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

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

Mozilla Corporation*
{gal,brendan,shaver,danderson,mrbkap,graydon,dmandelin,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. 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 highly 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 reliable 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 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: Initially, we execute JavaScript code using a traditional bytecode interpreter. Once a frequently executed (“hot”) code region has been identified, we record the instructions executed by the interpreter as it executes this region of code. We call this sequence of instructions a *trace*.

While traditional dynamic compilers tend to operate at the method level, our dynamic compiler operates at the granularity of individual loops. This design choice is based on the observation that programs tend to spend a disproportionate percentage of their runtime inside small portions of loopy code. Furthermore, even in dynamically typed languages such loopy code tends to operate on variables in a type-stable manner. (13). An integer loop counter, for example, is very likely to remain an integer across all iterations of the loop. We exploit these two observations in our approach, and type-specialize the code we record for such “hot” loops. The resulting machine code is only entered if the interpreter reaches the entry point of the trace *and* the types of all variables in the current scope match the *type map* of the trace which we captured at recording time.

PLDI ’08 June 15-20, 2009, Dublin, Ireland

A loop can have more than one frequently executed path through it. Alternative paths through a loop can occur as the result of explicit programming constructs such as `if` statements, but also implicitly as the result of *dynamic typing*, since in JavaScript the behavior of builtin operators varies based on the types of their arguments.

If a loop has multiple frequently executed paths, we form a tree of traces. Nested loops are represented as trace trees that contain other trace trees inside of them.

Our approach also offers a cheap and effective alternative to inter-procedural analysis to discover the flow of values and their types across functions. Trace trees always inline the code path executed inside invoked methods and embed the sequence of instructions directly in the trace. This makes tracing a very inexpensive yet effective tool to type specialize even complex function call-rich code.

Last but not least, traces can be compiled to efficient machine code in linear time since a trace is merely a linear sequence of instructions without control-flow merges. (11)

This paper makes the following contributions:

- Identification and recording of trees of frequently executed code traces that can span function calls. Complex control flow patterns are captured by nesting trees of traces.
- Type specialization of trace trees allowing the generation of efficient machine code for a dynamically typed programming language.
- An implementation inside the SpiderMonkey JavaScript interpreter with low overhead that achieves speedups of 2x-20x on many programs.

The remainder of this paper is organized as follows. Section 2 is a general overview of trace tree based compilation we use to capture and compile frequently executed code regions. In Section 3 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 4. Related work is discussed in Section 5. In Section 6 we evaluate our dynamic compiler based on a set of industry benchmarks. The paper ends with conclusions in Section 7 and an outlook on future work is presented in Section 8.

2. Dynamic Compilation with Trace Trees

Trace based compilation explores a different approach where the unit of compilation is a collection of code paths, or a *trace tree*. Frequently executed code paths are recorded and compiled using runtime profiling. A trace tree compiler generates code from these dynamically recorded *execution traces* and assembles them into a tree-like data structure that covers frequently executed (and thus optimization worthy) code paths originating at a program point, the *tree anchor*. Tree anchors are usually loop headers since they are the hottest program regions, and thus traces are loop cycles. (12) By capturing loop semantics within trace trees a trace tree compiler can benefit from many loop optimization techniques and produce very efficient code.

2.1 Anatomy of a Trace Tree

Figure 1 defines the various components of a trace tree. A *trace* is a consecutive sequence of instructions recorded during the execution of a program. A *trace tree* is a collection of connected traces in a general program region. Traces originate at program locations known as the *trace anchors* and loop back to the *tree anchor* which is the root of the trace tree. The first trace in a tree of traces is known as the *trunk trace*, and its trace anchor is referred to as the *tree anchor*. Additional traces branch of other traces and are called *branch traces*. Traces use *guard* instructions to constrain control flow to the recorded path. A trace based VM discovers potential tree anchors by profiling program execution.

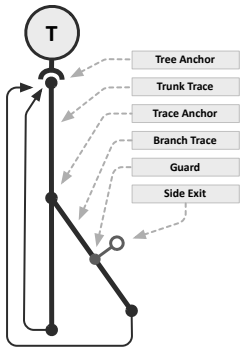

Figure 1. A tree with two traces, a trunk trace and one branch trace. The trunk trace contains a guard to which a branch trace was attached. The branch trace contain a guard that may fail and trigger a side exit. Both the trunk and the branch trace loop back to the tree anchor, which is the beginning of the trace tree.

2.2 Guards

All explicit control flow instructions are guarded. For example, if the conditional branch opcode `bl` (branch less than) instruction is taken during the recording of the trace, the guard instruction *guard-less-than* is inserted in the recorded trace to ensure that the branch is always taken during the execution of the trace tree. Implicit control flow instructions must also be guarded. For example an instruction whose behavior is determined by the concrete runtime types of its operands necessitates guards to check for the existence of those types during the execution of the trace tree. Anything that may divert control flow from the original recorded path must be guarded. When a guard fails the control flow has diverged from the expected path. Execution of the trace tree must be terminated and control given back back to the interpreter, which is capable of executing the divergent control flow.

This *side exiting* process is sometimes also referred to as de-optimization, where an optimized method switches back to an un-optimized version.

2.3 Discovering and Growing Trace Trees

We use backward branch profiling to discover tree anchors. We keep track of the number of times an instruction is executed as a result of a backward branch. If this number exceeds a certain threshold, the backward branch target is considered a potential tree anchor.

