can be reused immediately by the branch trace.

In programs with branchy trace trees with small traces, trace stitching has a noticeable cost. Although writing to memory and then soon reading back would be expected to have a high L1 cache hit rate, for small traces the increased instruction count has a noticeable cost. Also, if the writes and reads are very close in the dynamic instruction stream, we have found that current x86 processors often incur penalties of 6 cycles or more (e.g., if the instructions use different base registers with equal values, the processor may not be able to detect that the addresses are the same right away).

The alternate solution is to recompile an entire trace tree, thus achieving inter-trace register allocation (10). The disadvantage is that tree recompilation takes time quadratic in the number of traces. We believe that the cost of recompiling a trace tree every time a branch is added would be prohibitive. That problem might be mitigated by recompiling only at certain points, or only for very hot, stable trees.

In the future, multicore hardware is expected to be common, making background tree recompilation attractive. In a closely related project (13) background recompilation yielded speedups of up to 1.25x on benchmarks with many branch traces. We plan to apply this technique to TraceMonkey as future work.

6.3 Trace Recording

The job of the trace recorder is to emit LIR with identical semantics to the currently running interpreter bytecode trace. A good implementation should have low impact on non-tracing interpreter performance and a convenient way for implementers to maintain semantic equivalence.

In our implementation, the only direct modification to the interpreter is a call to the trace monitor at loop edges. In our benchmark results (see Figure 12) the total time spent in the monitor (for all activities) is usually less than 5%, so we consider the interpreter impact requirement met. Incrementing the loop hit counter is expensive because it requires us to look up the loop in the trace cache, but we have tuned our loops to become hot and trace very quickly (on the second iteration). The hit counter implementation could be improved, which might give us a small increase in overall performance, as well as more flexibility with tuning hotness thresholds. Once a loop is blacklisted we never call into the trace monitor for that loop (see Section 3.3).

Recording is activated by a pointer swap that sets the interpreter's dispatch table to call a single "interrupt" routine for every bytecode. The interrupt routine first calls a bytecode-specific recording routine. Then, it turns off recording if necessary (e.g., the trace ended). Finally, it jumps to the standard interpreter bytecode implementation. Some bytecodes have effects on the type map that cannot be predicted before executing the bytecode (e.g., calling `String.charCodeAt`, which returns an integer or *NaN* if the index argument is out of range). For these, we arrange for the interpreter to call into the recorder again after executing the bytecode. Since such hooks are relatively rare, we embed them directly into the interpreter, with an additional runtime check to see whether a recorder is currently active.

While separating the interpreter from the recorder reduces individual code complexity, it also requires careful implementation and extensive testing to achieve semantic equivalence.

In some cases achieving this equivalence is difficult since SpiderMonkey follows a *fat-bytecode* design, which was found to be beneficial to pure interpreter performance.

In fat-bytecode designs, individual bytecodes can implement complex processing (e.g., the `getprop` bytecode, which implements full JavaScript property value access, including special cases for cached and dense array access).

Fat bytecodes have two advantages: fewer bytecodes means lower dispatch cost, and bigger bytecode implementations give the compiler more opportunities to optimize the interpreter.

Fat bytecodes are a problem for TraceMonkey because they require the recorder to reimplement the same special case logic in the same way. Also, the advantages are reduced because (a) dispatch costs are eliminated entirely in compiled traces, (b) the traces contain only one special case, not the interpreter's large chunk of code, and (c) TraceMonkey spends less time running the base interpreter.

One way we have mitigated these problems is by implementing certain complex bytecodes in the recorder as sequences of simple bytecodes. Expressing the original semantics this way is not too difficult, and recording simple bytecodes is much easier. This enables us to retain the advantages of fat bytecodes while avoiding some of their problems for trace recording. This is particularly effective for fat bytecodes that recurse back into the interpreter, for example to convert an object into a primitive value by invoking a well-known method on the object, since it lets us inline this function call.

