Tracing for Web 3.0

Trace Compilation for the Next Generation Web Applications

Mason Chang[†*], Edwin Smith[*], Rick Reitmaier[*], Michael Bebenita[†], Andreas Gal[†§], Christian Wimmer[†], Brendan Eich[§], Michael Franz[†]

[*]Adobe Corporation [†]University of California, Irvine [§]Mozilla Corporation

{edwsmith,rreitmai}@adobe.com, {changm,mbebenit,gal,cwimmer,franz}@uci.edu, brendan@mozilla.org

Abstract

Today's web applications are pushing the limits of modern web browsers. The emergence of the browser as the platform of choice for rich client-side applications has shifted the use of in-browser JavaScript from small scripting programs to large computationally intensive application logic. For many web applications, JavaScript performance has become one of the bottlenecks preventing the development of even more interactive client side applications. While traditional just-in-time compilation is successful for statically typed virtual machine based languages like Java, compiling JavaScript turns out to be a challenging task. Many JavaScript programs and scripts are short-lived, and users expect a responsive browser during page loading. This leaves little time for compilation of JavaScript to generate machine code.

We present a trace-based just-in-time compiler for JavaScript that uses run-time profiling to identify frequently executed code paths, which are compiled to executable machine code. Our approach increases execution performance by up to 116% by decomposing complex JavaScript instructions into a simple Forth-based representation, and then recording the actually executed code path through this low-level IR. Giving developers more computational horsepower enables a new generation of innovative web applications.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors - Compilers

General Terms Design, Languages, Performance

Keywords JavaScript, Trace Trees, Tracing, Forth, Type Specialization, Tamarin, Dynamic Compilation, Dynamically Typed Languages

1. Introduction

The web has become an ubiquitous platform for content-rich applications. In the early days of "Web 1.0", most web sites were fairly static and provided little user interaction. The "Web 2.0" revolution redefined the browser as a vehicle for delivering richer media content and interactivity through a fusion of existing technologies, most notably Asynchronous JavaScript and XML (AJAX). As web applications become more sophisticated, they rely more and more on JavaScript to do the heavy lifting rather than just using it for trivial scripting tasks. As a result, JavaScript performance has become a real bottleneck to the web experience.

The flexibility and rapid prototyping abilities of dynamically typed languages such as JavaScript are the main factors contributing to their popularity on the web. Today's web applications are built using an array of largely independent technologies such as HTML, CSS, XML, and SQL. Making these technologies work together requires a flexible and dynamically typed language. Unfortunately, flexibility comes at a cost. The performance of dynamically typed languages has long been many orders of magnitude lower than their statically typed counterparts. Poor JavaScript performance is a severe limitation for web applications.

Due to their highly dynamic structure, dynamically typed languages are poor candidates for just-in-time compilation. For this reason, most dynamically typed language runtimes instead rely on interpretation as their primary means of execution. We introduce Tamarin-Tracing, a new JavaScript virtual machine that aims to break the poor performance barrier. Our approach is to make just-in-time (JIT) compilation effective in a dynamically typed language runtime environment. We accomplish this by using a novel trace-based compilation technique [13], which is effective in eliminating many of the problems that arise when building traditional JIT compilers for dynamically typed languages. Our trace-based compiler selects and compiles only frequently executed "hot" code paths, while interpreting rarely executed code. Moreover, the code selected for compilation can be type specialized according to the type profile of the running application, allowing our compiler to make aggressive type driven optimizations. This makes it possible for us to remove much of the principal overhead of the dynamically typed JavaScript language, compiling untyped code as typed code.

We compare Tamarin-Central, a conventional compiler used in the current virtual machine shipping with Adobe Flash 9, with Tamarin-Tracing, a trace-based just-in-time compiler. We found that Tamarin-Tracing outperforms Tamarin-Central by up to 116%. Our experience with Tamarin-Tracing allows us to make the following contributions:

1. We present a trace-based just-in-time compiler for JavaScript.
2. We compare our compiler against other production JavaScript runtimes.
3. We investigate type specialization on traces.
4. We show the performance implications of using complex vs. simple opcodes.

VEE'09, March 11–13, 2009, Washington, DC, USA.
http://doi.acm.org/10.1145/1508293.1508304

2. Tamarin

Tamarin is the code name used for Adobe's virtual machines that implement ActionScript 3 (AS3) [1], a flavor of ECMAScript [11]. JavaScript is also a flavor of ECMAScript, however AS3 supports multiple constructs that JavaScript does not, such as optional static typing, packages, classes, and early binding. AS3 has a syntax that is similar to JavaScript, and most JavaScript can be executed on Tamarin without any modification. Tamarin does not directly operate on AS3 source code, but instead executes a bytecode intermediate format known as ActionScript bytecode (ABC) [2]. Conceptually, this is the equivalent of the Java Virtual Machine Language (JVML) bytecode [19]. ActionScript bytecode, like JVML bytecode, is a stack based intermediate representation. ActionScript source files are converted to ActionScript bytecode using the Adobe Flex compiler [3].

There are currently two development branches of Tamarin: Tamarin-Central (TC) and Tamarin-Tracing (TT). Tamarin-Central is the current virtual machine shipping with Adobe Flash 9, while Tamarin-Tracing is a research virtual machine slated for future versions of Adobe Flash. Tamarin-Central uses a conventional interpreter and JIT compiler, while Tamarin-Tracing leverages a trace-based compiler to achieve performance improvements. Tamarin-Tracing is partially a self-hosting compiler, with many portions of the system itself written in ActionScript 3. Tamarin-Tracing uses the Forth language as an intermediate representation. Before ActionScript source can be executed, it is first compiled to ABC and then translated to Forth. Tamarin-Tracing is essentially a Forth virtual machine.

Forth is a stack-based programming language that uses reverse Polish notation and operates on subroutines known as Words. Conceptually, programming in Forth is similar to programming in JVML, each Forth word behaving like a JVML bytecode. Tamarin contains roughly 200 Primitive Words, which implement the basic operations such as performing arithmetic or stack manipulation. In Forth, words known as Super Words can be constructed by combining other Primitive Words. For example, consider the JVML *iinc* opcode that increments a local variable's value. Conceptually, *iinc* combines the opcodes *iload, iconst_1, iadd, istore* into one opcode. Tamarin contains roughly 500 Super Words.

Tamarin-Tracing profiles the execution of Forth code through the Forth interpreter and selects frequently executed code paths for compilation. Tamarin-Tracing profiles backward branches and triggers trace recording when a certain threshold is crossed. This metric is effective in identifying hot loop headers. Once a loop header is discovered, the virtual machine attempts to record the execution of Forth words until the loop header is reached again. This ensures that recorded path, or *trace*, is a path through a hot loop, and therefore worth compiling. During the trace recording process, Super Words are broken down into Primitive Words. This simplifies the compiler and allows for more Super Words to be added in the future without modifying the trace compiler.