Once a tree anchor is identified the VM uses the trace recorder to record traces. If the recorded trace leads back to the beginning of the trace tree, the trace is kept and a trace tree is planted at the discovered tree anchor. During the recording process instructions are translated into a trace-flavored Static Single Assignment (TSSA) form proposed by Gal et al. (11). Since traces are a linear sequence of instructions, we can transform them into TSSA form without having to worry about ϕ-node placement (10).

During the recording process, the trace recorder inserts *guard* instructions that ensure control flow stays on the recorded path during the execution of a trace tree. If a side exit occurs along such a guard, the trace recorder may attempt to add a *branch*

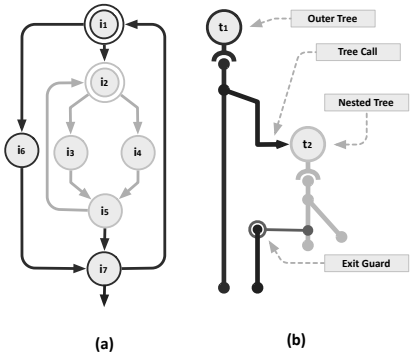

Figure 2. Control flow graph of a nested loop with an if statement inside the inner most loop (a). An inner tree captures the inner loop, and is nested inside an outer tree which "calls" the inner tree. The inner tree returns to the outer tree once it exits along its loop condition guard (b).

trace to the trace tree by recording the divergent control flow. Whether we should extend the tree from the side exit point or not is decided based on the syntactic structure of the input program. Since the input of our VM is JavaScript source code, we can easily generate and maintain hints regarding each bytecode-level branch instructions whether it is a conditional branch within a loop, or whether it is part of the loop condition check, for example. For the latter case we do not want to extend the tree, since such a trace is not part of the loop.

2.4 Trace Tree Compilation

Figure 2 describes the general trace tree compilation principles introduced by Gal et al. (12) applied to a trace tree through a nested loop with an if statement in the inner most loop. The tree anchor at instruction i_2 is detected sooner than the tree anchor at instruction i_1 since it is executed more frequently. For this reason, the trace tree covering the nested loop is anchored at instruction i_2. Once the tree anchor at instruction i_2 is detected, a trace is recorded starting at that location. The first recorded trace is a cycle 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.

In the general trace tree compilation approach proposed by Gal et al. traces allowed to exit loop scopes. For example, using this approach a second possible trace inside tree t_2 could be $\{i_2, i_3, i_5, i_7, i_1, i_6, i_7, i_1, \alpha\}$, branching off instruction i_5 and looping once around the outer loop until finally reaching the tree anchor at instruction i_2. Eventually enough traces can be recorded to cover the complete control flow in Figure 2.

In our implementation we use the syntactic structure of the input program to avoid such "outerlining". Outerlining can lead to excessive tail duplication by unrolling the outer loop several times inside the inner loop. The trace recorder is not allowed to record traces leaving loop scopes. For example, the nested loop control flow in Figure 2 cannot be captured with one trace tree. Instead, we trace the control flow of the inner loop independently of the outer loop. Once the outer loop becomes hot enough and is discovered as a tree anchor, a tree is constructed for it that nests the tree covering the inner loop. When the outer tree reaches the anchor point of the inner tree, it *calls* the inner tree instead of attempting to re-record its interior. We call this technique *tree nesting*.

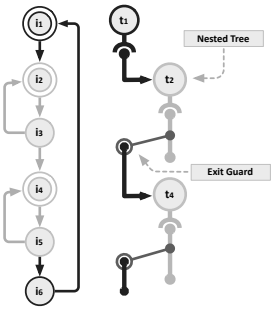

Figure 3. Control flow graph of a loop with two nested loops (left) and its nested trace tree configuration (right). The outer tree calls the two inner nested trace trees and places guards at their side exit locations.

2.5 Nested Trace Trees

Nested trace trees build on previous trace tree compilation research by Gal et al. (12) to provide a way to capture more complex control flow and reduce some of the negative effects of tail duplication. Conceptually nested trace trees are a generalization of trace trees. As defined earlier, a trace is a sequence of recorded instructions. Instructions that affect control flow, or whose internal control flow is determined by external variables, necessitate guards to ensure that at runtime their behavior is exactly as observed during recording. For example, dynamic invocation instructions must guard on the receiver type in order to ensure that at runtime, control flow stays on the recorded path. This is because dynamic invocation instructions can transfer control flow to any number of program locations. Much in the same way that dynamic invocation instructions transfer control flow to multiple program locations, trace trees also do the same. A trace tree can be thought of as a single complex instruction that has one entry point and multiple exits, trace trees are entered through their tree anchors and exit through any side exit. This is the key observation that allows for the construction of nested trace trees.

If trace trees are nothing more than mere instructions then they can be recored as part of other traces. During the recording of a trace, if another trace tree (nested tree) is encountered, the trace recorder executes the nested tree and resumes the recording of the original trace at the exit location of the nested tree. Since the nested tree can exit at any of its side exits, a guard is necessary to guard on the exit taken from the nested tree. This guard checks at runtime that the nested trace tree always exits at the same location as it did when it was originally recored. If this guard fails, an additional branch trace can be attached to the outer tree that captures the path leading from the new nested tree exit.

