MSR-TR-2010-27

SPUR: A Trace-Based JIT Compiler for CIL

Michael Bebenita [*] Florian Brandner [†] Manuel Fahndrich Francesco Logozzo
Wolfram Schulte Nikolai Tillmann Herman Venter

Microsoft Research

{hermanv,logozzo,maf,nikolait,schulte}@microsoft.com

Abstract

Tracing just-in-time compilers (TJITs) determine frequently executed traces (hot paths and loops) in running programs and focus their optimization effort by emitting optimized machine code specialized to these traces. Prior work has established this strategy to be especially beneficial for dynamic languages such as JavaScript, where the TJIT interfaces with the interpreter and produces machine code from the JavaScript trace.

This direct coupling with a JavaScript interpreter makes it difficult to harness the power of a TJIT for other components that are not written in JavaScript, e.g., the DOM implementation or the layout engine inside a browser. Furthermore, if a TJIT is tied to a particular high-level language interpreter, it is difficult to reuse it for other input languages as the optimizations are likely targeted at specific idioms of the source language.

To address these issues, we designed and implemented a TJIT for Microsoft's Common Intermediate Language CIL (the target language of C#, VisualBasic, F#, and many other languages). Working on CIL enables TJIT optimizations for any program compiled to this platform. In addition, to validate that the performance gains of a TJIT for JavaScript do not depend on specific idioms of JavaScript that are lost in the translation to CIL, we provide a performance evaluation of our JavaScript runtime which translates JavaScript to CIL and then runs on top of our CIL TJIT.

1. Introduction

Tracing just-in-time compilers (TJITs) determine frequently executed traces (hot paths and loops) in running programs and focus their optimization effort by emitting optimized machine code specialized to these traces. Tracing JITs have a number of advantages over traditional JITs and static compilers in that they are able to harness runtime information, such as indirect jump and call targets, dynamic type specialization, and the ability to inline calls and unroll loops more aggressively due to the focus on hot traces [2].

Prior work has established this strategy to be beneficial for dynamic languages such as JavaScript, where the TJIT interfaces with an interpreter and produces machine code from the JavaScript trace [17]. In such a setting, the TJIT works on the high-level programming constructs from the original source language (e.g., JavaScript) and can take advantage of that information for its optimizations. As a result, it is not clear whether similar performance benefits can be achieved by first using standard compilation of the high-level dynamic language to an intermediate typed language such as Microsoft's Common Intermediate Language [14] (CIL) and applying trace jitting to this intermediate language instead.

In this paper we thus examine the following hypothesis: *Trace jitting a typed intermediate language (such as CIL) and compiling high-level dynamic languages and their runtimes to that intermediate level, enables similar performance gains as directly trace jitting the high-level language.*

To confirm this hypothesis, we designed and implemented SPUR, a trace JIT for CIL (the target language of C#, VisualBasic, F#, and many other languages). Working on CIL enables TJIT optimizations for any program compiled to this platform. To validate that the performance gains of a trace JIT for JavaScript do not depend on specific idioms of JavaScript that are lost in the translation to CIL, we provide a performance evaluation of our JavaScript runtime which translates JavaScript to CIL and then runs on top of our CIL trace JIT. Our evaluation shows that the performance of the final machine code generated for standard JavaScript benchmarks on SPUR is competitive with the best current JavaScript engines.

1.1 Basic Insight

Statically compiling dynamic languages, such as JavaScript, to efficient code ahead of runtime is difficult due to the dynamic nature of the operations in the language. For instance,

[*] Contributed to this project during an internship at Microsoft Research.
[†] Contributed to this project during an internship at Microsoft Research.

the addition operation $+$ of JavaScript has numerous semantics, based on the actual runtime types of its arguments. It can be the addition of numbers, or concatenation of strings, along with conversions of operands to appropriate types. E.g., adding a string and a number forces the number to be converted to a string and the operation ends up being string concatenation.

TJITs such as [17] produce efficient code for such an addition operation by observing the particular types of the operands at runtime and then specializing the operation to those operand types, thereby saving many otherwise necessary tests to determine what operation to apply. Note that some tests still remain as guards in the trace to determine if the special case applies on future executions, but the number and placement of guards can usually be optimized or hoisted out of inner loops.

In contrast, our TJIT knows nothing about the semantics of JavaScript's addition operation. Instead, our JavaScript runtime (written in C# and compiled to CIL with the standard C# compiler) simply contains methods for each primitive JavaScript operation such as addition. These methods perform the necessary tests to determine the types of the operands and what operation to actually perform. Thus, these methods encode all the necessary tests and conversions in CIL code.

If a JavaScript program translated to CIL contains an addition operation, it will show up as a call into the JavaScript runtime's addition method. Our TJIT then traces from the CIL instructions resulting from the JavaScript code *into* the CIL instructions of the JavaScript runtime's addition operation. The trace produced navigating the numerous conditional branches in the implementation of addition is the trace specialized to the current operand types.

As a result, our TJIT does not depend on any JavaScript semantics but still specializes the dynamic behavior of the CIL code just as a TJIT working at the source language level would.

1.2 Motivation

Coupling a TJIT with a high-level language makes it difficult to harness the power of the TJIT for other components that are not written in that language, e.g., the DOM implementation or the layout engine inside a browser. Furthermore, if a TJIT is tied to a particular high-level language, it is difficult to reuse it for other input languages as the instructions traced and optimized are language specific.

Instead, our approach of adding trace jitting to a language neutral intermediate language provides a basis for reaping the benefit of trace compilation for an entire platform and multiple languages compiled onto the platform. The ability to trace through multiple software components and abstraction layers provides an excellent way to optimize away the "abstraction tax" of good software engineering, which calls for many small components and abstraction barriers. As part of the SPUR project, we are also in the process of implementing a browser DOM and layout engine in C#, which will enable us in the future to optimize traces that span JavaScript code, JavaScript runtime code, DOM, and layout code.

In comparison to previous work on trace jitting intermediate languages such as Java byte code [19, 29], our work differentiates itself by not having an interpreter for CIL at all. Instead, we use multiple levels of jitting: 1) a fast profiling JIT producing unoptimized code that determines hot paths, 2) a fast tracing JIT producing instrumented code for recording traces, and 3) the optimizing JIT for traces itself.