It is important to note that we split fat opcodes into thinner opcodes only during recording. When running purely interpretatively (i.e. code that has been blacklisted), the interpreter directly and efficiently executes the fat opcodes.

6.4 Preemption

SpiderMonkey, like many VMs, needs to preempt the user program periodically. The main reasons are to prevent infinitely looping scripts from locking up the host system and to schedule GC.

In the interpreter, this had been implemented by setting a "preempt now" flag that was checked on every backward jump. This strategy carried over into TraceMonkey: the VM inserts a guard on the preemption flag at every loop edge. We measured less than a 1% increase in runtime on most benchmarks for this extra guard. In practice, the cost is detectable only for programs with very short loops.

We tested and rejected a solution that avoided the guards by compiling the loop edge as an unconditional jump, and patching the jump target to an exit routine when preemption is required. This solution can make the normal case slightly faster, but then preemption becomes very slow. The implementation was also very complex, especially trying to restart execution after the preemption.

6.5 Calling External Functions

Like most interpreters, SpiderMonkey has a foreign function interface (FFI) that allows it to call C builtins and host system functions (e.g., web browser control and DOM access). The FFI has a standard signature for JS-callable functions, the key argument of which is an array of boxed values. External functions called through the FFI interact with the program state through an interpreter API (e.g., to read a property from an argument). There are also certain interpreter builtins that do not use the FFI, but interact with the program state in the same way, such as the `CallIteratorNext` function used with iterator objects. TraceMonkey must support this FFI in order to speed up code that interacts with the host system inside hot loops.

Calling external functions from TraceMonkey is potentially difficult because traces do not update the interpreter state until exiting. In particular, external functions may need the call stack or the global variables, but they may be out of date.

For the out-of-date call stack problem, we refactored some of the interpreter API implementation functions to re-materialize the interpreter call stack on demand.

We developed a C++ static analysis and annotated some interpreter functions in order to verify that the call stack is refreshed at any point it needs to be used. In order to access the call stack, a function must be annotated as either FORCESSTACK or REQUIRESSTACK. These annotations are also required in order to call REQUIRESSTACK functions, which are presumed to access the call stack transitively. FORCESSTACK is a trusted annotation, applied to only 5 functions, that means the function refreshes the call stack. REQUIRESSTACK is an untrusted annotation that means the function may only be called if the call stack has already been refreshed.

Similarly, we detect when host functions attempt to directly read or write global variables, and force the currently running trace to side exit. This is necessary since we cache and unbox global variables into the activation record during trace execution.

Since both call-stack access and global variable access are rarely performed by host functions, performance is not significantly affected by these safety mechanisms.

Another problem is that external functions can reenter the interpreter by calling scripts, which in turn again might want to access the call stack or global variables. To address this problem, we made the VM set a flag whenever the interpreter is reentered while a compiled trace is running.

Every call to an external function then checks this flag and exits the trace immediately after returning from the external function call if it is set. There are many external functions that seldom or never reenter, and they can be called without problem, and will cause trace exit only if necessary.

The FFI's boxed value array requirement has a performance cost, so we defined a new FFI that allows C functions to be annotated with their argument types so that the tracer can call them directly, without unnecessary argument conversions.

Currently, we do not support calling native property get and set override functions or DOM functions directly from trace. Support is planned future work.

6.6 Correctness

During development, we had access to existing JavaScript test suites, but most of them were not designed with tracing VMs in mind and contained few loops.

One tool that helped us greatly was Mozilla's JavaScript fuzz tester, JSFUNFUZZ, which generates random JavaScript programs by nesting random language elements. We modified JSFUNFUZZ to generate loops, and also to test more heavily certain constructs we suspected would reveal flaws in our implementation. For example, we suspected bugs in TraceMonkey's handling of type-unstable loops and heavily branching code, and a specialized fuzz tester indeed revealed several regressions which we subsequently corrected.