Figure 3 shows the control flow graph of a loop with two nested loops and its nested trace tree representation. This control flow graph is generally not possible to capture without using nested trace trees. In this example, instruction i_2 is detected as a tree anchor first. This will lead to the construction of trace tree t_2 anchored at this location. Once a side exit is taken from t_2 at i_3 the trace recorder will attempt to record a trace leading back to i_2. The trace recorder will end up unrolling every loop iteration in the second nested loop before reaching i_2 again, recording the trace $\{i_4, i_5, ..., i_4, i_5, i_6, i_1, i_2, \alpha\}$. The trace recorder generally disallows loop unrolling in order to reduce trace sizes and instead aborts tracing all together. A similar problem occurs when trace tree t_4 is built. Once a side exit is taken from t_4 at i_5 the trace recorder will again attempt to record a trace leading back its tree anchor, i_4, which will cause the unrolling of the first loop. This pathological

behavior prevents the construction of trace trees capable of capturing the entire the control flow shown in Figure 3 if tree nesting is not allowed.

In the Figure 3 example, instruction i_1 is the last tree anchor to be detected. A trace originating at i_1 leading back to itself would have to unroll both inner nested loops and would not be allowed by the trace recorder. If tree nesting is allowed, the trace recorder will notice that a tree t_2 exists at instruction i_2 and will nest the tree. The same will happen at instruction t_4, resulting in the trace $\{i_1, t_2, t_4, \alpha\}$. Nested trace trees can capture the entire control flow shown in Figure 3.

Nested trace trees preserve all the properties of trace trees. Optimization algorithms that operate on trace trees can operate on nested trace trees without any modifications. Nested trace trees can also be used to capture recursion, another example of control flow that cannot be captured with trace trees. The only differences between a trace for a recursion code region is the fact that the stack is not balanced at the loop edge, and either grows (down recursion), or shrinks (up recursion).

3. Speculative Type Specialization along Traces

When compiling traces, we type specialize all values in the current scope. In the virtual machine interpreter model this includes values currently on the interpreter stack as well as in local variables in the current interpreter frame. The types of these values are captured in a "type map", which is stored in the trace along with the intermediate representation code. The interpreter only enters a trace tree when the types of all values in the current scope match exactly the types of a tree that was recorded for the current program counter location.

As the interpreter transitions into the native code all values are unboxed and stored on the native machine stack. Once the native code starts executing it operates exclusively on these unboxed types.

At the loop edge of a trace we want to loop back to the loop entry point without having to re-box and re-unbox values, so for a trace to "close" (loop back to itself), the types of all values at the end of the loop must match the entry type map.

For the fundamental JavaScript types we can precisely observe the flow of values and types at recording time since the resulting type of JavaScript operations is either known (i.e. adding two numbers always produces a number,) or it is guarded (a native function call is expected to produce the same type at runtime as at recording time, otherwise we have to take a side exit and attach another trace.) In addition to specializing for these concretely known types, we also perform speculative type specialization to generate optimized code for loops that operate on numbers in the 32-bit integer domain.

For this, when we start the recording of a loop we initially speculate that all numeric values that are whole numbers will actually remain integers along the trace. After completing the trace we verify this speculation. If the value is still an integer number at the end of the trace, we close the loop and indicate in the entry type map that we only accept integer values for this variable. The interpreter will only execute this trace if at runtime the value is an integer number again. If the speculation fails, i.e. because we started out with an integer number but the variable now contains a value that is not integral, we throw out the currently recorded trace and re-record it. To ensure that we don't incorrectly speculate on this variable again, we inform an *oracle* about this failed speculation attempt. This oracle is consulted every time we are about to speculate on integral values, and if prior adverse information is available for a variable we do not attempt to demote it to an integer and instead compile floating point code.

In dynamic languages the set of types expected upon entry to a loop can differ from one run to another. To address this, we record

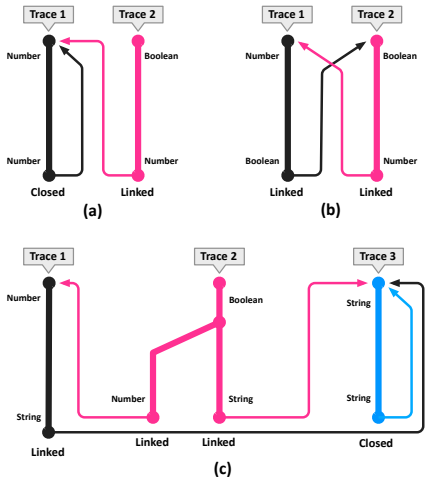

Figure 4. We handle type-unstable loops by allowing traces to compile that cannot loop back to themselves due to a type mismatch. As such traces accumulate, we attempt to connect their loop edges to form groups of trace trees that can execute without having to side-exit to the interpreter to cover odd 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.

multiple trees per loop, each specialized for the set of types the loop expects. It is problematic, however, if the set of types expected on entry is not compatible with the set of types at the loop edge. For example, concatenation of a number and string could result in a type change during the loop. This is particularly problematic in JavaScript where such implicit coercions are more prevalent.

Loops with such type instability cannot be closed as no conversions between the mismatching types were observed by the trace recorder. Rather than discard these traces, we simply leave the loop edge unclosed, such that running the trace will result in only one iteration before exiting. These unstable edges can happen along stable trees as well, since a branch extended from a type-stable tree could change the expected set of types. Every time new traces are added, we scan the existing set of traces and join compatible trace tails.