Our JITs are not based on the production Microsoft CLR JIT; instead, they have been written from scratch for the SPUR project; besides our trace optimizations, our JITs do not perform any optimizations, and employ a simple register allocator. One design goal was to enable efficient transitions between the different code versions.

Having all code compiled and run as machine code in a common environment allows our JIT to support scenarios not handled by previous work, such as calling into precompiled runtime code from an optimized trace without any additional overhead, even if that code may throw exceptions or causes garbage collection. In fact, there are no inherent operations that force us to abort a trace.

Our contributions in this paper are:

- We provide a positive answer to the hypothesis that tracing of a typed intermediate language results in similar performance gains when JavaScript is compiled to that level, as trace jitting JavaScript directly.
- To our knowledge, SPUR is the first tracing JIT based on jitting alone, without the need for an interpreter and without the need for adapter frames to mediate between different code versions. We provide details of how to transition between the different code versions emitted by our system.
- Our approach is also unique in that we have both precompiled x86 and CIL code for some components (e.g., the JavaScript runtime) available at runtime. This allows us to either call into precompiled code, or record a trace, optimize and recompile the original CIL intermediate form for particular calling contexts.
- Our approach avoids duplicating the JavaScript semantics in both an interpreter and a compiler, as we use a single compiler-runtime implementing JavaScript's semantics.
- The machine code generated for optimized traces seamlessly integrates with other machine code, including statically precompiled machine code, and code jitted at runtime. This enables difficult scenarios not found in previous tracing JITs, such as unified calling conventions, garbage collection, and exception handling, without having to exit the trace.

- We present a number of novel optimizations for traces, such as store-load elimination (or sinking), and speculative guard strengthening.

The rest of the paper is organized as follows. Section 2 gives a motivating example of a loop in C#, Section 3 describes the SPUR architecture and engine, Section 4 describes our various versions of just-in-time compilers and transitions between them, whereas Section 5 focusses on our trace recording and optimizations. Section 6 shows how we compile JavaScript to CIL and Section 7 re-examines our motivating example, which we re-write in JavaScript, showing how it is translated and eventually optimized. Section 8 contains our performance evaluation, and Section 9 discusses related work.

2. Example 1: A Loop in C#

Consider the following example, which is written in C#, but shows the basic problem of the "abstraction tax" paid in JavaScript programs, where all values are (potentially boxed) objects. An interface IDictionary describing a general mapping from objects to objects is used in method ArraySwap with the implicit assumption that it is a dense array, mapping integers to doubles.

```
interface IDictionary {
  int GetLength();
  object GetElement(object index);
  void SetElement(object index, object value);
}
...
void ArraySwap(IDictionary a) {
  for (int i = 0; i < a.GetLength() - 1; i++) {
    var tmp = a.GetElement(i);
    a.SetElement(i, a.GetElement(i + 1));
    a.SetElement(i + 1, tmp);
  }
}
```

Furthermore, let us assume that ArraySwap is most commonly used with the following implementation of IDictionary, which can only store floating point values in a dense array:

```
class DoubleArray : IDictionary {
  private double[] _elements;
  ...
  override int GetLength() {
    return _elements.Length;
  }
  override object GetElement(object index) {
    return (object)_elements[(int)index];
  }
  override void SetElement(object index,
                           object value) {
    _elements[(int)index] = (double)value;
  }
}
```

Without tracing through the virtual calls in ArraySwap to the interface methods, the arguments and results of GetElement and SetElement are continually boxed, type-checked for proper types, and unboxed. With tracing, only a few simple checks, called *guards*, need to remain in place which ensure that all explicit and implicit branching decisions that were observed during tracing will repeat later on. If not, then the optimized code will jump back into unoptimized code. Here, we need guards to ensure that the actual dictionary is not null and actually has type DoubleArray, that a._elements is not null, and also that i is a valid index, and fulfills the loop condition. All virtual method calls can be inlined, all boxing, type-checks and unboxing code eliminated. Here, most guards can be hoisted out of the loop, and code similar to the following is produced by SPUR for the loop, preceeded by the hoisted guards:

```
if (a == null ||
    a.GetType() != typeof(DoubleArray))
{ ... /* transfer back to unoptimized code */ }
var elements = a._elements;
if (elements == null)
{ ... /* transfer back to unoptimized code */ }
var length1 = elements.Length - 1;
while (true) {
  if (i < 0 || i >= length1)
  { ... /* transfer back to unoptimized code */ }
  double tmp = elements[i];
  elements[i] = elements[i + 1];
  elements[i + 1] = tmp;
  i++;
}
```

We will revisit this example in Section 7, where we see in more detail how a similar method written in JavaScript source gets compiled, traced, and optimized by SPUR.

3. SPUR

The basic input program to run on top of SPUR is any program compiled to Microsoft's Common Intermediate Language CIL [14]. CIL is a stack based intermediate language with support for object-oriented features such as inheritance, interfaces, and virtual dispatch. It also supports value types (like C structs), pointers to object fields and to locals on the runtime stack, and method pointers (usually wrapped in CIL delegate types) with indirect method calls. Instructions operate on the evaluation stack by popping operands and pushing results. CIL is the target for languages such as C#, VisualBasic, F#, and in this paper, JavaScript.

3.1 Architecture

Figures 1 and 2 show the architecture of SPUR configured to run JavaScript code. JavaScript source code is compiled by the JavaScript compiler which produces CIL. The SPUR engine jits this CIL to profiling code that we run on the *Bartok* runtime. Bartok is a managed runtime and static compiler developed at Microsoft Research. When profiling