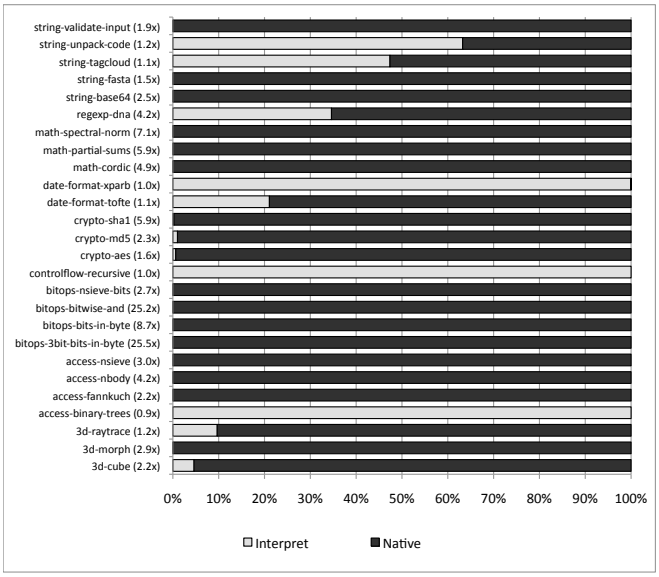


Figure 11. Fraction of dynamic bytecodes executed by interpreter and on native traces. The speedup vs. interpreter is shown in parentheses next to each test. The fraction of bytecodes executed while recording is too small to see in this figure, except for `crypto-md5`, where fully 3% of bytecodes are executed while recording. In most of the tests, almost all the bytecodes are executed by compiled traces. Three of the benchmarks are not traced at all and run in the interpreter.

7. Evaluation

We evaluated our JavaScript tracing implementation using SunSpider, the industry standard JavaScript benchmark suite. SunSpider consists of 26 short-running (less than 250ms, average 26ms) JavaScript programs. This is in stark contrast to benchmark suites such as SpecJVM98 (3) used to evaluate desktop and server Java VMs. Many programs in those benchmarks use large data sets and execute for minutes. The SunSpider programs carry out a variety of tasks, primarily 3d rendering, bit-bashing, cryptographic encoding, math kernels, and string processing.

All experiments were performed on a MacBook Pro with 2.2 GHz Core 2 processor and 2 GB RAM running MacOS 10.5.

Benchmark results. The main question is whether programs run faster with tracing. For this, we ran the standard SunSpider test driver, which starts a JavaScript interpreter, loads and runs each program once for warmup, then loads and runs each program 10 times and reports the average time taken by each. We ran 4 different configurations for comparison: (a) SpiderMonkey, the baseline interpreter, (b) TraceMonkey, (d) SquirrelFish Extreme (SFX), the call-threaded JavaScript interpreter used in Apple's WebKit, and (e) V8, the method-compiling JavaScript VM from Google.

Figure 10 shows the relative speedups achieved by tracing, SFX, and V8 against the baseline (SpiderMonkey). Tracing achieves the best speedups in integer-heavy benchmarks, up to the 25x speedup on `bitops-bitwise-and`.

TraceMonkey is the fastest VM on 9 of the 26 benchmarks (`3d-morph`, `bitops-3bit-bits-in-byte`, `bitops-bitwise-and`, `crypto-sha1`, `math-cordic`, `math-partial-sums`, `math-spectral-norm`, `string-base64`, `string-validate-input`).