The simplest case is when an unstable edge is compiled, and there exists a tree for the same loop whose entry types are identical to the unstable edge's exit types. The unstable trace can be linked directly to the stable trace, closing the loop. This frequently happens in case of outer trees nesting an inner tree that has an undefined variable during the first iteration, which is then set to an actual value with a true type within the loop body.

Figure 4 shows a number examples of traces being linked together to cover type-unstable loops. The first case is a loop where a value is initially observed as a number both at trace entry as well as at the loop edge. Since we are more likely to catch a loop "in flight" when it is already type-stable, such type-stable loop traces are usually discovered first. Later on, i.e. when we try to nest this loop into another trace, we might encounter a situation where the value is not initialized before entering the first iteration of the loop. In this case the value appears to be of boolean type, since we use the boolean value space to represent the `Undefined` value in JavaScript. Hence we compile a trace fragment that can be entered with an undefined

value in the variable, and it connects directly to the existing loop trace once the value is set and its type changes to number.

The more involved case is when unstable edges are compiled, but there exists no type-compatible tree for the same loop. It is possible that such a compatible tree will be compiled later, so information about all unstable edges are cached. Whenever a trace is compiled, the list of unstable edges is re-evaluated for any possible connections that can be made. This approach can effectively deal with a number of type-unstable loop scenarios.

For example, the type of a variable inside a loop sometimes toggles between two different types on each iteration, resulting in two traces that are mutually unstable. However, as a whole they become stable if we connect their respective edges (Figure 4).

4. 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 used in Mozilla's Firefox open-source web browser (2), which is used by more than 200 million users world-wide.

SpiderMonkey is a high performance, open-source Virtual Machine interpreter for JavaScript, implemented in C/C++. Its input is JavaScript source which is parsed and translated to a bytecode representation which is then executed by the SpiderMonkey's VM interpreter.

SpiderMonkey's bytecode is untyped and the VM uses runtime type tagging of data values to store values along with their types. The fundamental value type of the VM is a signed machine word (*jsval*) that contains either a signed integer value (if the low bit is set), or a type-tagged pointer or boolean value (if the low bit is clear). Tagged pointers all refer to 8-byte-aligned things in the GC heap.

Objects consist of a possibly shared structural description, called the map or scope; and unshared property values in a vector, called the slots. Object properties are associated with nonnegative integers stored in *jsval*'s, or with atoms (unique string descriptors) if named by an identifier or a non-integral index expression.

The Garbage Collector (GC) in SpiderMonkey is a mark-and-sweep, non-conservative (exact) collector. It can allocate only fixed-sized things and it is used to hold JS object and string descriptors (but not property lists or string bytes), and double-precision floating point numbers.

4.1 Lightweight Trace Recording

SpiderMonkey uses a highly tuned threaded bytecode interpreter to execute JavaScript programs. We use SpiderMonkey's original interpreter for the initial profiling of loops and to record traces once a hot loop crosses the recording threshold. Since we continue to use the interpreter for non-loopy code or for code that we were not able to trace, i.e. because we don't support certain instructions along that trace or because the code is too type unstable, not impacting the performance of the interpreter was a key design principle for our system.

While profiling code, the only interaction between the dynamic compiler and the interpreter is a callback from the interpreter to the dynamic compiler for every taken backwards branch. Similar to Dynamo (7) we use frequently taken backward branches as an indication that the destination of the jump is a loop header, and hence a potential starting point for a trace tree.

Updating the threshold counter at every loop edge is surprisingly expensive (7.8% on average across our set of benchmark applications). The main cost is looking up the fragment structure for the bytecode location in the fragment cache, since the fragment structure contains the threshold counter. We are unable to store the counter directly in the bytecode sequence due to the fact that SpiderMonkey is a multi-threaded interpreter and multiple concurrent threads might execute the same bytecode.

The overhead of the fragment lookup is particularly pronounced for tight loops such as simple counting loops. Complex loops on the other hand spend significantly more time in the loop body, and the fragment lookup overhead falls below 1%. In practice, the fragment lookup along loop edges is not critical for overall system performance. We use a fairly low loop threshold value ($n = 2$), which means that loops are compiled to native code almost instantly and the interpreter will no longer incur a performance overhead for the loop edge since the code is executed natively.

When recording a trace, we modify the dispatch table of the threaded interpreter to jump to a per-bytecode recording stub instead of directly dispatching to the implementation of the bytecode. This means that recording stubs see the state of the VM prior to executing the effect of the bytecode.

Each recording stub inspects the VM stack and local variables to identify the input values and their types, and generates a sequence of machine-independent low-level intermediate instructions (LIR) into a IR code buffer (LIR buffer). For most bytecode instructions the recorder can predict the precise effects of the instruction based on the types of the input arguments. A few selected instruction, on the other hand, require us to actually execute the instruction first before we can observe some of its precise effects. The `String.charCodeAt` function, for example, returns the character code if the provided index is less than the total length of the string, and *NaN* (Not a Number) otherwise. For these bytecode instructions we use a post execution hook to communicate the result of the operation to the recorder. The overhead of these hooks is small, especially since such instructions tend to be expensive themselves, dwarfing the overhead of the hook when it is not taken because we are currently not recording.

Once the recording stub has performed all necessary recording steps it then dispatches to the actual implementation of the bytecode which will execute the bytecode. Recording stubs are only activated when we are recording a trace, and thus the recorder does not impact interpreter performance when disconnected.