During trace recording, Tamarin-Tracing translates Primitive Forth Words into a low-level intermediate representation (LIR). The LIR passes through various optimization stages before it is finally compiled into machine code by the back-end. Tamarin-Tracing currently supports ARM, Thumb, and x86. Once the machine code finishes execution, control is returned back to the interpreter.

3. Opcode Granularity

Web applications differ in many ways from server and client side applications. Web applications need to load fast, and execute code in short bursts quickly. This means that JavaScript JIT compilers must have short startup times and keep compilation time to a minimum. A fast interpreter is also essential in the case of a snippet of JavaScript code that runs for only a few milliseconds. The argument that a slow interpreter is acceptable is no longer valid on the web.

The poor performance of interpreters is largely due to the high cost of opcode *dispatch*. Each opcode dispatch is at a minimum one indirect branch, which often introduces processor pipeline stalls. The complexity of individual instructions is largely dwarfed by the high dispatch overhead. A "simple" opcode has a fine granularity yet higher dispatch overhead. A "complex" opcode has a coarse granularity with the advantage of low dispatch cost. For this reason, a common interpreter optimization is to group single simple opcodes into larger complex opcodes. This reduces the number of dispatches at the expense of opcode complexity, which can lead to better interpreter performance. Forth is able to mend itself well to varying levels of opcode granularity. During interpretation, the coarse granularity of Super Words can reduce opcode dispatch cost, while during trace recording the compiler is able to reduce the Super Words to finer grained primitive words.

4. Forth

Forth operates on two stacks: the *data* stack and the *return* stack. The data stack is used for arithmetic operations, while the return stack is used for control flow operations. Consider the sample Forth code:

```
10 20 + .
```

This Forth code sequence pushes the numbers 10 and 20 onto the data stack. These numbers are then popped off the data stack, added, and their sum is then pushed back onto the stack. The "`.`" Forth word prints the result onto the screen.

Forth was chosen as an intermediate language due to the language's ease in refactoring and resulting code size. Forth contains a small number of primitive words, such as add. New words are constructed by concatenating previous words. Word definitions do not take parameters, and instead operate on either the data or return stack without any explicit parameters. Words can be rewritten and new words can be defined without changing large portions of code. The LIR can be created by translating only primitive words. Thus, we can quickly refactor the Forth as needed without changing the underlying LIR. Finally, we are targeting Tamarin to be used in both Desktop and Mobile environments, which requires Tamarin to have a small code size. Therefore, Forth suits our needs due to its compact code size.

4.1 Generating Super Words

Super Words are created by searching for sequences of words that do not span basic block boundaries. All available defined words are candidates for the super wording process. The Forth compiler looks at all such words and constructs Super Words out of them. Consider the following example for the Forth word *PrintScopeObject*:

```
: PrintScopeObject
        FindScope DUP IF EXIT THEN
        DROP SearchScope PrintObject ;
```

Both `FindScope` and `DUP` do not cross basic block boundaries, and are merged into one Super Word. The sequence `IF EXIT THEN` affects control flow and therefore cannot become a Super Word. The final sequence of words `DROP SearchScope PrintObject` again does not cross basic block boundaries, and can be merged into one Super Word. The final implementation of `PrintScopeObject` that the Forth interpreter actually executes is:

```
(op_FindScope_DUP)
(BRF + 2) // IF becomes Branch False
(EXIT) // BRF + 2 Jumps past this exit
```

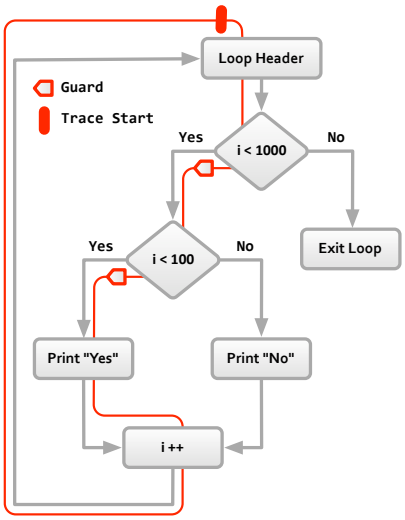
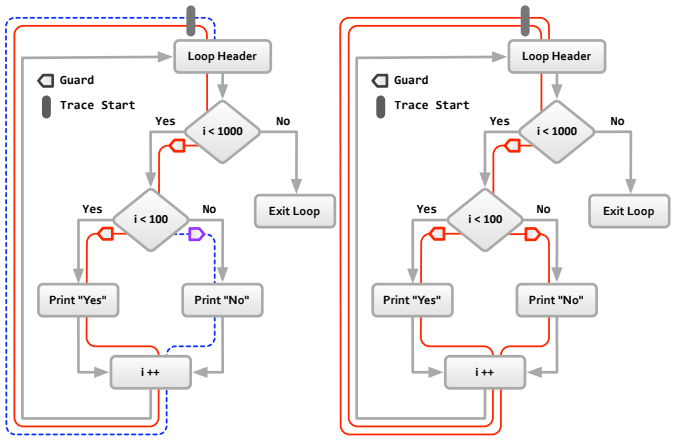


Figure 1: Example of a compiled trace. All recorded instructions are compiled into machine code as straight line code. Guard instructions are inserted where the control flow potentially changes.



(a) Trace linking (b) Trace trees

Figure 2: The difference between trace linking and trace trees. Trace linking jumps in between individual trace fragments. Trace trees attach other traces where they diverge in control flow so that they are optimized together as a whole.

```
(op_DROP_SearchScope_PrintObject)
(EXIT)
```

The `IF EXIT THEN` is converted to a branch if false statement. The last EXIT in the implementation is added because there is an implicit EXIT for all Forth words that are called. The word must return to the point in the program that called the word. This is similar to a method call/return pair in other programming languages.

5. Trace Compilation and Trace Trees

Trace-based compilation is a novel compilation technique especially applicable to dynamically typed languages. Tracing is a mixed-mode execution environment in which bytecode is initially interpreted, and only frequently executed ("hot") code is compiled. To detect such "hot" code regions that are suitable candidates for dynamic compilation, we use a simple heuristic where the interpreter keeps track of backward branches. The targets of such branches are mostly loop headers. For each target of a backward branch, there exists a counter that tracks how often this backward branch has been taken. When the counter exceeds a certain threshold, a loop header has been discovered.

The interpreter then enters "trace recording" mode, i.e. it records the instruction path subsequently taken, starting with the loop header. During trace recording, metadata of the ongoing program execution are stored, such as the current program counter and the state of local variables. If the recorded path leads back to the loop header, then a path has been recorded. This is called an initial "trace" through the loop.

Consider the example in Figure 1, which is the resulting trace of a `while` loop. Trace recording begins at the header of the loop and continues until the loop header is reached again. The check to ensure that `i < 100` is compiled as a guard instruction. The rest of the loop is inlined and compiled as straight line code since control flow does not diverge within a trace.