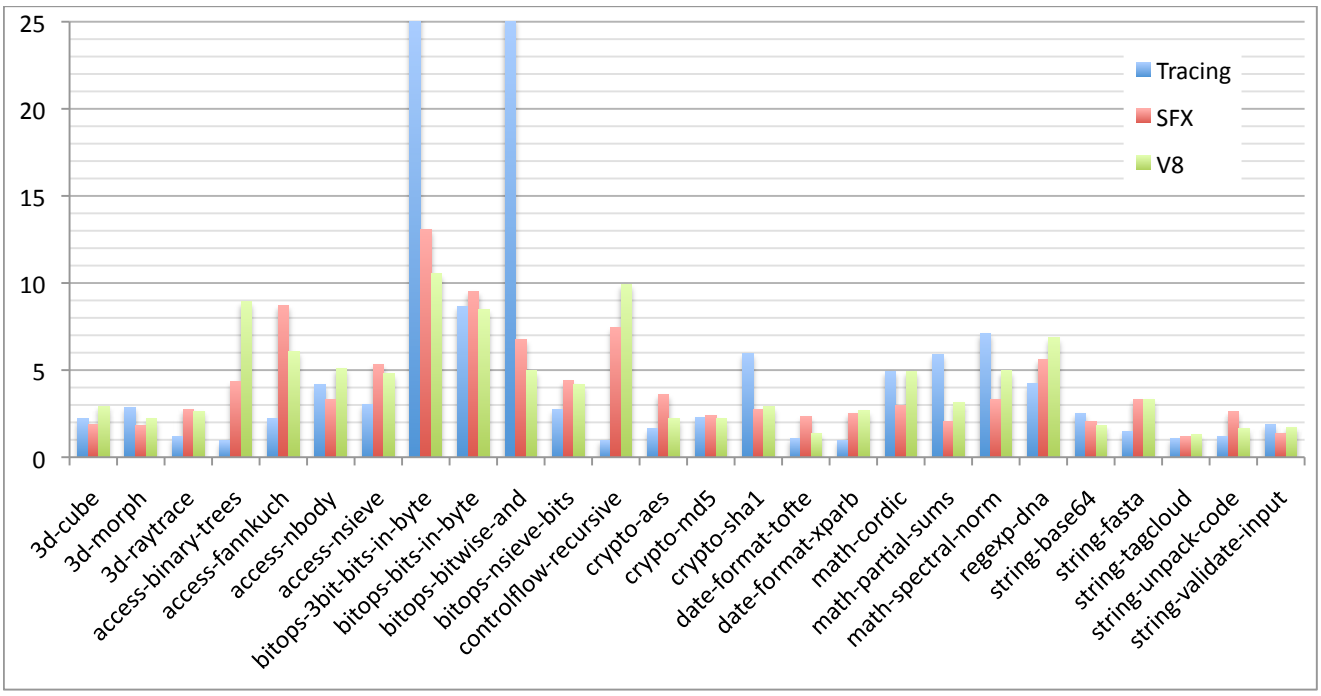


Figure 10. Speedup vs. a baseline JavaScript interpreter (SpiderMonkey) for our trace-based JIT compiler, Apple's SquirrelFish Extreme inline threading interpreter and Google's V8 JS compiler. Our system generates particularly efficient code for programs that benefit most from type specialization, which includes SunSpider Benchmark programs that perform bit manipulation. We type-specialize the code in question to use integer arithmetic, which substantially improves performance. For one of the benchmark programs we execute 25 times faster than the SpiderMonkey interpreter, and almost 5 times faster than V8 and SFX. For a large number of benchmarks all three VMs produce similar results. We perform worst on benchmark programs that we do not trace and instead fall back onto the interpreter. This includes the recursive benchmarks `access-binary-trees` and `control-flow-recursive`, for which we currently don't generate any native code.

In particular, the `bitops` benchmarks are short programs that perform many bitwise operations, so TraceMonkey can cover the entire program with 1 or 2 traces that operate on integers. TraceMonkey runs all the other programs in this set almost entirely as native code.

`regexp-dna` is dominated by regular expression matching, which is implemented in all 3 VMs by a special regular expression compiler. Thus, performance on this benchmark has little relation to the trace compilation approach discussed in this paper.