4.2 Intermediate Representation and Trace Compilation

For each recorded bytecode instruction the trace recorder emits a sequence of LIR instructions in the LIR buffer of the currently active trace tree. The LIR used by our compiler is encoded in a trace-flavored Static Single Assignment (TSSA) form proposed by Gal et al. (11). The `LIR_add` instruction, for example, takes two operands; these operands must both be references to prior instructions on the same trace. The `LIR_add` instruction itself produces a value that may be used as input to other instructions.

The trace compiler architecture uses "filters" to customize and optimize LIR. The recorder emits LIR instructions into the first filter in a pipeline of filters, and each filter can inspect, modify, emit or omit LIR instruction that pass through it. Each filter independently implements an optimization step, and hence neatly encapsulates all the code and data associated with that particular compilation step. The constant folding filter, for example, watch for arithmetic opcodes with two constant operands, and inserts a folded immediate value instead. A sink filter at the end of the pipeline encodes the remaining sequence of LIR instructions into the LIR buffer.

Once the entire trace has been recorded, the LIR buffer is read back backwards (starting with the last instruction, working all the way forward to the first instruction in the trace) and LIR instructions are sent through a second filter pipeline, which contains filters that prefer to see the code in use-def order (whereas filters in the forward pipeline see all instructions in def-use order). Our code generator contains filters for common subexpression elimination, expression/strength reduction, load/store and dead code elimina-

tion as well as soft-float filter, which can translate floating-point LIR instructions into equivalent calls to helper functions for processors that do not have a floating point unit.

The actual machine code generation occurs in the last filter in the backwards pipeline, which means that the loop edge jump of the trace is generated first. Instructions that survived the dead code elimination filter are emitted and registers are assigned to these live instructions using a linear scan allocation. Register assignments are recorded in a reservation table, which either indicates the physical register that holds the value of an instruction, or an offset relative to the machine stack in case the value was spilled. As the filter emits code backwards, when a referenced instruction is reached it is responsible for placing its result in the storage location indicated by the reservation table.

If there are no registers available of the requested register type, the assembler is forced to evict a suitable reservation. The least recently reserved instruction (furthest use away) is chosen as a simple heuristic. Once the victim is selected, its reservation is marked as owning a machine stack offset in addition to its register. Machine code is then immediately emitted to restore this register from the stack. Afterwards the new instruction is assigned the victim's register, and code generation can continue.

Value-producing instructions are responsible for freeing their reservations. If a freed reservation has assignments for both a register and stack offset, the spilling logic is invoked to complement a previous restore. If only a stack offset is assigned, a register can be temporarily assigned for the sole purpose of spilling. If an instruction needs to be spilled (meaning it was an evicted victim earlier), machine code is generated to store the assigned register into the assigned stack location. The register and stack location are then marked as free for re-use.

A consequence of this algorithm is that the result-producing register can be re-used as one input register, which is of particular importance for 2-address instruction sets like Intel x86.

4.3 Trace Stitching

Trace trees start out as a single trace that was recorded as an iteration through a loop. At suitable side exits new traces can be attached to this initial trace. In previous work it was shown that trace trees can be compiled by compiling each individual trace but carrying over the register allocating state, hence producing a globally optimized compilation result (11). We have explored this approach but we found that it was suboptimal for trees with many traces since we would have to recompile the entire tree every time we attach another trace. While compiling the tree is a linear effort, having to recompile the tree for each attached trace would make the overall compilation effort quadratic.

Instead, we use trace stitching to assemble trace trees in the code cache. Once the first trace in a tree has been compiled, it is executed. When a side exit occurs, the trace is left and a secondary trace is recorded if appropriate. In the trace stitching model, when transitioning from the primary trace to the secondary trace, the primary trace writes back its state into memory as if it would take a side exit and when we generate code for the secondary trace we start out by emitting code to fetch the state afresh from memory. This approach of course has the downside that at every conditional branch in the original code we have to write the value of all live expressions into memory and re-read it "at the other side" if we switch to a different trace. On the upside, this approach allows building complex trace trees quickly with minimal compilation overhead.

While tree stitching is superior when low compilation latency is key, background threads are ideally suited to recompile the tree after a quick trace attachment through stitching. Multicore systems offer the software designers significantly more compute power to exploit. While dual-core mobile systems have gained widespread acceptance, multicore handheld systems are not too far from reality. While the compiler used in this work does not currently perform background trace tree compilation, in a closely related project (14) background recompilation yielded speedups of up to 1.25x for compilation intensive benchmarks that require compiling a large number of trace.

4.4 Garbage Collection and Preemption

Javascript, like many dynamic languages, provides certain automatic runtime services outside the control of the programmer, such as garbage collection and timed interrupt execution.

Such runtime services are typically performed, in interpreters, by polling conditions at regular points during interpretation, for example as part of the memory allocation routine, or part of an opcode dispatch loop. In contrast, integrating these services with arbitrary runtime generated machine code is a potential challenge. Our approach is to defer such services entirely while a trace is running. When a service should be performed and a trace is running, the trace is induced to exit temporarily. Once the trace has safely exited, and execution has returned to the interpreter, the service (i.e. garbage collection) runs and the trace is resumed. This simplifies interactions between the code in the trace and the runtime services, but raises the risk of deferring such services for too long.

In the case of garbage collection, new object allocation simply fails with a code value that causes a premature exit, when the allocator reaches its memory threshold. Therefore there is never a delay between the allocator refusing an allocation and a trace exiting, at which point garbage collection can run.