Forward branches inside a trace indicate alternative paths that may not yet have been discovered by our compiler. Branches to potential paths that were not detected during trace recording are compiled into *guard* instructions. A guard instruction is a check to ensure that the condition encountered during trace recording is still the same. Failure of a guard means that the program has arrived at a path that has not yet been compiled. This is called a "side exit" from the loop. In the case of a side exit, the compiled machine *bailout* code restores the interpreter state and hands control back to the interpreter. Since returning control back to the interpreter can be expensive, the alternative path that led to the side exit should be compiled as well if it is frequently executed. To handle these scenarios, interpretation resumes at the taken side exit, and the trace recorder continues recording instructions starting at the side exit location. These secondary traces are completed when the interpreter revisits the original loop header. This results in a collection of traces, spanning all observed paths through the "hot" code region.

5.1 Organizing Traces

There are three distinct methods of trace compilation: 1) trace fragments, 2) trace linking, and 3) trace trees. With trace fragments, each trace is completely independent of every other trace. No trace is aware of any other trace, and cannot use the context of when it was traced for machine code optimization purposes. Individual traces have the disadvantage of containing a lot of bailout code to restore state back to the interpreter. Bailout code can account for up to 80% of the total amount of generated machine code. A better solution is to connect the traces together if they meet at the same point in the program.

Consider the case where we wish to connect two traces: trace A jumps to trace B. Each individual trace must have code to first save the interpreter state and restore the interpreter prior to leaving the trace. Therefore, trace A saves state, executes, and restores state back to the interpreter. The interpreter then jumps to trace B, which also has to save state, execute, and restore state. Performance would be significantly better if trace A saves the interpreter state prior to execution, executes the trace, jumps to the machine code trace B, and allows trace B to restore the interpreter state when a side exit occurs. Trace linking extends trace fragments and exploits this optimization. Trace linking allows traces to call other traces without returning back to the interpreter, with execution staying in optimized machine code.

In contrast, a trace tree is a collection of traces that share the same entry point. A trace tree can only occur when traces originate at the same entry point in the program such as a while loop header. Trace linking can jump to traces wherever they connect, not necessarily at a loop header. The important distinction between trace linking and trace trees is that trace linking allows individual traces to call other traces, but each individual trace is compiled as an independent entity. Compiler optimizations are only applied to that one specific trace. Trace trees allow all traces to be compiled as a whole, enabling a larger view of the code and therefore better optimizations.

5.2 Type Specialization via Traces

One of the challenges of compiling dynamically typed languages is that there are numerous run-time checks, mostly on types and for method dispatch, which results in poor performance. Traditionally, type information has been stored either statically or by boxing and unboxing objects. Boxing refers to wrapping type information around an object, while unboxing refers to reading the type information and parsing the data as that type. This is useful in the case of adding a level of abstraction to the opcodes and lazily discovering the type of an object. However, there is a performance penalty for boxing. Types of objects usually do not change in most real world code. This observation was shown to be true for object receiver types by Garret et al. [14] in SELF, C++, and Cecil. It has also been our albeit anecdotal experience when dealing with user generated code. We use the phrase "type stability" to refer to the types of objects not changing during program execution, and exploit this fact to heavily optimize machine code. Type specialization refers to compiling code assuming specific types for objects.

Tracing has the intrinsic property of type specialization and method dispatch resolution for free. At trace recording time, the interpreter knows the type of an object at that point in the program. Whether it is an integer or a user defined type, trace compilation exploits "type stability" to generate machine code assuming the types remain the same during execution. For example, if we add two objects and both are integers during trace recording, we can compile the code with a native integer addition and a guard for an integer overflow. By assuming that most objects do not change their type during execution, we can achieve a significant performance increase by utilizing that type information for aggressive code optimization, compiling untyped code as typed code, and inserting guards when needed to ensure that object types have not changed.

In case of a static method invocation, the target method is fixed and no additional run-time checks are required. For dynamically dispatched methods where the callee's object type is unknown, the trace recorder inserts a guard instruction to ensure that the same actual method implementation that was found during trace recording is executed. If the guard fails because the actual callee method is not the same, control returns back to the interpreter. If the guard succeeds, we have effectively performed method callee specialization, removing the overhead required for a method dispatch on an unknown type. Since tracing can handle multiple alternative paths, eventually the trace compiler specializes method invocations for all commonly occurring callee object types. Finally, we can extend this concept one step further and specialize not only on method dispatch callee types, but on the actual types of the method parameters.

5.3 Compiling Traces

Instead of directly compiling Forth, the trace recorder in Tamarin-Tracing translates the Forth instructions to a low-level intermediate representation (LIR). The LIR instructions actually become the IR for optimization, and finally machine code. Super Words are broken down into their individual primitive words and are translated to LIR instructions during trace recording. LIR is in static single assignment (SSA) form [10] with LIR references pointing to values.

One interesting aspect is that the phi nodes do not need to write values upon leaving SSA form. Phi nodes indicate that a variable has been changed at a merge point in a control flow graph. Traces are straight line sections of code, so control flow can merge only at the backward branch at the end of the traced loop. One implementation of traces would be to have phi nodes for the variables that changed inside the loop and generate code at the beginning and end of a trace to read/write variable values in the correct place. Tamarin-Tracing does not require this code, but uses the Forth data stack. Instead of reading variable data at the beginning of the loop, Tamarin-Tracing reads the Forth stack location of the variable and directly operates on the Forth stack in machine code. When the execution exits machine code, there is no need to restore variable values into a specific location. Tamarin-Tracing performs optimizations on recorded code with constant folding during actual trace recording time, and common subexpression elimination and redundant store elimination on the LIR.

Tamarin-Tracing can use either trace linking or trace trees. With trace linking, the trace recorder begins recording from the side exit point when a side exit occurs. If the trace recorder successfully records a hot side exit, the trace is compiled into machine code, and the pointer to the newly compiled code is returned to the side exit. The side exit patches a jump to the newly recorded trace. The link between the two traces is now complete, with one trace exiting to the side code, which then jumps to the other trace fragment. This transition is all in machine code without the need to return to the interpreter. Trace trees require the whole tree to be recompiled in order to take advantage of the context for both traces. Trace recording occurs because of a hot side exit. Instead of having the side exit patch a jump to the new trace, both the trace where the side exit occurred and the newly recorded trace are compiled together. Throwing away the old machine code and recompiling the whole tree allows for faster machine code as we can optimize both traces within the context of each other.

One important optimization where Tamarin-Tracing can gain impressive performance increases is type specialization. During interpretation, objects are boxed and unboxed into 64 bit "atoms" with the upper 32 bits containing type information and the lower 32 bits as the actual value. In our implementation, compiled code still boxes and unboxes atoms at most operations. Only some boxing and unboxing operation pairs are optimized away through common subexpression elimination in SSA form, but boxing and unboxing actions still occur within the loop code. An additional significant performance optimization would be to be more aggressive in type specialization, i.e. to leave atoms unboxed in machine code and rebox them prior to returning to the interpreter.

5.4 Merge Nodes

One problem with traces is the amount of duplicate code in a trace. If multiple traces flow through the same program path, the code for the path is duplicated. In order to reduce code size, we use merge nodes. A merge node is where two predecessors in the control-flow graph both jump to the same location. The trace representing the merge point in the control flow graph is known as a merge node.