TraceMonkey's smaller speedups on the other benchmarks can be attributed to a few specific causes:

- The implementation does not currently trace recursion, so TraceMonkey achieves a small speedup or no speedup on benchmarks that use recursion extensively: `3d-cube`, `3d-raytrace`, `access-binary-trees`, `string-tagcloud`, and `controlflow-recursive`.
- The implementation does not currently trace `eval` and some other functions implemented in C. Because `date-format-tofte` and `date-format-xparb` use such functions in their main loops, we do not trace them.
- The implementation does not currently trace through regular expression `replace` operations. The replace function can be passed a function object used to compute the replacement text. Our implementation currently does not trace functions called as replace functions. The run time of `string-unpack-code` is dominated by such a `replace` call.
- Two programs trace well, but have a long compilation time. `access-nbody` forms a large number of traces (81). `crypto-md5` forms one very long trace. We expect to improve performance on this programs by improving the compilation speed of nanojit.
- Some programs trace very well, and speed up compared to the interpreter, but are not as fast as SFX and/or V8, namely `bitops-bits-in-byte`, `bitops-nsieve-bits`, `access-fannkuch`, `access-nsieve`, and `crypto-aes`. The reason is not clear, but all of these programs have nested loops with small bodies, so we suspect that the implementation has a relatively high cost for calling nested traces. `string-fasta` traces well, but its run time is dominated by string processing builtins, which are unaffected by tracing and seem to be less efficient in SpiderMonkey than in the two other VMs.

Detailed performance metrics. In Figure 11 we show the fraction of instructions interpreted and the fraction of instructions executed as native code. This figure shows that for many programs, we are able to execute almost all the code natively.

Figure 12 breaks down the total execution time into four activities: interpreting bytecodes while not recording, recording traces (including time taken to interpret the recorded trace), compiling traces to native code, and executing native code traces.

These detailed metrics allow us to estimate parameters for a simple model of tracing performance. These estimates should be considered very rough, as the values observed on the individual benchmarks have large standard deviations (on the order of the

	Loops	Trees	Traces	Aborts	Flushes	Trees/Loop	Traces/Tree	Traces/Loop	Speedup
3d-cube	25	27	29	3	0	1.1	1.1	1.2	2.20x
3d-morph	5	8	8	2	0	1.6	1.0	1.6	2.86x
3d-raytrace	10	25	100	10	1	2.5	4.0	10.0	1.18x
access-binary-trees	0	0	0	5	0	-	-	-	0.93x
access-fannkuch	10	34	57	24	0	3.4	1.7	5.7	2.20x
access-nbody	8	16	18	5	0	2.0	1.1	2.3	4.19x
access-nsieve	3	6	8	3	0	2.0	1.3	2.7	3.05x
bitops-3bit-bits-in-byte	2	2	2	0	0	1.0	1.0	1.0	25.47x
bitops-bits-in-byte	3	3	4	1	0	1.0	1.3	1.3	8.67x
bitops-bitwise-and	1	1	1	0	0	1.0	1.0	1.0	25.20x
bitops-nsieve-bits	3	3	5	0	0	1.0	1.7	1.7	2.75x
controlflow-recursive	0	0	0	1	0	-	-	-	0.98x
crypto-aes	50	72	78	19	0	1.4	1.1	1.6	1.64x
crypto-md5	4	4	5	0	0	1.0	1.3	1.3	2.30x
crypto-sha1	5	5	10	0	0	1.0	2.0	2.0	5.95x
date-format-tofte	3	3	4	7	0	1.0	1.3	1.3	1.07x
date-format-xparb	3	3	11	3	0	1.0	3.7	3.7	0.98x
math-cordic	2	4	5	1	0	2.0	1.3	2.5	4.92x
math-partial-sums	2	4	4	1	0	2.0	1.0	2.0	5.90x
math-spectral-norm	15	20	20	0	0	1.3	1.0	1.3	7.12x
regexp-dna	2	2	2	0	0	1.0	1.0	1.0	4.21x
string-base64	3	5	7	0	0	1.7	1.4	2.3	2.53x
string-fasta	5	11	15	6	0	2.2	1.4	3.0	1.49x
string-tagcloud	3	6	6	5	0	2.0	1.0	2.0	1.09x
string-unpack-code	4	4	37	0	0	1.0	9.3	9.3	1.20x
string-validate-input	6	10	13	1	0	1.7	1.3	2.2	1.86x