The case of timed interruption of a running trace is more subtle, as the time of interruption may fall at any moment of trace execution. While a small gap between the intended time of interruption and the actual time of trace exit is acceptable, we have to ensure that we can forcefully preempt running traces once an interrupt event has occurred (even if the underlying loop would prefer to keep iterating if not interrupted).

Since we compile only *cyclic* code paths into traces, the final native instruction in each trace is an unconditional jump back to the beginning of the trace. Such instructions are referred to as "loop jumps". Loop jumps are a naturally occurring "boundary" in each trace, and therefore are a good candidate point to check for an interruption signal. When a watchdog timer thread determines that an interrupt should occur, it locks the trace storage pool and rewrites all loop jumps to jump into a special exit path. This has no immediate effect on the thread executing the trace, but when it next encounters a (rewritten) loop jump at the end of a trace iteration, it will exit. Upon exit, the thread rewrites all the loop jumps back to their initial configuration, repairing the trace, and then services the interrupt. This approach incurs no performance overhead when interruption does not occur, which is the most common case.

4.5 Foreign Function Interface

The JavaScript engine has an existing Foreign Function Interface (FFI), which can be used to access the Document Object Model (DOM), which is implemented in C++. To maximize time spent in trace code, it is necessary to record across such calls into native C++ code and remain on trace. Two optimizations our trace compiler performs complicate this matter: As a trace executes, stores to global JavaScript variables are not immediately written to the global object; and stack values are not immediately stored on the JavaScript stack. In both cases this is done to avoid the overhead of boxing the values. The last-stored values of these slots are boxed and stored in the appropriate places when a trace exits. However, if a native function is called from trace, we must ensure that it does not read or write the boxed values.

One way to avoid this problem is to box all global properties and stack frames before entering native code, and unbox them again when returning to trace code. However, most natives do not examine the JavaScript stack or examine globals, so we chose to do this lazily, on demand. We modified the engine to ensure that any access to the JavaScript stack and global objects is done through a single function. When this function is called while a trace is executing, it immediately flushes the native stack and global frame to their interpreter-ready counterparts, and sets a flag indicating that the trace cannot proceed. The native function runs to completion, but when control returns to the trace, it immediately side-exits.

To ensure that we intercept all stack and global accesses, we performed a static analysis of the SpiderMonkey engine using the Treehydra C/C++ static analysis framework (6).

The design of the SpiderMonkey's FFI predates the trace compiler. Native functions expect the JavaScript arguments to be boxed and stored in an array in memory rather than in registers. This layout is convenient for the interpreter but inconvenient for native code, which must examine and unbox each argument.

This also means the trace recorder cannot fully take advantage of the argument types, which are known at record time. When calling a native, the trace must box all the JavaScript arguments, store them in an array, then push the native function's arguments to the C stack. The native then typically unboxes all the arguments again right away. When the native is ready to return, it boxes the result; the trace immediately unboxes that result before proceeding.

The technique described above imposes additional overhead on the function call. Before calling a native, the trace must store a pointer to type information about the native stack and global frames. This is used only if the builtin must flush the stack and globals. After the call, an additional guard is needed to check that the builtin did not leave trace.

This approach allows much more code to remain on trace without imposing a burden on application code. We have also added new FFI features to allow applications to provide zero or more type-specialized implementations, with type information, for each native. The tracer recorder selects the appropriate implementation at record time, using its knowledge of the argument types to eliminate the boxing and unboxing.

4.6 Correctness and Testing

We used a wide range of standalone and in-browser tests to ensure the correctness of our trace compiler. In addition to the existing test suits of the SpiderMonkey JavaScript VM and the test infrastructure available for Firefox, we also used Fuzz testing as an effective method to identify errors in the code generator.

Fuzz testing is usually considered to be a security technique: Find a crash–maybe it's exploitable. During the development of our trace compiler, we found that fuzzing made it much easier to develop the compiler rapidly. For this we used Mozilla's jsfunfuzz tool, which generates random JavaScript code, often nesting language constructs in fairly complicated ways.

Because jsfunfuzz was initially designed to test an interpreter, it did not make heavy use of loops and rarely triggered the trace recorder. But with small additions, it was able to find dozens of bugs in the trace recorder implementation. Adding simple loops allowed it to test the basic tracing infrastructure by making some fuzz-generated code "hot" enough to trace. Generating loops that iterate over mixed-type arrays, it was able to find bugs in the tracer code designed to handle type-unstable loops. Finally, intentional branch-instability helped find more subtle bugs in the way trace fragments interact with each other.

We found that it was significantly easier to diagnose and correct compiler regressions based on small code snippets generated by the fuzzer than debugging our compiler within a web browser running actual web content, which often is highly complex and also frequently uses code obfuscation methods.

5. 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 (17) 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, Pysco transforms loops to mutual recursion before running and traces all operations.

Pal'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. (12) in the context of Java, a statically typed language. Their trace trees actually inlined parts of outer loops within the inner loops (because inner loops become hot first), leading to much greater tail duplication.

YETI, from Zaleski et al. (20) 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-eningeered as a tracing VM.