Consider the example in Figure 3. The trace fragments 1 and 2 diverge in program control flow, but both *merge* at the point the if/else statement ends. When trace fragment 2 is being recorded, the recorder knows that both fragments 1 and 2 have the code where the if/else statement ends in their trace. Instead of duplicating this code into trace fragment 2, the merge point becomes its own independent fragment, which is called a *merge node*. The machine code of trace fragment 2 jumps to the newly created merge node. From the

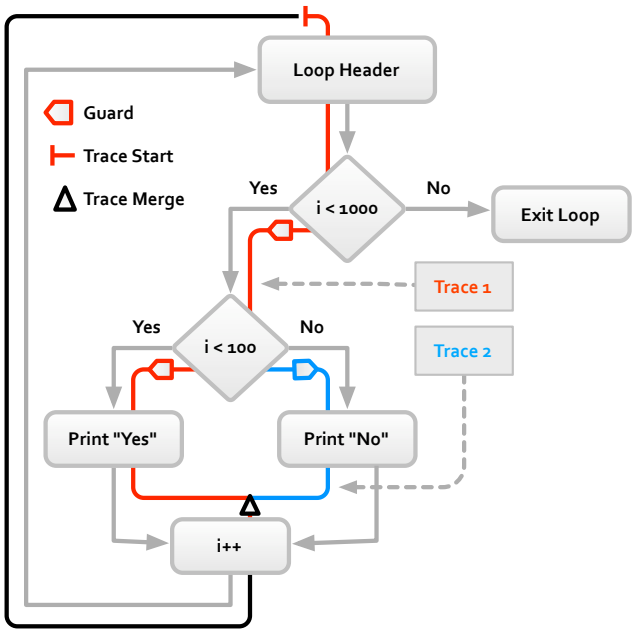


Figure 3: Merge nodes are located where two nodes in the control-flow graph both flow to the same location. This merge node is compiled into its own trace, and the two predecessor traces jump to the newly compiled trace.

```
var i, iLess20, iLess40, elseState = 0;

for (i = 0; i < 100; i++) {
    if (i < 20) { iLess20 = i; }

    if (i < 40) { iLess40 = i; }
    else { elseState = i; }
}
```

Figure 4: Example source code used in Tamarin-Tracing execution example.

compiler's perspective, the merge node is exactly the same as any other trace.

Trace fragment 1 is not recompiled to jump directly to the merge node, and contains the merge node in its own duplicated code. We do not recompile fragments since it is too expensive in our target environment. All further traces that follow the same control flow call the merge node. Therefore, the code savings emerge from the second, third, etc, compiled traces. It is a configurable option when merge nodes are created in Tamarin-Tracing. The default configuration is to only compile a merge node when a third trace is detected. We chose the third trace as a balance between compilation time and space.

6. Example of Tamarin-Tracing Execution

Gluing all the different components of Tamarin-Tracing together is a comprehensive task. This section aims to provide a concrete example of what occurs from the point of ActionScript bytecode until machine code is executed. We start by taking a look at ABC and how it is converted to Forth. We then examine what is traced when the code gets hot, show the LIR that is generated during trace recording, and finally show the generated machine code. Consider the source code in Figure 4, specifically the `if` statement construct testing the condition `i < 20`. A flowchart depicting the whole process of converting ActionScript 3 to x86 instructions is shown in Figure 5.

Consider the ActionScript bytecode that is generated for the ActionScript3 `i<20`. The first opcode `getslot` represents an access to a local variable. Local variable numbers are known at compile time, and are therefore associated with dedicated memory locations called slots. The ActionScript variable `i` is given slot number 1. The second opcode `pushbyte` reads a number from the file and pushes it onto the virtual ActionScript stack. The third opcode `ifge` represents comparison and conditional jump if greater or equal. If the condition is true, control flow jumps to opcode 51.

These three ActionScript bytecodes are then converted to Forth at run time. The ABC `getslot 1` becomes the Forth sequence of a literal constant 1, followed by a fetch to get the boxed value of that slot location. The ABC `pushbyte 20` becomes the Forth word `Constant 20`. The number 20 is then pushed onto the Forth data stack. The Forth word *Integer Box* takes the value on the top of the data stack, and boxes it to become an integer. The Forth word *ifIt* checks the ActionScript condition `i < 20` and pushes the result onto the Forth data stack. *Branch False* checks the condition on top of the Forth data stack and branches to the specific address if the result of *ifIt* is false.

When the loop is hot enough, in our case once a backwards branch has been taken 10 times, the ActionScript variable `i` is 10. The trace recorder inlines the ActionScript statements where `i < 20`. During tracing, the actually executed Forth words are converted to LIR. The LIR does not contain the branch anymore, but uses a *guard* instruction to ensure that `i < 20`.

An object that contains pointers to all the local variables is on top of the Forth stack. The LIR opcode loadQuadWord (ldq1) reads the C address of the object from the Forth data stack. The LIR variable is named ldq1 because LIR is in SSA form and this is the first instance of ldq. Each object on the Forth stack is boxed as a 64 bit word with the top 32 bits as type information and the bottom 32 bits as the actual data in binary form. loadLowerQuadWord (qlo) reads the lower 32 bits from a boxed item, and stores that value in qlo1. The getSlotvalueBox (svb) gets the local variable from the object address located in qlo1 and fetches an object located in slot 1. The newly read object is then unboxed as an integer, with the result being stored in svb1. We now have the value of the ActionScript variable `i`. lessThan (lt1) checks that svb1 is less than 20, which is the ActionScript condition `i < 20`. Note that instead of storing the number 20 in a variable, it is used as an immediate operand in the `lessThan` instruction.

Finally, TraceExitIfFalse (xf4) is an example of a *guard* instruction. The lessThan guards the ActionScript condition `i < 20`. LIR TraceExitIfFalse ensures that this value is true, otherwise the trace is exited. It exits to location 51, which is the jump target of the original ABC jump bytecode. The LIR is now optimized, and physical registers are assigned to the operands by the register allocator. The sequence of LIR is compiled to x86 instructions.

The value qlo1 is stored into a register, and a native call to a C function getSlotvalueBox is executed. The value of that function is returned in EAX, which is the integer value of the ActionScript variable `i`. Then the returned value is compared to the number 20. If the condition `i < 20` is false, there is a jump to *bailout* code that restores the state back to the Forth interpreter.

This same process is done for the whole trace. We can see the whole execution of the example source code and the resulting compiled traces in Figure 6. The first trace is "Trace 6" representing the control flow where the ActionScript condition `i < 20` is true.