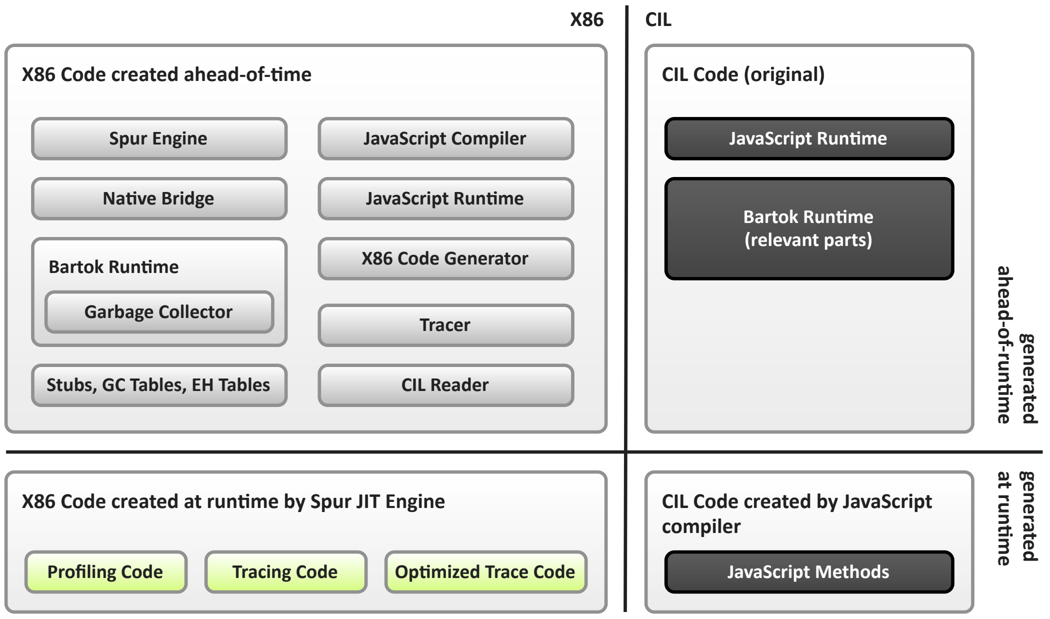

Figure 1. Code Quadrants of SPUR.

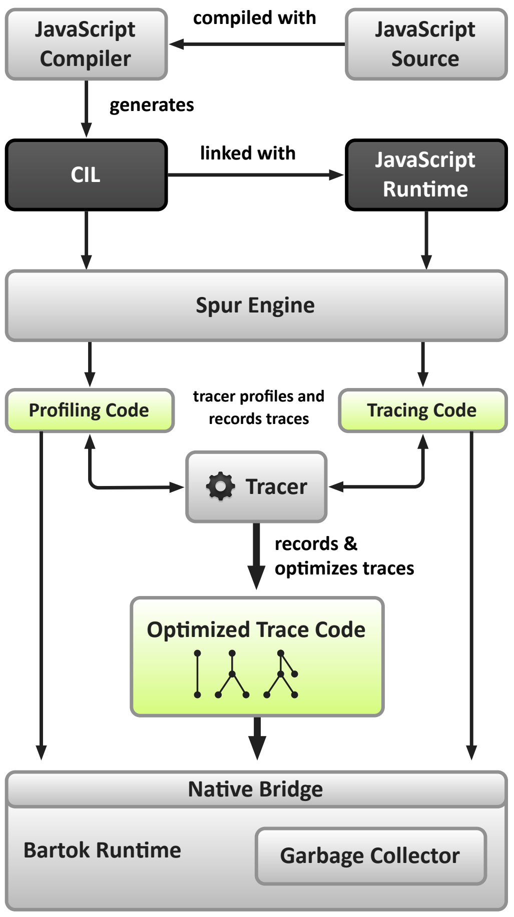

Figure 2. The SPUR architecture.

determines a hot path, tracing code is jitted and executed. The recorded trace is then optimized and jitted.

The native bridge in the figures refers to linking information that is generated ahead-of-time via the Bartok compiler. This linking information consists of runtime entry-points as well as object layout information of runtime types. It is used in order to make dynamically compiled X86 interoperate with ahead-of-runtime produced X86. In the following, we often refer to ahead-of-runtime code as *native* code.

Figure 1 elucidates what code is produced ahead-of-runtime (native code), what code is produced dynamically (jitted code), and whether the code is CIL or X86. The upper part of the figure refers to code produced ahead-of-runtime, while the bottom of the figure shows code produced during SPUR's execution. The left side of the figure shows code in the form of X86 binary instructions (our native target), while the right side of the figure shows code in the form of CIL instructions. The four quadrants thus give rise to all combinations of ahead-of-runtime or dynamically generated, X86 or CIL code.

Examples of ahead-of-runtime code in X86 form are the JavaScript compiler code, the garbage collector (GC), and our X86 code generator itself. An example of ahead-of-runtime code in CIL form is the CIL for the JavaScript runtime, produced by compiling our C# sources to CIL.

CIL code generated at runtime consists of the code produced by the JavaScript compiler for JavaScript source code. X86 code generated at runtime is code produced by our JIT for CIL instructions from the right-side of the architecture, both from CIL generated ahead-of-runtime and CIL generated by the JavaScript compiler at runtime.

3.2 Runtime

Our runtime for CIL code consists of an adaption of the production generational "mark and sweep" garbage collector of the Microsoft CLR, and the Bartok runtime. The garbage collector maintains three separate generations, and a separate large-object heap; all write accesses of references require a write barrier.

The Bartok system also provides a static compiler developed at Microsoft Research, compiling CIL to X86 code ahead-of-time.

Most of the Bartok runtime, including the base class libraries is written in C# itself and compiled to CIL via C#'s compiler and then to X86 code via the Bartok compiler. The same holds for the JavaScript compiler and runtime. As Figure 1 shows, we actually have both X86 and CIL versions of the Bartok runtime code and the JavaScript runtime code available at runtime. Having the CIL form may seem unnecessary at first as we can just call directly into the X86 form. However, in order to optimize traces that span calls into the runtime, we use the CIL of runtime methods and dynamically compile them into a form that contains profiling counters or callbacks to our trace recorder. This way, the recorded trace contains CIL instructions of any inlined runtime methods along with CIL instructions from compiled JavaScript code. This is crucial in order to optimize traces that span runtime code, in particular for the JavaScript runtime, as its methods contain most of the specialization opportunities.

3.2.1 Data Representations

CIL objects (instances of classes) are represented at runtime using two header words, followed by the field representations. The header contains a pointer to the virtual dispatch table of the type and the other header word is used during garbage collection.

Value types, including all primitive types such as Boolean values, integers, and floating point values, are not represented as heap objects, but stored either in registers, the runtime stack, or as fields of other objects.

Pointers in our runtime are not tagged, meaning that there are no unions encompassing pointers and primitive types in the same storage location. The garbage collector depends on GC tables of each type to identify the offsets of managed pointers within the object layout.

Garbage collector tables also exist to identify the offsets of managed pointers within stack frames. Exception handlers are identified via exception tables identifying protected code regions and the appropriate handlers are found via lookup operations.