Suganuma et al. (19) 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 are 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 (18) 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 (15) 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 gen-

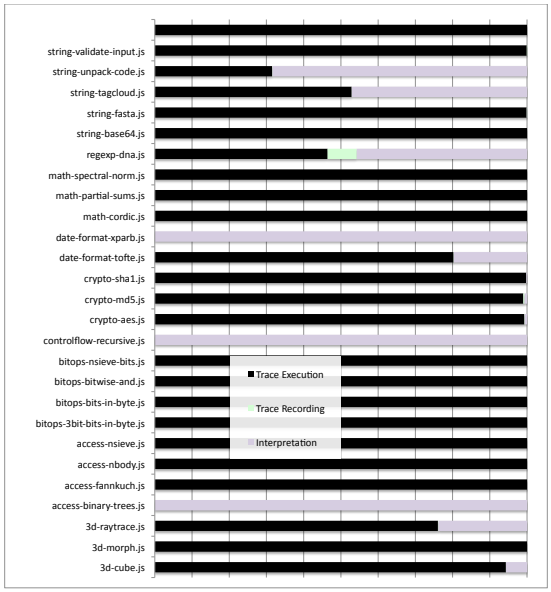

Figure 6. Number of opcodes executed as native code (on trace) and by the interpreter during recording as well as pure interpretation. On average, we execute recorded instructions 3,500 times in native code form. In addition to the two recursive benchmarks we also don't generate any native code for `date-format-xparb` and `string-unpack-code`, which perform a regular expression replacement using a lambda function, which we currently cannot record.

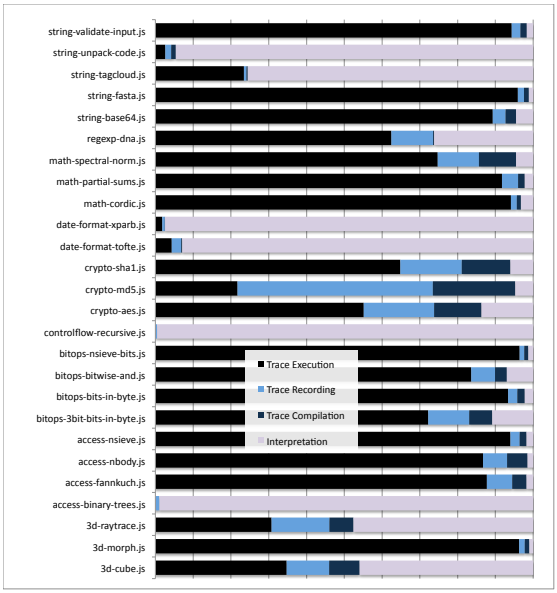

Figure 7. Time spent interpreting, recording, compiling and actual native code execution. While Figure 6 shows that we execute a large percentage of instructions natively, the much slower interpreter still makes up a significant fraction of the execution time for certain benchmark programs. In case of `crypto-md5`, for example, we spend 70% of the execution time recording and compiling a trace. Once we have obtain the native trace, we very quickly finish the remaining iterations. The overall speedup of this test appears to be low, however, this is in part due to the fact that the benchmark program was designed at a time when all JS virtual machines used interpretation only. Thus the problem size was artificially reduced to a point where its almost instantly computed by the native code we generate, allowing for very little time to recoup compilation code. A similar effect can be observed for the other crypto benchmarks. Run with our system the average execution time for SunSpider benchmark programs is a mere 26ms, which highlights the importance of fast and cheap compilation.

erate 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 (16) copies chunks of code from the interpreter into a native code buffer, thus acting as a simple per-method JIT compiler. The resulting dispatch is free.

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 JavaScript implementation primarily based on inline threading, with call threading only for very complex operations.

6. 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.

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 5 different configurations for comparison: (a) SpiderMonkey, the baseline interpreter, (b) our tracing VM, with tracing turned off, to measure the cost of incorporating tracing features, (c) the tracing VM, (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 5 shows the relative speedups achieved by tracing, SFX, and V8 against the baseline (SpiderMonkey). We do not show the tracing VM with tracing turned off in the graph because it performs the same as the baseline. (The overall speedup is 1.02x, and the per-program speedups vary between 0.91x and 1.06x. There is no reason to expect the tracing infrastructure to make programs run faster without tracing, so we conclude the differences are noise.)

For integer-heavy benchmarks we achieve the best speedups, with up to 25x over the interpreter in case of `bitops-bitwise-and`.