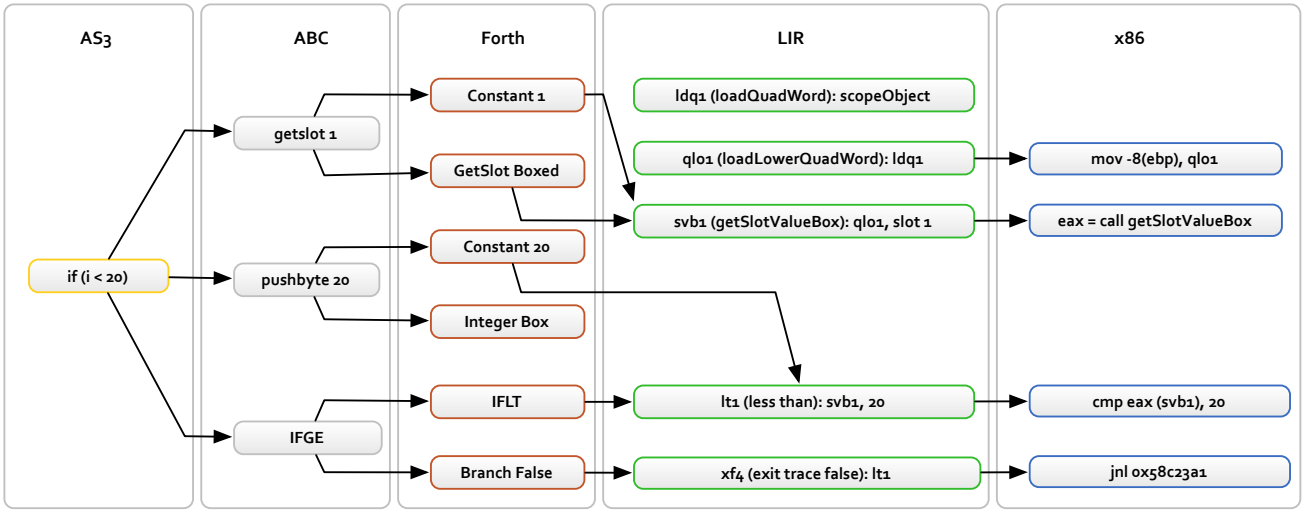


Figure 5: Flowchart of ActionScript 3 source to x86 machine code. Optimizations are applied during the translation of Forth to LIR.

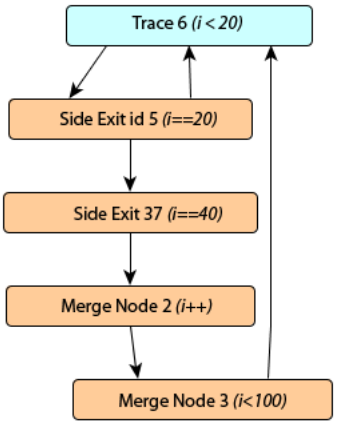


Figure 6: A visualization of traces, where side exits occur, and how traces are linked together. Side exit id 5 occurs when `i == 20`. Side exit id 37 occurs when variable `id == 40`. Finally two merge nodes are created. One merge node contains `i++` while the other checks the loop condition `i < 100`.

When `i == 20`, a side exit occurs, which in this case is side exit id number 5.

Trace recording begins again from the side exit location, compiling the path ignoring the block where `i < 20` and compiling the statement where `i < 40`, which is `iLess40 = i`. Compiling this trace eventually reaches back to the main loop header with side exit id 5 jumping back to "Trace 6". Once `i == 40`, we again exit the trace at side exit id 37. However, we now have two merge node conditions.

Adobe Flex compiles loop checks at the end of the loop with two basic blocks. Once the loop body finishes, control flow jumps to the first basic block that contains the ActionScript code `i++`. The second basic block contains the actual loop condition `i < 100`. Therefore, "Merge Node 2" contains the code to increment and set the ActionScript variable `i`. "Merge Node 3" is the loop condition. When side exit 37 occurs, trace recording compiles both of these basic blocks as independent fragments, i.e. independent merge nodes. Side exit 37 calls "Merge Node 2", which jumps to "Merge Node 3", and finally "Merge Node 3" has reached the same point in program flow as "Trace 6".

During our explanation of merge nodes, we stated that a merge node occurs when two traces meet at a point in the control flow. This is our criteria for detecting merge nodes, but Tamarin-Tracing by default does not compile a merge node when two traces share the same merge point. Instead, Tamarin-Tracing waits for a third trace to occur prior to compiling the merge node. Hence, in the earlier example, a merge node could have been formed once the first side exit occurred. The first side exit duplicated the `i++` and while loop condition inside its own trace. Once the second side exit occurred, the merge nodes were generated.

7. Results

We evaluate our implementation using Apple's SunSpider [27] benchmark. We compare Tamarin-Tracing to three other JavaScript virtual machines: Tamarin-Central [23], which is the current ActionScript virtual machine in Adobe Flash; SquirrelFish [28], the virtual machine in Apple's Safari; and SpiderMonkey 1.8 [22], the current JavaScript implementation in Mozilla Firefox 3.0. All tests were done with RELEASE builds, or in the case of SpiderMonkey, with optimizations enabled. The benchmarks were run on a MacBook Pro 2.4 Ghz with 4 GByte of memory running Mac OS X Leopard 10.5.4. Performance is evaluated by calling getTime(), which returns a UNIX timestamp. This method is called prior to running the SunSpider test, again calling getTime() at the end of the test, and subtracting the start time from the end time.

Figure 7 shows the results of the performance measurements. The baseline is Tamarin-Tracing's interpreter where tracing is disabled, i.e. where no JIT compilation occurs. Considering Tamarin-Tracing and Tamarin-Central both have JIT compilers, SquirrelFish outperforms all available implementations on most of the tests. Both SquirrelFish and SpiderMonkey have a small number of complex opcodes that do a lot of work, which gives them a significant performance advantage as the overhead of the interpreter dispatch is minimized. SquirrelFish is a direct threaded interpreter, while