3.2.2 Code Representations

In a preprocessing step before jitting code, CIL is transformed into an equivalent representation where all implicit operand stack accesses have been made explicit, using auxiliary local variables. This simplifies SPUR's JIT, and the transfers into and out of optimized traces.

3.3 New Instructions

SPUR adds a few instructions to CIL to aid in tracing.

3.3.1 Trace Anchors

A trace anchor is an instruction representing a potential start point of a trace. Trace anchors are inserted in a preprocessing step for all targets of loop back-edges, as trace compilation is most profitable for loops. We also place trace anchors after all potential exit points from existing traces in order to combine multiple traces together into trace trees. Trace anchors are also inserted at entry points of potentially recursive methods.

3.3.2 Transfers

A transfer instruction is the only way optimized trace code may give control back to regular profiling code. A transfer instruction details the layout of nested stack frames that may have to be reconstructed, corresponding to methods that were inlined during tracing.

3.4 Execution Modes

At any point in time, a thread running under SPUR is either in *native mode* (running code pre-compiled with Bartok, e.g., the X86 code generator itself), or in one of the following JIT modes depicted in Figure 3:

- *Profiling*: runs unoptimized code with counters to identify hot loops and paths.
- *Tracing*: runs unoptimized code with call-backs to record a trace.
- *On-trace*: runs the optimized trace code.

The execution mode is uniquely determined by the kind of code pointed at by the program counter. Thus, no extra state for maintaining the mode is used.

A transition from profiling mode to tracing mode occurs when a hot loop or path is identified at a trace-anchor.

At trace anchors, the code can be patched to jump to optimized trace code. If such an anchor is reached while in profiling mode, the thread transitions to on-trace mode. During tracing mode, execution transitions back to profiling mode when one of the following cases occurs:

- The initial trace anchor is reached again. This means, we identified a loop trace.
- For traces starting from a loop anchor, we end the trace when it leaves the loop body. For nested loops within a single method, we consider the outer loop part of the inner loop as well, in order to trace entire loop nests.
- When a trace anchor is reached for which we already have an optimized trace.

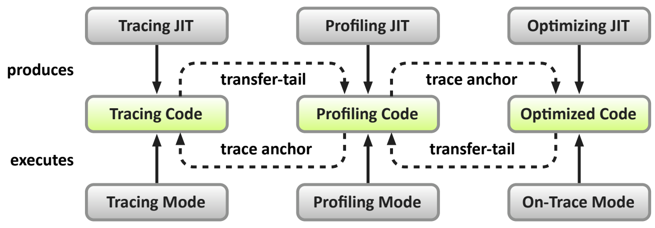

Figure 3. JIT version, code, and execution modes.

- When the trace length exceeds a configurable threshold (currently 100k instructions).
- When the stack depth through inlining exceeds a configurable threshold (currently 20).
- When an instruction throws an exception.

When a non-loop trace ends, we determine via heuristics if it is profitable to optimize the trace and patch it as a non-loop trace into the trace anchor.

When running optimized trace code, execution remains on-trace as long as all guards hold. (Guards arise from conditional control flow decisions that were recorded earlier during tracing.) Otherwise, a trace exit is reached, at which point the execution transfers back to profiling.

4. Just-in-Time Compilers

SPUR uses different JITs to produce different code versions. The JITs are not based on the production Microsoft CLR JIT, but have been developed from scratch to enable efficient transitions between the different code version. The current implementation of SPUR targets the 32-bit X86 architecture.

All JITs are derived by subclassing a basic visitor pattern over CIL instructions. In addition to the three main JITs depicted in Figure 3, there is one additional specialized JIT called the *transfer-tail* JIT. Its job is to bridge the execution from an arbitrary instruction within a block from tracing or on-trace mode to the next safe point to transfer to the profiling code.

We use an abstract JIT base class without a register allocator from which the tracing, transfer-tail, and an abstract base class for JITs with a register allocator are derived. From the latter, the profiling JIT and the optimizing JIT are derived, both of which perform register allocation.

The register allocator is very simple. We distinguish basic blocks with multiple predecessors as *head basic blocks* from basic blocks with a single predecessor. At all head basic block boundaries, all registers are spilled. We don't allocate registers across head basic blocks in order to simplify the transition into and out of profiling mode. For each basic block, the register allocator performs one backward scan to determine if generated values will be used. Then it performs a forward register allocation, allocating up to three callee-save X86 registers (EBX, ESI, EDI). EBP serves as the frame pointer, ESP is the stack pointer, and the caller-save X86 registers (EAX, ECX, EDX) are used as scratch registers in the compilation of the individual CIL instructions. The X86 floating-point stack is used to allocate floating point values in registers. Write barriers for the garbage collector are inlined within loops.

4.1 Code Transitions

The virtual dispatch table of any object at runtime always points to the native version of a method (ahead-of-time X86, see Figure 1). This invariant is trivially maintained as SPUR does not support dynamic type creation at the moment. In the future, when types are created dynamically, their dispatch tables would point to profiling versions of the methods.

Delegate objects represent closures and consist of a method pointer and an optional target object. Delegates created in native mode point to the native mode version, whereas delegates created in any JIT mode always point to the profiling version.

The initial execution mode is native. Transitions from native mode can only occur via indirect calls through method pointers, and thus only to profiling mode. Otherwise execution remains in native mode.

Transitions from JIT modes to native mode occur only through indirect method calls through delegates created in native mode, or when calling methods marked with [NativeCall] attributes. Such annotations are useful to prevent tracing through runtime methods that were written in C, contain unsafe pointer manipulations, or whose trace expansion would be non-beneficial.

Execution may transfer from tracing or optimized trace code back to profiling code at any instruction in a basic block. Instead of persisting and restoring register allocation decisions at every instruction, we only do so at head basic block boundaries; we use *transfer-tail* code jitted without register allocation to execute the instructions up to the start of the next head basic block. Thus the actual transitions from tracing and on-trace mode in Figure 3 goes through a small portion of transfer-tail code.

4.2 Profiling JIT

For each dynamically generated CIL method an X86 stub is put in place. The purpose of the stub is to determine if an X86 version of the method has been compiled and is ready to call. If not, the stub invokes the profiling JIT (when in profiling mode) to translate the CIL to X86.