Figure 13. Detailed trace recording statistics for the SunSpider benchmark set.

mean). We exclude `regexp-dna` from the following calculations, because most of its time is spent in the regular expression matcher, which has much different performance characteristics from the other programs. (Note that this only makes a difference of about 10% in the results.) Dividing the total execution time in processor clock cycles by the number of bytecodes executed in the base interpreter shows that on average, a bytecode executes in about 35 cycles. Native traces take about 9 cycles per bytecode, a 3.9x speedup over the interpreter.

Using similar computations, we find that trace recording takes about 3800 cycles per bytecode, and compilation 3150 cycles per bytecode. Hence, during recording and compiling the VM runs at 1/200 the speed of the interpreter. Because it costs 6950 cycles to compile a bytecode, and we save 26 cycles each time that code is run natively, we break even after running a trace 270 times.

The other VMs we compared with achieve an overall speedup of 3.0x relative to our baseline interpreter. Our estimated native code speedup of 3.9x is significantly better. This suggests that our compilation techniques can generate more efficient native code than any other current JavaScript VM.

These estimates also indicate that our startup performance could be substantially better if we improved the speed of trace recording and compilation. The estimated 200x slowdown for recording and compilation is very rough, and may be influenced by startup factors in the interpreter (e.g., caches that have not warmed up yet during recording). One observation supporting this conjecture is that in the tracer, interpreted bytecodes take about 180 cycles to run. Still, recording and compilation are clearly both expensive, and a better implementation, possibly including redesign of the LIR abstract syntax or encoding, would improve startup performance.

Our performance results confirm that type specialization using trace trees substantially improves performance. We are able to outperform the fastest available JavaScript compiler (V8) and the fastest available JavaScript inline threaded interpreter (SFX) on 9 of 26 benchmarks.

8. Related Work

Trace optimization for dynamic languages. The closest area of related work is on applying trace optimization to type-specialize dynamic languages. Existing work shares the idea of generating type-specialized code speculatively with guards along interpreter traces.

To our knowledge, Rigo's Psyco (16) is the only published type-specializing trace compiler for a dynamic language (Python). Psyco does not attempt to identify hot loops or inline function calls. Instead, Psyco transforms loops to mutual recursion before running and traces all operations.

Pall's LuaJIT is a Lua VM in development that uses trace compilation ideas. (1). There are no publications on LuaJIT but the creator has told us that LuaJIT has a similar design to our system, but will use a less aggressive type speculation (e.g., using a floating-point representation for all number values) and does not generate nested traces for nested loops.

General trace optimization. General trace optimization has a longer history that has treated mostly native code and typed languages like Java. Thus, these systems have focused less on type specialization and more on other optimizations.

Dynamo (7) by Bala et al, introduced native code tracing as a replacement for profile-guided optimization (PGO). A major goal was to perform PGO online so that the profile was specific to the current execution. Dynamo used loop headers as candidate hot traces, but did not try to create loop traces specifically.