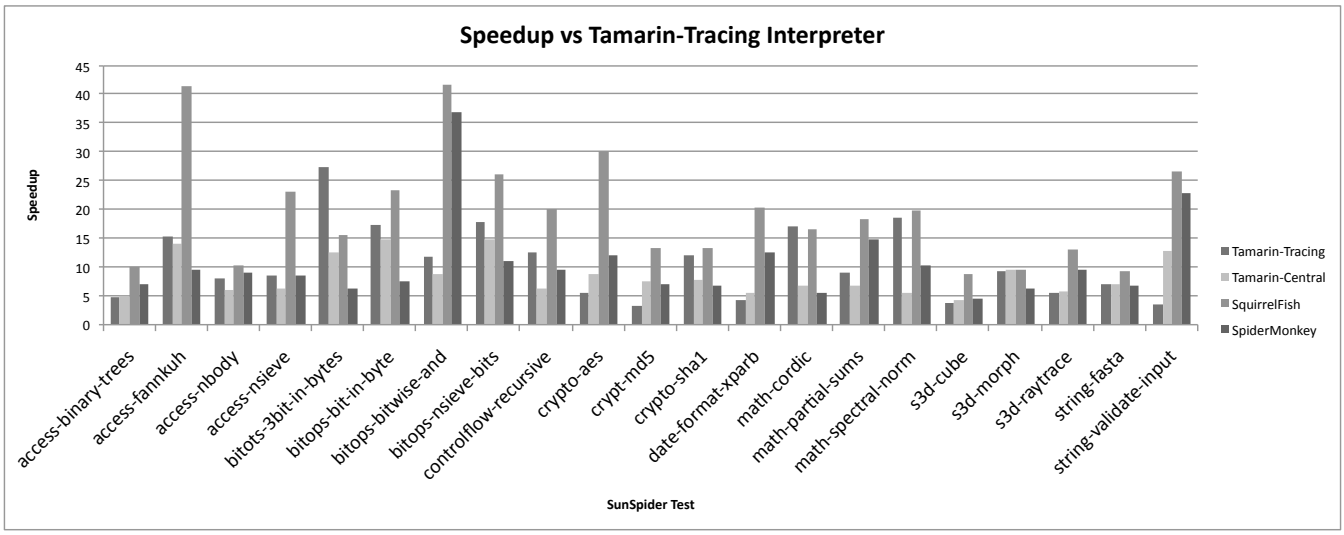


Figure 7: Performance of Tamarin-Tracing, Tamarin-Central, SpiderMonkey, and SquirrelFish. Baseline is Tamarin-Tracing running in interpreter-only mode without JIT compilation. SquirrelFish has the fewest yet most complex opcodes. SpiderMonkey and Tamarin-Central both have simpler opcodes than SquirrelFish, with Tamarin-Tracing having the simplest. Simple opcodes result in a performance loss.

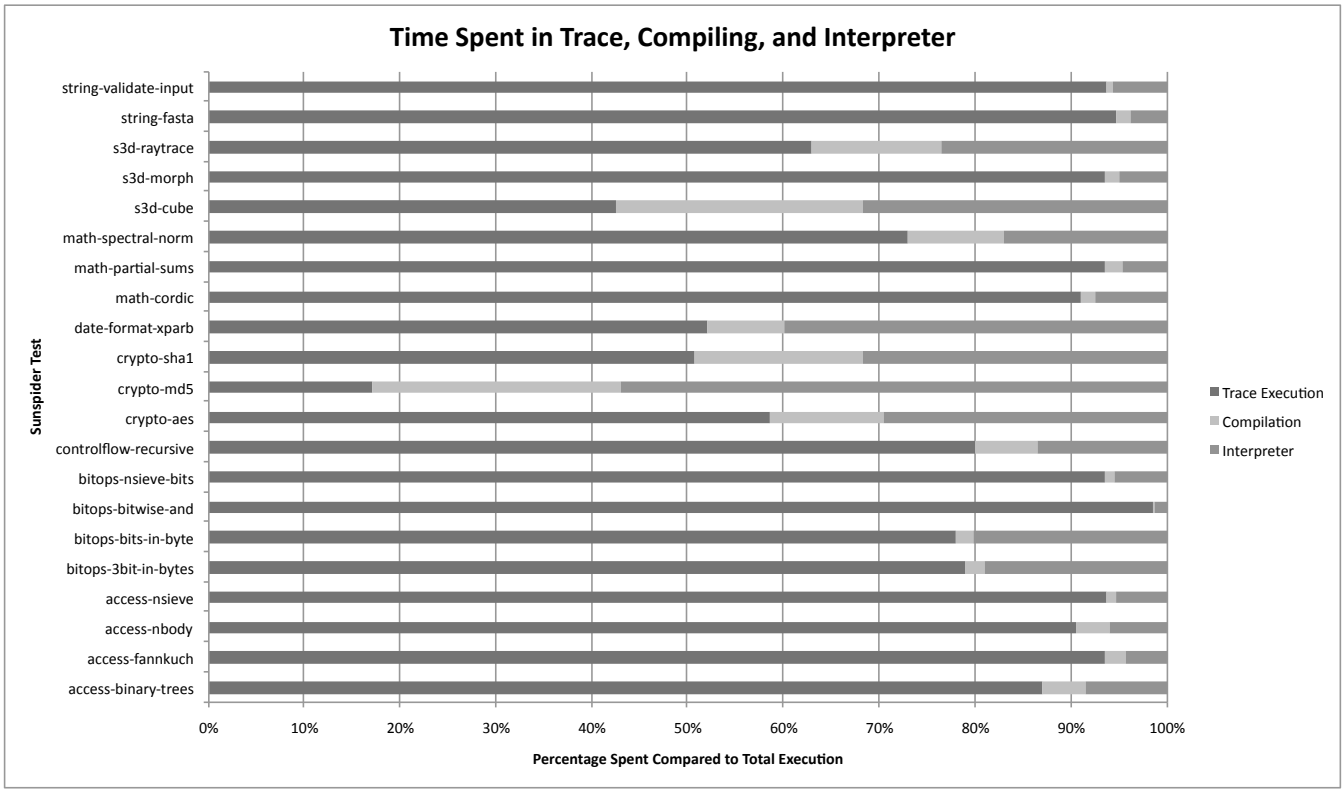


Figure 8: Execution time spent in a trace, compiling a trace, and the interpreter excluding startup costs. When a large amount of time is spent in a trace, Tamarin-Tracing outperforms Tamarin-Central.

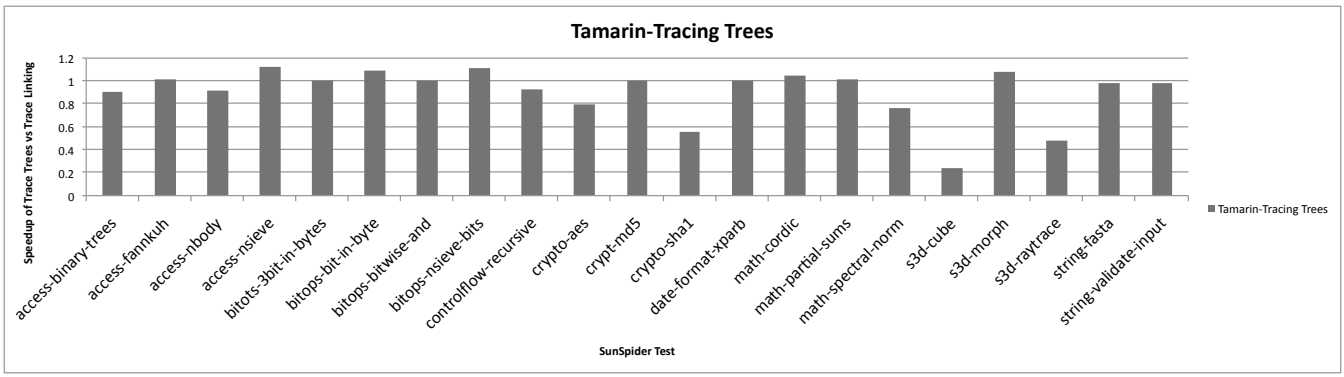


Figure 9: Performance of Tamarin-Tracing with and without trace trees. Most of the time in linear code, the performance is roughly equal. In branch heavy code, the overhead of recompiling the whole trace tree negates any performance improvements within the optimized machine code.