The profiling JIT is a very simple and direct JIT performing no optimizations, but it does employ a register allocator. The task of the profiling JIT is to allow us to start running dynamic code immediately and to emit profiling counters to identify hot paths.

For each trace anchor, the profiling JIT inserts a counter variable. For loop-related trace anchors, these counters are local variables. Counters for individual trace exits and counters for trace anchors for tracing potentially recursive method calls are global. A counter decrement operation and a conditional branch is inserted at each trace anchor. The counter starts at a certain threshold (default: 3 for local counter, and 317 for global counters); when it reaches zero, execution transitions to tracing mode, which involves branching to the corresponding basic block in the tracing code version of the current method. The tracing code version is emitted by the tracing JIT.

Direct calls always invoke the profiling code of the target method, except if the target method is marked [NativeCall]. In the latter case, execution proceeds in native mode until the call returns, at which point it reverts back to profiling. At virtual call sites, the profiling JIT produces code that finds the profiling version of the target method via table lookups. To amortize the lookup cost, we employ inline caches in the code. Finally, at indirect call sites through method pointers, the method pointer is directly invoked. According to our invariants, the method pointer points to native code (in which case we temporarily transition to native mode), or the pointer points to the profiling code version of the method.

Note that we jit profiling versions of methods in the runtime using the CIL versions depicted in the upper right quadrant of Figure 1. As a result, profiling might determine the start of a hot loop or path *inside* the runtime.

4.3 Tracing JIT

In tracing mode, we need X86 code that performs call-backs to our trace infrastructure in order to record the sequence of CIL instructions being executed on the current path. We call this code tracing code and it is emitted by the tracing JIT (Figure 3).

The tracing JIT handles method calls similarly to the profiling JIT but with the goal of remaining in tracing mode. At direct call sites, the JIT simply emits a direct call to the tracing version of the target method, unless the method is marked [NativeCall]. At virtual call sites, we determine the target method via a table lookup and use inline caches to amortize the cost. For indirect calls via method pointers, the tracing JIT needs to do more work than the profiling JIT. It uses the method pointer as an index into a table of profiling code methods. If found, the JIT targets the corresponding tracing JIT version of the method. Otherwise, the method pointer must be a native method and it is called directly, transitioning temporarily out of tracing mode until the return from the native method.

Thus, by default, traces extend through runtime methods, except when the target method is marked as [NativeCall] or cannot be determined due to an indirect call with a native code target.

The ability to selectively inline runtime methods into traces is the major advantage of our approach and enables on-demand specialization of complicated runtime methods to particular calling contexts.

4.4 Optimizing JIT for On-Trace Code

Via call-backs, the tracing code can record a trace of CIL instructions. When a profitable trace has been recorded, it is optimized by our optimizing trace compiler, which produces a special CIL method for the trace. The optimizing JIT for on-trace code compiles this trace method to machine code. A trace method is special in several ways:

- It inherits the arguments and local variables of the method where the trace started.
- It does not contain a regular return instruction, and its control-flow must not allow the termination of the method via an exception. Instead, execution may only leave this method via a special `transfer` instruction.

Even though the code generated by the Optimizing JIT has these unusual entry and exit semantics, the result is a seamless integration into the existing calling conventions and stack layouts, garbage collector and exception handling tables.

Because our trace recorder inlines method calls aggressively and a trace may exit in the middle of many inlined calls, a `transfer` instruction must recover all of the stack frames of all active inlined methods of the current trace. This involves reconstructing the local variables and architecture dependent state, such as frame pointers, return addresses, registers, etc.

When transfering out of the trace method, transfer-tail code is employed. Once machine code for the optimized trace code has been generated, a jump to the resulting X86 code is patched into the trace anchor code of the profiling code, overriding the original counting logic.

The optimizing JIT uses the standard register allocator. Note that a trace tree with a loop has exactly one head basic block which acts as the loop head. Thus, in this case, register allocation happens globally over the entire tree.

4.5 Transfer-Tail JIT

Transfer-tail code enables transitions from tracing or optimized trace code back to profiling code. At the transfer source, all registers are spilled onto the stack. Transfer-tail code itself does not use any register allocation, and so execution can start at any instruction. At the beginning of each head basic block, control transfers back to regular profiling code. Transfer-tail code is jitted on demand.

4.6 Design Implications

This section briefly touches on a few consequences of our design.

4.6.1 Stack Reconstruction

At trace exits, we have to reconstruct the stack layout of inlined active methods. This reconstruction is complicated by the fact that CIL supports managed pointers into the runtime stack, e.g., for passing the address of a local to another method. Consider the following example method:

```
void SwapWithSmaller(ref int x, int[] data) {
  for (int i=0; i< data.Length; i++) {
    if (data[i] < x) {
      int temp = data[i];
      data[i] = x;
      x = temp;
      return;
    }
  }
}

void Use(int[] data) {
  int a = 55;
  SwapWithSmaller(ref a, data);
}
```

SwapWithSmaller takes an address to an integer x and an array of integers and swaps the first element in the array that is smaller than the current value of x. The runtime stack for the call from within Use looks as follows:

		top-of-stack
SwapWithSmaller	x	&a
	data	*p*
		(fp,sp,...)
Use	a	55
	data	*p*
		(fp,sp,...)

Here, *p* refers to the address of the array and (sp,fp,...) refers to calling convention registers and saved registers. Note how x points to the stack location of a. If a trace is compiled that inlines the call from Use and the trace needs to be exited in the middle of the execution, then the exact stack configuration above must be produced.

The optimized trace code typically needs a fraction of the original stack space. E.g., it doesn't need duplicate locations for data and stack elements for calling conventions can also be saved.

If we naively optimize the stack frame of the trace we have the problem that the address of local a in the optimized frame is *different* from the address in the configuration above, which implies that we have to adjust all locations containing pointers to a, such as x. Worse, the CIL execution model assumes and requires that stack locations do not move at runtime. Thus, such an approach isn't viable.

To address this issue, our trace code uses stack frames that are as large as the reconstructed stack frames it might need to produce on trace exit. This allows us to place stack locals whose address is taken into the exact position they occupy in the reconstructed frames. The resulting optimized frame simply contains gaps of memory that may remain unused until a trace exit forces the reconstruction of the stack.