Trace trees were originally proposed by Gal et al. (11) in the context of Java, a statically typed language. Their trace trees actually inlined parts of outer loops within the inner loops (because

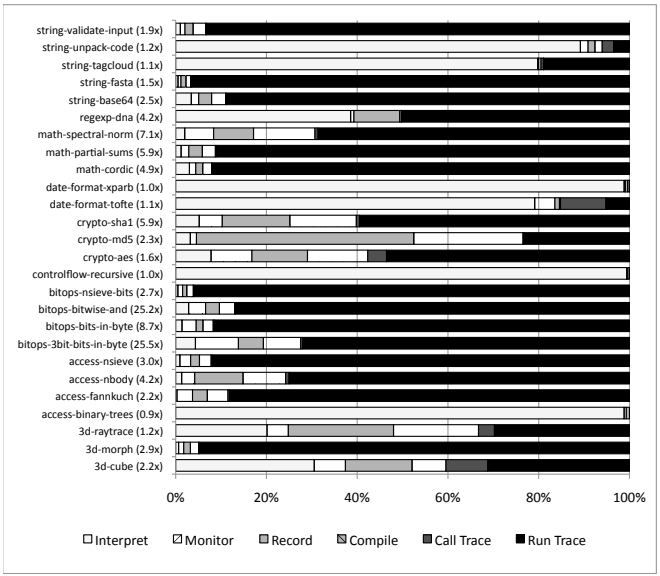


Figure 12. Fraction of time spent on major VM activities. The speedup vs. interpreter is shown in parentheses next to each test. Most programs where the VM spends the majority of its time running native code have a good speedup. Recording and compilation costs can be substantial; speeding up those parts of the implementation would improve SunSpider performance.

inner loops become hot first), leading to much greater tail duplication.

YETI, from Zaleski et al. (19) applied Dynamo-style tracing to Java in order to achieve inlining, indirect jump elimination, and other optimizations. Their primary focus was on designing an interpreter that could easily be gradually re-engineered as a tracing VM.

Suganuma et al. (18) described region-based compilation (RBC), a relative of tracing. A region is an subprogram worth optimizing that can include subsets of any number of methods. Thus, the compiler has more flexibility and can potentially generate better code, but the profiling and compilation systems are correspondingly more complex.

Type specialization for dynamic languages. Dynamic language implementors have long recognized the importance of type specialization for performance. Most previous work has focused on methods instead of traces.

Chambers et. al (9) pioneered the idea of compiling multiple versions of a procedure specialized for the input types in the language Self. In one implementation, they generated a specialized method online each time a method was called with new input types. In another, they used an offline whole-program static analysis to infer input types and constant receiver types at call sites. Interestingly, the two techniques produced nearly the same performance.

Salib (17) designed a type inference algorithm for Python based on the Cartesian Product Algorithm and used the results to specialize on types and translate the program to C++.

McCloskey (14) has work in progress based on a language-independent type inference that is used to generate efficient C implementations of JavaScript and Python programs.

Native code generation by interpreters. The traditional interpreter design is a virtual machine that directly executes ASTs or machine-code-like bytecodes. Researchers have shown how to generate native code with nearly the same structure but better performance.

Call threading, also known as context threading (8), compiles methods by generating a native call instruction to an interpreter method for each interpreter bytecode. A call-return pair has been shown to be a potentially much more efficient dispatch mechanism than the indirect jumps used in standard bytecode interpreters.

Inline threading (15) copies chunks of interpreter native code which implement the required bytecodes into a native code cache, thus acting as a simple per-method JIT compiler that eliminates the dispatch overhead.

Neither call threading nor inline threading perform type specialization.

Apple's SquirrelFish Extreme (5) is a JavaScript implementation based on call threading with selective inline threading. Combined with efficient interpreter engineering, these threading techniques have given SFX excellent performance on the standard SunSpider benchmarks.

Google's V8 is a JavaScript implementation primarily based on inline threading, with call threading only for very complex operations.

9. Conclusions

This paper described how to run dynamic languages efficiently by recording hot traces and generating type-specialized native code. Our technique focuses on aggressively inlined loops, and for each loop, it generates a tree of native code traces representing the paths and value types through the loop observed at run time. We explained how to identify loop nesting relationships and generate nested traces in order to avoid excessive code duplication due to the many paths through a loop nest. We described our type specialization algorithm. We also described our trace compiler, which translates a trace from an intermediate representation to optimized native code in two linear passes.