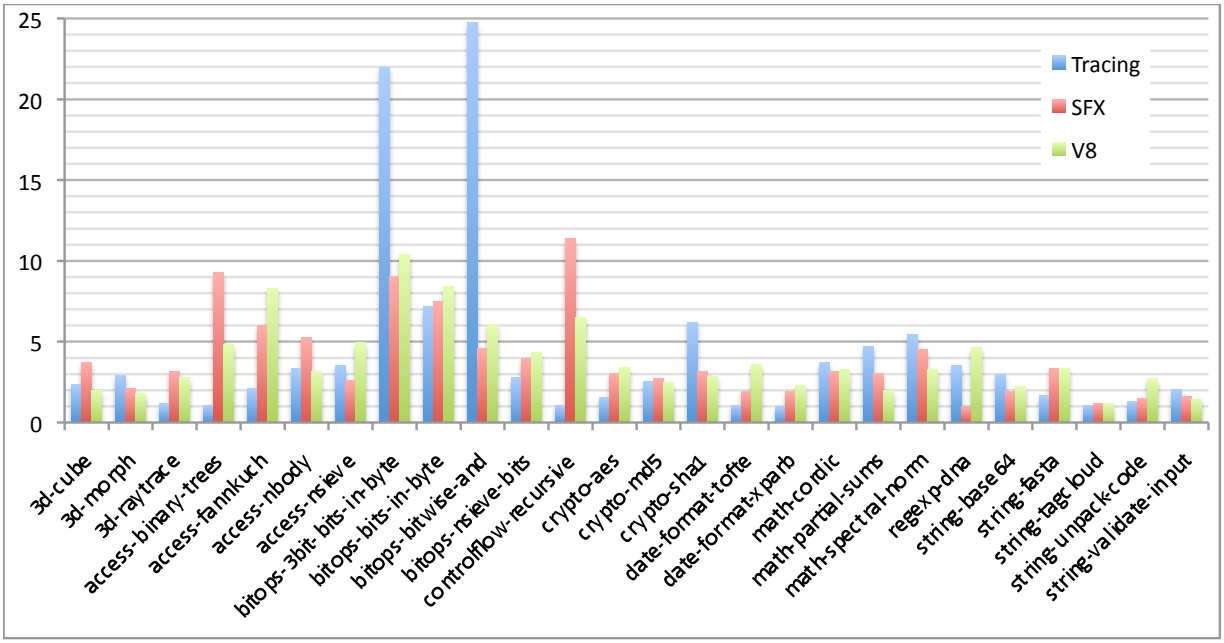

Figure 5. Speedup vs. a baseline JavaScript interpreter (SpiderMonkey) for our trace-based JIT compiler, Apple's SquirrelFish Extreme inline threading interpreter and Google's V8 JS compiler. Our system generates particularly efficient code for programs that benefit most from type specialization, which 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 Figure 6 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 7 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.

The data in Figure 6 and Figure 7 explain the high variance in the overall speedups in Figure 5. For example, on the crypto benchmarks, we obtain a modest speedup of 2x-3x even though the crypto benchmarks are integer-heavy, and other integer-heavy benchmarks see much greater speedups. The operation profile in Figure 6 shows that we do actually compile all relevant code and execute most instructions natively. But the timing profile in Figure 7 reveals that in all three benchmarks much of the time is spent recording and compiling traces. These benchmarks all have complex loop bodies that run for only a few iterations, giving a relatively high compilation cost with less benefit.

Note that some of the benchmarks use regular expressions extensively, which are not affected by general VM technology. For example, `regexp-dna` spends 99% of its time running regular expression matches. The speedups achieved by our implementation and SFX on `regexp-dna` are actually from using an improved regular expression implementation that compiles to native code. `crypto-aes` also uses a regular expression replace in its hottest loop with a callback function (that returns the replacement string). Our implementation does not trace these callbacks, and thus we attain less of a speedup.

Two programs, `access-binary-tree` and `controlflow-recursive`, spend a lot of time calling recursive functions. Although trace trees should be able to represent recursion, our implementation does not, so we do not speed up these benchmarks.

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.

	Anchors	Trees	Traces	Trees/Anchor	Traces/Tree	Traces/Anchor	Trees/Anchor	Speedup
3d-cube	14	26	29	1.85	1.11	2.07	1.85	2.19
3d-morph	5	7	10	1.4	1.42	2	1.4	2.72
3d-raytrace	14	22	81	1.57	3.68	5.78	1.57	1.11
access-binary-trees	3	3	6	1	2	2	1	1.02
access-fannkuch	10	32	81	3.2	2.53	8.1	3.2	2.17
access-nbody	8	14	21	1.75	1.5	2.62	1.75	3.46
access-nsieve	4	5	7	1.25	1.4	1.75	1.25	3.42
bitops-3bit-bits-in-byte	2	2	2	1	1	1	1	20.85
bitops-bits-in-byte	3	3	5	1	1.66	1.66	1	6.65
bitops-bitwise-and	1	1	1	1	1	1	1	21.21
bitops-nsieve-bits	3	3	5	1	1.66	1.66	1	2.72
controlflow-recursive	1	1	1	1	1	1	1	0.99
crypto-aes	33	67	96	2.03	1.43	2.90	2.03	1.50
crypto-md5	4	3	5	0.75	1.66	1.25	0.75	1.94
crypto-sha1	5	4	10	0.8	2.5	2	0.8	5.68
date-format-tofte	4	4	15	1	3.75	3.75	1	0.99
date-format-xparb	3	4	9	1.33	2.25	3	1.33	0.94
math-cordic	2	3	6	1.5	2	3	1.5	4.87
math-partial-sums	2	3	7	1.5	2.33	3.5	1.5	5.07
math-spectral-norm	8	19	21	2.37	1.10	2.62	2.37	5.58
regexp-dna	2	2	2	1	1	1	1	3.54
string-base64	3	3	7	1	2.33	2.33	1	2.69
string-fasta	5	11	19	2.2	1.72	3.8	2.2	1.52
string-tagcloud	3	7	14	2.33	2	4.66	2.33	1.01
string-unpack-code	9	9	35	1	3.88	3.88	1	1.22
string-validate-input	5	8	15	1.6	1.875	3	1.6	1.83

Figure 8. Detailed trace recording statistics for the SunSpider benchmark set.

7. 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.

8. 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.

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] Treehydra - Mozilla. https://developer.mozilla.org/en/Treehydra.

[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] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

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

[12] 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.

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

[14] J. Ha, M. R. Haghighat, S. Cong, and K. S. McKinley. A concurrent javascript JIT compiler. In *International Conference on Compiler Construction 2009, Submitted*, 2009.

[15] B. McCloskey. Personal communication.

[16] 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.

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

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

[19] 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.

[20] 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.