4.6.2 Stacks of Trace Contexts

Allowing native calls from within our jitted code results in interesting runtime configuration that may not be immediately apparent. Recall that indirect calls via delegates from native contexts may enter profiling code if the delegate was constructed in jitted code. As native contexts can be invoked from any mode (profiling, tracing, on-trace), it is possible to build up a stack of distinct trace contexts at runtime, separated by native stack frames as shown below:

	⋮
context 3	profiling
	native
	tracing
context 2	profiling
	native
	on-trace
context 1	profiling
	native

In principle any combination of active modes is possible at runtime, in particular, there might be multiple contexts actively recording a trace. In order to simplify our infrastructure, we currently allow only a single trace to be recorded at a time. Thus, if a nested context wants to start tracing, but an outer context is already recording a trace, we remain in profiling mode instead of entering tracing mode.

5. Trace Recording and Optimization

The trace tree optimizer performs many standard optimizations, such as constant folding, common subexpression elimination, arithmetic simplifications, and dead code elimination. In addition, we perform some novel optimizations which we describe in more detail.

5.1 Simplifying Assumptions

The SPUR infrastructure is not a full implementation of Microsoft's .NET platform. In particular, we don't support the following core features:

- Runtime reflection is limited to the generation of new methods. In particular, new types cannot yet be generated dynamically and metadata cannot be inspected at runtime.
- We do not yet support multiple user threads, which simplifies stub patching and counter management.
- We do not yet support tracing of code involving
  - typed references, which are used to realize C-like vararg calling conventions (ordinary managed pointers realizing interior pointers are supported),
  - multi-dimensional arrays (jagged arrays are supported), or
  - unsafe code.

The addition of these missing features does not require a change in the architecture and we don't anticipate any major issues. The addition of threading will impact certain optimizations as mentioned below.

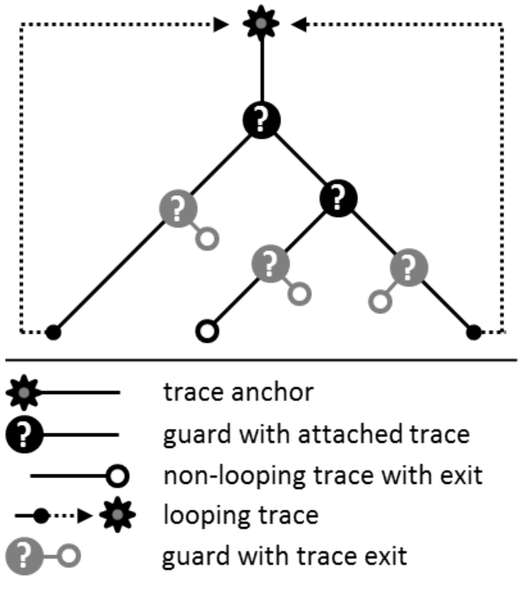

Figure 4. Trace Trees in SPUR.

5.2 Growing Trace Trees

Figure 4 illustrates the structure of trace trees in SPUR. A trace tree is created when the first trace is recorded for a particular trace anchor in the profiling code. Along the recorded trace, guards make sure that future executions will take the previously recorded path. If not, the trace is exited. After optimized trace tree code has been jitted, a jump to the optimized code is patched into the original trace anchor in the profiling code. Later, when the optimized trace tree code is executed, a particular guard may fail frequently. In that case, a trace for the continuation after the corresponding trace exit is recorded; it is attached to the previously recorded trace at the previously failing guard, which now selects between the old and the new trace continuation. All associated recorded traces are composed to form the trace tree. Individual traces may or may not loop back to the trace anchor. SPUR does not currently support nested trace trees.

Up to five tracing attempts are made from each trace anchor; if each attempt is aborted, e.g. because execution left the relevant loop body, or an exception is thrown, or because a non-loop trace was not profitable, then the trace anchor is retired, and the trace anchor code is patched with X86 no-operation (NOP) instructions.

During trace recording, some trivial optimizations, in particular constant folding, are already applied to reduce the amount of collected data.

5.3 Trace Intermediate Representation

We represent recorded traces in an SSA form. Whenever possible, indirect memory accesses, which can occur in CIL via managed pointers, are resolved. We represent updates of value types and selection of fields from values types in SSA form as well. Most instructions which can throw an exception implicitly—a null-dereference, a bounds check at an array access, or an arithmetic exceptions—are split into *guard* instructions to ascertain that the operation won't fail, and the actual operation without the check. All guard instructions are annotated with the live stack state that needs to reconstructed on exiting the trace.

However, some instructions for which the check for the exceptional case would be too complicated or redundant, e.g. a multiplication with overflow check, or a call to a native method, are instead annotated with live stack state to reconstruct inlined stack frames, so that they can be reconstructed when an exception is actually thrown. Besides spilling of registers around the protected region with exception handling, there is no overhead for exception handling.

5.4 Guards

A recorded trace typically contains many guards, i.e., instructions corresponding to conditional branches and other conditions which would force a deviation of the execution from the recorded trace (e.g., via an exception). A typical place where a guard is needed is at a virtual call instruction. The guard makes sure the target method is the same as the one recorded during tracing.

We perform aggressive guard optimizations, which are crucial for gaining performance. The main optimization performed is evidently *guard implication*, i.e., guards that have been checked earlier and are thus redundant don't need to be checked again. In simple cases, this corresponds to common subexpression elimination combined with constant propagation and dead code elimination. SPUR can also detect non-trivial implications by analyzing the relationship of integer equalities and inequalities.

To further reduce the number of guards and possibly hoist them out of loops, we perform novel *guard strengthening* optimizations. These optimizations are speculative (but safe) in that they might cause a trace to be exited earlier than absolutely necessary. Guard strengthening identifies guards implied by later guards and then strengthens earlier guards to the stronger guard occurring later in the trace. This change makes the later guard redundant but might cause execution to leave the optimized trace at the earlier guard rather than the later guard. Leaving a trace earlier isn't really a problem, as the trace would be left anyway in the same loop iteration. There is a small performance loss in the final iteration where the trace is left, but no semantic problem. If a particular trace exit is triggered frequently, SPUR will record a trace starting from that trace exit. If the guard that caused the trace exit was involved in speculative guard strenghtening, then it might be the case that the recorded trace initially takes an already known path after the failing guard through the trace tree, and only later deviates at the later guard that was made redundant by the speculative guard strengthening. SPUR detects this scenario, and then only attaches the new part of the trace at the later guard. Once a new trace has been attached, speculative guard strengthening will no longer be performed at or above the branching point.