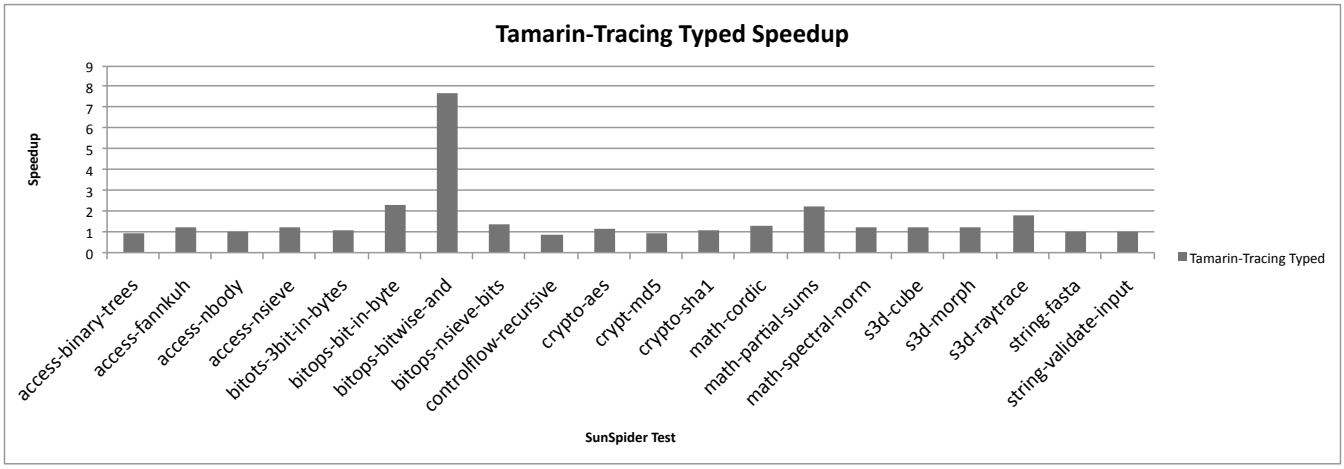


Figure 10: Performance of Tamarin-Tracing typed vs. untyped code.

SpiderMonkey is an indirect threaded interpreter. Dispatch is especially important as SunSpider benchmarks run only for tens of milliseconds. For example, in crypto-md5, s3d-cube, date-format-xparb, and crypto-aes, Tamarin-Tracing is the slowest implementation. In crypto-aes and s3d-raytrace, Tamarin-Tracing is slower than Tamarin-Central.

The slow performance for these tests is in large part due to the fact that much time is spent in the interpreter. Figure 8 shows the percentage of time spent executing compiled traces, compiling traces, and in the interpreter. These numbers exclude the startup costs. Due to the results, we believe having semantically complex opcodes to increase interpreter performance is the proper method of building a JavaScript Tracing compiler.

However, the performance results also gives insight into the potential for tracing, despite the overhead of the interpreter. Tamarin-Tracing shows a good performance beating or matching SquirrelFish in bitops-3bit-in-byte, crypto-sha1, s3d-morph, math-spectral-norm, and math-cordic. In bitops-3bit-in-byte, Tamarin-Tracing is the fastest implementation, and is 116% faster than Tamarin-Central. In about half of the tests, Tamarin-Tracing performs better than SpiderMonkey. We can also see that in these benchmarks, a large percentage of execution time is within a trace.

While string tests such as string-validate-input and string-fasta are executed within traces, Tamarin-Tracing is still behind its competitors. This is due to the overhead of the internal string representation and is a known performance bottleneck. Thus while tracing is able to achieve a significant performance improvement, the interpreter is the bottleneck for overall performance. In our current implementation, the dispatch is too expensive due to indirect threading and the simplicity of the Forth words. Another dispatch technique could speed up the Forth interpreter. However, since a small code size is a requirement, implementing direct threading is not a viable option.

We can also see the limits of a traditional compiler in Tamarin-Central. Tamarin-Central compiles a method the first time it is called, and works specifically on atoms in machine code. This means that everything is boxed/unboxed in machine code. While it is still faster than an interpreter, the limits of compiling dynamically typed code without traces is evident.

Figure 9 compares Tamarin-Tracing performance when compiling with trace linking versus trace trees. The performance is roughly the same on most of the tests because the code is not particularly branch heavy, and most of the code is linear. When code is branch heavy, such as in s3d-cube and s3d-raytrace, the amount of time necessary for recompiling the whole tree after every branch

is added to a trace tree voids any gain that can be reached by better optimized machine code. We will examine ways of reducing this overhead in our future work.

Figure 10 shows the performance of Tamarin-Tracing with typed versus untyped code. To introduce static typing, we changed the benchmark source code and use features that are available in ActionScript 3, but not in JavaScript. The performance increase of static typing ranges from none at all to 7.5x faster as the case with bitops-bitwise-and. It shows the upper bound of speedup that could be achieved by Tamarin-Tracing if it eliminated all dynamic type checks from the compiled code. If we would specialize traces more aggressively, we could expect this performance increase in untyped code. We could not compare typed code results to SpiderMonkey or SquirrelFish because they support only standard JavaScript.

8. Related Work

Dynamically typed languages have traditionally been implemented with interpretation. However, the need for increased performance has led to a wide range of compilers for these languages including Python, Ruby, and PHP. Most optimized implementations of dynamic languages are based on existing virtual machines such as Java or .NET, i.e. they generate statically typed bytecodes that simulate dynamic typing. Other than Tamarin and TraceMonkey [18], we are not aware of other existing JIT compilers built directly and specifically for JavaScript. Mozilla SpiderMonkey [22] is a pure interpreter, SquirrelFish [28] is a fast interpreter used in Safari from Apple, but still an interpreter. JScript [20] is Microsoft's implementation of JavaScript, but traditional JScript uses an interpreter. JScript.NET again uses the .NET runtime to compile code.

The idea of running traces to specialize on hot path code has been explored before in the binary rewriting system Dynamo [6]. It utilizes run-time information to find hot paths, and optimizes machine code based on these hot paths. Dynamo compiles only single traces, and utilizes trace linking to link traces together if possible. Tamarin-Tracing has the option to take a similar approach, even utilizing some of the same terminology such as fragments that only start at a backward branch. Tamarin-Tracing also extends Dynamo by having the option of compiling trace trees.

Tracing as a means to speedup interpretation is proposed by Zaleski et al. [30] with their YETI system. YETI identifies regions of hot code, which can be either methods, partial methods, or traces and compiles these hot regions into machine code. Their interpreter then calls these compiled regions of hot code. Tamarin-Tracing only detects hot regions via traces, and inlines any portion of a method that may be found during trace recording.

Traces were extended with the concept of trace trees by Gal et al. [13]. Instead of connecting individual traces to each other at end of a trace with each fragment being independent of each other, trace trees allow traces to be attached to other traces as long as they connect to a loop header. Tamarin-Tracing also supports this feature with a command line switch. Lua JIT [25] is starting to explore trace-based compilation for the lua programming language.