Our experimental results show that in practice loops typically are entered with only a few different combinations of value types of variables. Thus, a small number of traces per loop is sufficient to run a program efficiently. Our experiments also show that on programs amenable to tracing, we achieve speedups of 2x to 20x.

10. Future Work

Work is underway in a number of areas to further improve the performance of our trace-based JavaScript compiler. We currently do not trace across recursive function calls, but plan to add the support for this capability in the near term. We are also exploring adoption of the existing work on tree recompilation in the context of the presented dynamic compiler in order to minimize JIT pause times and obtain the best of both worlds, fast tree stitching as well as the improved code quality due to tree recompilation.

We also plan on adding support for tracing across regular expression substitutions using lambda functions, function applications and expression evaluation using `eval`. All these language constructs are currently executed via interpretation, which limits our performance for applications that use those features.

Acknowledgments

Parts of this effort have been sponsored by the National Science Foundation under grants CNS-0615443 and CNS-0627747, as well as by the California MICRO Program and industrial sponsor Sun Microsystems under Project No. 07-127.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Any opinions, findings, and conclusions or recommendations expressed here are those of the author and should

not be interpreted as necessarily representing the official views, policies or endorsements, either expressed or implied, of the National Science foundation (NSF), any other agency of the U.S. Government, or any of the companies mentioned above.

References

[1] LuaJIT roadmap 2008 - http://lua-users.org/lists/lua-l/2008-02/msg00051.html.

[2] Mozilla — Firefox web browser and Thunderbird email client - http://www.mozilla.com.

[3] SPECJVM98 - http://www.spec.org/jvm98/.

[4] SpiderMonkey (JavaScript-C) Engine - http://www.mozilla.org/js/spidermonkey/.

[5] Surfin' Safari - Blog Archive - Announcing SquirrelFish Extreme - http://webkit.org/blog/214/introducing-squirrelfish-extreme/.

[6] A. Aho, R. Sethi, J. Ullman, and M. Lam. Compilers: Principles, techniques, and tools, 2006.

[7] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12. ACM Press, 2000.

[8] M. Berndl, B. Vitale, M. Zaleski, and A. Brown. Context Threading: a Flexible and Efficient Dispatch Technique for Virtual Machine Interpreters. In *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, pages 15–26, 2005.

[9] C. Chambers and D. Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed O bject-Oriented Programming Language. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 146–160. ACM New York, NY, USA, 1989.

[10] A. Gal. *Efficient Bytecode Verification and Compilation in a Virtual Machine Dissertation*. PhD thesis, University Of California, Irvine, 2006.

[11] A. Gal, C. W. Probst, and M. Franz. HotpathVM: An effective JIT compiler for resource-constrained devices. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 144–153. ACM Press, 2006.

[12] C. Garrett, J. Dean, D. Grove, and C. Chambers. Measurement and Application of Dynamic Receiver Class Distributions. 1994.

[13] J. Ha, M. R. Haghighat, S. Cong, and K. S. McKinley. A concurrent trace-based just-in-time compiler for javascript. Dept.of Computer Sciences, The University of Texas at Austin, TR-09-06, 2009.

[14] B. McCloskey. Personal communication.

[15] I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 291–300. ACM New York, NY, USA, 1998.

[16] A. Rigo. Representation-Based Just-In-time Specialization and the Psyco Prototype for Python. In *PEPM*, 2004.

[17] M. Salib. Starkiller: A Static Type Inferencer and Compiler for Python. In *Master's Thesis*, 2004.

[18] T. Suganuma, T. Yasue, and T. Nakatani. A Region-Based Compilation Technique for Dynamic Compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(1):134–174, 2006.

[19] M. Zaleski, A. D. Brown, and K. Stoodley. YETI: A graduallY Extensible Trace Interpreter. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 83–93. ACM Press, 2007.