5.5 Hints and Guarantees

SPUR supports annotations in the traced code to guide tracing and enable further optimizations: Some JavaScript runtime methods are annotated with a [TraceUnfold] attribute, that indicates that contained loops should be unfolded in outer traces, and no local trace trees should be constructed. This annotation is used mainly on those runtime methods that implement traversals of JavaScript prototype chains in a loop, as these prototype chains tend to be stable at particular callsites.

SPUR also supports annotations to propagate safety guarantees of the JavaScript runtime. In particular, we use [DoNotCheckArrayAccesses] attributes to eliminate bounds checks on certain JavaScript runtime functions written in C# which access array elements holding object properties which are guaranteed to be present according to the JavaScript type of the object.

5.6 Store-Load Propagation

We use an alias analysis to perform a store-load propagation along the trace to avoid (or delay) writes and reads to the heap and the runtime stack. Recall that CIL allows managed pointers into the stack and call-by-reference parameters. Store-load propagation enables short-circuiting such indirect stores and loads when inlining methods with by-ref parameters.

When applied to heap locations, this optimization relies on the single-threaded assumption of our current infrastructure. In the presence of threads, the alias analysis will need to restrict this optimizations according to the CIL memory model, and it can only be applied fully to thread local data (typically allocated on the trace). Note that the optimization applied to stack locations remains fully applicable even in the presence of multiple threads.

Store-load optimizations are often able to eliminate temporary object allocations in a trace. Object fields may be kept in registers, and the allocation of value types may be delayed until a trace exits.

5.7 Invariant code motion

We first determine which local variables are guaranteed to not change during execution along any looping path of a trace tree. We mark those memory locations as invariant. All pure non-exception-throwing computations which depend only on invariant values are marked invariant as well. All invariant computations are hoisted.

Invariant code motion is extended to guards, load operations from the heap, and memory allocations as well: If a guard only depends on invariant values, and if it appears somewhere along *all* looping paths of the trace tree, then the guard is hoisted. This might disable non-looping paths as a result (similarly to guard strengthening), but optimizing the looping paths is more important.

Similarly, if a load operation from the heap only depends on invariant references and indices, and it appears somewhere along *all* looping paths of the trace tree, and any required implicit null and/or type check has been already hoisted, and no looping path can possibly overwrite the memory to be loaded, then the load operation is hoisted. Memory allocations which do not escape are also hoisted; in the case of an array, the array size must be invariant. In that case, at the end of each path that may write to a field or element of a hoisted memory allocation, an instruction is inserted to fill all fields or elements of the new object with a zero bit pattern to restore the initial state of the object as if it were re-allocated.

5.8 Loop Unrolling

For looping trace trees with limited branches, we aggressively unroll the loop. Our heuristic tries to keep the size of the unrolled trace tree reasonable, as it potentially explodes with the number of branches in the trace tree. We only unroll the most frequently taken branches.

5.9 Delayed Computations in Transfers

Besides optimizing the trace, the optimizer is responsible for emitting guard checks at each potential trace exit, and transfer instructions to exit the trace. When a guard fails, some values used in the transfer instruction to reconstruct nested stack frames might still have to be computed, as they were not used in the main execution of the optimized trace code. All pure and non-exceptional computations might be delayed until a transfer requires them.

6. JScript Compiler

On startup, SPUR starts executing the JavaScript compiler, which in turn reads the JavaScript source, compiles it to CIL in memory. Then SPUR jits the CIL to machine code, which finally gets invoked.

Our JavaScript compiler evolved from the JScript.NET code base [1], a JavaScript compiler for the standard .NET platform. Before targeting SPUR, we put a fair amount of work into the compiler and the JavaScript runtime to increase performance when running on Microsoft's standard .NET platform.

6.1 Argument and Variables of Primitive Types

Arguments and local variables of functions are specialized to be of type bool, int, or double, if the static type inference can prove that the values of these arguments or variables will always be in the range of the specialized type.

6.2 Function Calls

When a function is called that was defined in JavaScript code, the function is type-specialized according to the statically known argument types at the call site. A cache maintains jitted variations of a function, one per type-specialization.

When the function being called is implemented as a native runtime method in C#, rather than as a JavaScript function, then the native function cannot be specialized and jitted to suit the call site. Instead, an adaptor stub is generated to convert the call site arguments to the type signature expected by the native method.

6.3 Object Representation

JavaScript objects are represented as CIL objects, deriving from a common base class. In this CIL base class, one field holds a reference to a type object, and another field holds a reference to an array of property values. (Fields of an object in JavaScript are called properties.) The type object maps property and function names to indices into the property array. When a JavaScript object gets a new property, or loses a property, its type object is replaced with another type object and its property value array is resized (if need be).

When a JavaScript object is used as a hash table from strings to values, a naive version of this scheme would generate a new type object every time a new key is added. To avoid this proliferation of types, any JavaScript object whose (non-array index) properties are created via the [] syntax gets a mutable (unshared) copy of its type object and thus all further additions of properties to the object are done in place in the mutable type object. In effect, the type object itself becomes a non-shared hash table.

JavaScript arrays are derived fom JavaScript objects that in our implementation additionally have a native field that tracks the types of array elements (bool, int, double or *any*). Depending on the element type of the array one of four additional fields is initialized with a native array with the appropriate element type. JavaScript arrays can be either dense or sparse. When sparse, the first n entries are represented densely in the native array and the rest are stored in a specialized hash table. Note that whenever the name of a property is an array index, the property is not added to the type object. It either goes to the native array or the specialized (per instance) hash table. If the object is not a JavaScript array, it only maintains the specialized hash table.

6.4 Lookup Implementation

Every type object is a mapping from property name to property offset. Every code location where an object property is accessed has a dedicated "cache" object[1]. When control reaches the access, the current type of the object is extracted and compared to the type stored in the cache. If there is a hit, the cache contains the corresponding property offset. If there is a miss, the type object is asked to map the property name to an offset and the cache is updated with the type object and offset. In this context "property offset" means index of the value in the array of property values.

[1] In the current SPUR implementation, these caches are two-way, out-of-line.