Ertl et al. [12] extensively studied the performance and implementation of efficient interpreters, comparing the performance of various interpreter dispatch techniques. They found that direct threaded [7] dispatch is the most efficient. The work of Shi et al. [26] reduced the number of interpreted instructions by converting stack based code to a register based representation. SquirrelFish uses these two techniques to achieve a significant performance increase in their interpreter, and is something we are looking to add to Tamarin.

Type inference has previously been studied extensively, with many algorithms to statically derive the types of specific variables. Type inference to statically check for type errors is explored by Thatte [29], and further extended in [21] by adding a universal type. Adding a type system has been previously proposed in object oriented languages such as Smalltalk [15]. Agesen et al. [4] studied type inference in the prototype-based language SELF. Aiken et al. [5] investigated the use of type inference to improve the performance of dynamically typed languages through the use of abstract interpretation [9, 24]. Our solution differs in that we use concrete type information that is observed during actual program execution, and assume type stability in the compiled trace code. If the type changes, which rarely occurs in real-world programs, we can just speculate again on the new type.

Using concrete type information to optimize compiled code has been studied in other dynamically typed languages. Chambers et al. [8] customized methods based on the callee type of a procedure, creating multiple copies of optimized machine code for each callee type. This is similar to our method invocation type specialization, except that we can also specialize on method parameter types, not just the callee type. Hölzle et al. [17] developed a system of using run-time feedback-based method specialization in SELF by profiling the running application and recording the callee of a method dispatch. The difference with tracing is that method specialization is done in one step by trace recording, without the need of an extra profiling phase. Hölzle et al. [16] introduced "polymorphic inline caches" that cache all observed callee types. Our technique can apply the same principles by compiling multiple traces, with each trace compiled with one callee type.

9. Future Work

As the benchmark results show, a fast interpreter is essential to a high performance JavaScript virtual machine. Tracing has huge potential, but it cannot overcome the penalties of using a slow interpreter. In our next implementation, we will be taking Tamarin-Central and converting it to a direct-threaded interpreter. Tamarin-Central has more complex opcodes and a much faster interpreter than Tamarin-Tracing. The Tamarin-Central interpreter is already at least 50% faster when adding direct threading. We will incorporate a tracing-jit into Tamarin-Central, and follow aggressive type specialization instead of boxing/unboxing objects in a trace. We also plan to switch our tracing methodology by going away from tracing specific opcodes as a whole, as we do in Tamarin-Tracing, to tracing specific portions of a complex opcode. Finally, we will also add a tracing compiler to SpiderMonkey, in a project called TraceMonkey [18]. SpiderMonkey has complex opcodes and an even faster baseline interpreter than Tamarin-Central.

10. Conclusions

In this paper, we explored a trace-based JavaScript compiler. We had impressive performance results compared to Tamarin-Central, which shows the potential of tracing. However, when programs run only for a few milliseconds, a fast interpreter is crucial. Tamarin-Tracing answered the question: What granularity should an opcode be as it relates to tracing? Can trace compilation overcome the overhead of a slow interpreter? That answer is "no" for typical JavaScript web applications. Trace compilation has an enormous amount of potential by exploiting type stability. Therefore, in order to unlock the performance that tracing can provide, the interpreter needs complex opcodes and the trace compiler needs to trace and compile through specific portions of an opcode.

Acknowledgements

This research effort was partially funded by the National Science Foundation (NSF) under grant CNS-0615443. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science Foundation (NSF), or any other agency of the U.S. Government.

References

[1] Adobe Systems Inc. *ActionScript 3 Language Specification*, 2006. http://livedocs.adobe.com/specs/actionscript/3/wwhelp/wwhimpl/js/html/wwhelp.htm.

[2] Adobe Systems Inc. *ActionScript Virtual Machine 2 Overview*, 2007. http://www.adobe.com/devnet/actionscript/articles/avm2overview.pdf.

[3] Adobe Systems Inc. *Adobe Flex SDK*, 2008. http://www.adobe.com/products/flex/flexdownloads/#sdk.

[4] O. Agesen, J. Palsberg, and M. I. Schwartzbach. Type Inference of SELF: Analysis of Objects with Dynamic and Multiple Inheritance. *Software Practice and Experience*, 25(9):975–995, 1995.

[5] A. Aiken and B. Murphy. Static Type Inference in a Dynamically Typed Language. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 279–290. ACM Press, 1991.

[6] 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.

[7] J. R. Bell. Threaded Code. *Communications of the ACM*, 16(6):370–372, 1973.

[8] C. Chambers and D. Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 146–160. ACM Press, 1989.

[9] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 269–282. ACM Press, 1979.

[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*, 13(4):451–490, 1991.

[11] ECMA. *Standard ECMA-262: ECMAScript Language Specification*, 3rd edition, 1999. http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[12] M. Ertl and D. Gregg. The Structure and Performance of Efficient Interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25, 2003.

[13] 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.

[14] C. Garrett, J. Dean, D. Grove, and C. Chambers. Measurement and Application of Dynamic Receiver Class Distributions. Technical Report CSE-TR-94-03-05, University of Washington, 1994.

[15] J. O. Graver and R. E. Johnson. A Type System for Smalltalk. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 136–150. ACM Press, 1990.

[16] U. Hölzle, C. Chambers, and D. Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 21–38. LNCS 512, Springer-Verlag, 1991.

[17] U. Hölzle and D. Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 326–336. ACM Press, 1994.

[18] JavaScript:TraceMonkey - Mozilla Wiki, 2008. https://wiki.mozilla.org/JavaScript:TraceMonkey.

[19] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[20] Microsoft Developer Network. *JScript (Windows Script Technologies)*, 2008. http://msdn.microsoft.com/en-us/library/hbxc2t98(VS.85).aspx.

[21] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[22] Mozilla Foundation. *SpiderMonkey (JavaScript-C) Engine*, 2008. http://www.mozilla.org/js/spidermonkey/.

[23] Mozilla Foundation. *Tamarin Central*, 2008. http://hg.mozilla.org/tamarin-central/.

[24] F. Nielson. Program Transformations in a Denotational Setting. *ACM Transactions on Programming Languages and Systems*, 7(3):359–379, 1985.

[25] M. Pall. *LuaJIT Roadmap*, 2008. http://lua-users.org/lists/lua-l/2008-02/msg00051.html.

[26] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg. Virtual Machine Showdown: Stack Versus Registers. *ACM Transactions on Architecture and Code Optimization*, 4(4):1–36, 2008.

[27] *SunSpider JavaScript Benchmark*, 2008. http://webkit.org/perf/sunspider-0.9/sunspider.html.

[28] *Surfin' Safari - Blog Archive - Announcing SquirrelFish*, 2008. http://webkit.org/blog/189/announcing-squirrelfish/.

[29] S. Thatte. Type inference with partial types. In *Proceedings of the International Colloquium on Automata, Languages and Programming*, pages 615–629. LNCS 317, Springer-Verlag, 1988.

[30] 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.