6.5 Eval

Since eval needs to be able to inspect and modify the stack frame of a function, a function containing eval cannot store its locals in true locals, but uses a heap allocated "activation object" to represent locals as properties. These properties are modeled and accessed exactly like the properties of normal JavaScript objects.

The code generated by eval is turned into a function whose (heap allocated) scope chain is the same as the calling function. Global code is also represented as a function with a heap allocated activation object. Thus, global variables live in properties of an implicit global JavaScript object, whose type evolves just as the type of all other JavaScript objects.

Note that other constructs, such as nested functions and with statements also cause a function to allocate its locals on the heap.

6.6 Static Analysis

We use abstract interpretation on the JavaScript code to perform sound type-specialization by inferring ranges for atomic local variables and function arguments [22]. Expressions that involve constants or local variables and function arguments that are type-specialized are compiled into typed CIL, rather than into calls to polymorphic helper functions. This results in shorter traces and fewer guards.

6.7 Boxed Values

JavaScript object properties, elements of polymorphic JavaScript arrays, and local variables and arguments whose type could not be restricted by the static analysis, hold boxed values. However, in the context of SPUR's runtime, boxed objects are not allocated on the heap. Instead, they are stored in a 16-byte value type (in a 32-bit environment), which is usually passed around by-reference (and sometimes by-value). This value type effectively implements a union via an explicit struct layout in C#:

```
[StructLayout(LayoutKind.Explicit)]
struct ValueWrapper {
  [FieldOffset(0)] object wrappedObject;
  [FieldOffset(4)] TypeCode typeCode;
  [FieldOffset(8)] bool wrappedBool;
  [FieldOffset(8)] char wrappedChar;
  [FieldOffset(8)] double wrappedDouble;
  [FieldOffset(8)] int wrappedInt;
  [FieldOffset(8)] uint wrappedUInt;
  [FieldOffset(8)] long wrappedLong;
  [FieldOffset(8)] ulong wrappedULong;
}
enum TypeCode {
  Empty, Int32, Double,
  String, Object, DBNull,
  Boolean, Char, UInt16, UInt32, Missing,
}
```

```
void arraySwap(ref ValueWrapper a) {
  for (double i = 0.0;
       Less(i,
            Minus(GetProperty(c, a, "length"), 1.0));
       i += 1.0) {
    ValueWrapper tmp = GetElement(a, i);
    SetElement(a, i, GetElement(a, i + 1.0));
    SetElement(a, i + 1.0, tmp);
  }
}
```

Figure 5. C# corresponding roughly to JavaScript example

6.8 Tweaking the JScript Compiler for Tracing

We found a few idioms in the generated code of the JScript compiler that needed to be changed in order to get the best performance out of the TJIT.

- Our caches for property lookups are realized as objects which live on the heap. However, only persistent data should be stored in such a cache object, and all temporary data involved in cache lookups should be passed via the stack.
- We use a high-water mark for non-empty slots in arrays; this often allows hoisting checks for empty elements out of loops.
- General guidelines to improve performance of object oriented programs apply here as well, in particular avoiding casts to non-sealed classes.

7. Example 2: A Loop in JavaScript

In this section, we will revisit the C# example from Section 2, which we now rewrite in JavaScript as the source language:

```
function arraySwap(a) {
  for (var i = 0; i < a.length - 1; i++) {
    var tmp = a[i];
    a[i] = a[i + 1];
    a[i + 1] = tmp;
   }
}
```

Static analysis determines that i is a number, but a could be any object (it doesn't have to be an array), so the type of a.length is unknown statically, so the arithmetic expression a.length-1 as well as the comparison against i has to be computed with boxed values. The type of the array elements is also not known.

The CIL code generated by our JavaScript compiler is roughly the code corresponding to the C# in Figure 5. Note that JavaScript values of unknown types are stored in the value type called ValueWrapper shown in Section 6.7. We don't write out the implicit boxing of double values to ValueWrapper nor do we show that value wrappers are actually passed by reference. The JavaScript runtime helper methods Less and Minus implement the corresponding JavaScript semantics for $<$ and $-$; GetProperty implements the retrieval of an object property; a cache object c stores information about the index of the property given a particular JavaScript object type; GetElement and SetElement implement array accesses.

```
--------- hoisted loop invariant code
          guard a Cne null
    (2) = a.typeCode
          guard (2) Ceq TypeCode.Object
    (6) = a.wrappedObject
    (7) = (6) as MS.JScript.ObjectInstance
   (23) = (6) as MS.JScript.ArrayInstance
          guard (6) Ceq (23)
          guard (6) Cne null
   (12) = (6).type
   (13) = c.type1
          guard (12) Ceq (13)
   (15) = c.dynamicPropertyOffset1
          guard (15) Clt 0
   (17) = c.property1
          guard (17) Cne null
   (21) = call (17).System.Object::GetType()
          guard (21) Ceq MS.JScript.ArrayLengthProperty
   (27) = (6).len
   (28) = Conv_R_Un (27)
   (34) = (28) Sub 1.0D
          guard (7) Cne null
   (58) = call (7).System.Object::GetType()
          guard (58) Ceq MS.JScript.ArrayInstance
   (62) = (7).denseArrayLength
   (65) = (7).elementTypeCode
          guard (65) Ceq TypeCode.Double
   (68) = (7).indexOfFirstMissingEntry
   (71) = (7).doubleArray
  (143) = (6) as System.String
          guard (143) Ceq null
   (69) = (68) Min_Un (62)
--------- loop body
          guard i Clt (34) // exit1
   (60) = Conv_I4 i
          guard (60) Clt_Un (69) // exit2
   (72) = (71) [(60)]
  (121) = i Add 1.0D
  (127) = Conv_I4 (121)
          guard (127) Clt_Un (69) // exit3
  (139) = (71) [(127)]
          (71) [(60)] = (139)
          (71) [(127)] = (72)
   (81) = update default(ValueWrapper).wrappedDouble=(72)
   (82) = update (81).typeKind = TypeCode.Double
   (83) = update (82).wrappedObject = null
          loop tail state: i = (121), tmp = (83)
```

Figure 6. The trace intermediate representation.

```
loop:
    //      guard i Clt (34)
    fld     qword ptr [ebp-1Ch]
    fld     qword ptr [ebp-17Ch]
    fucomip st,st(1)
    jp      exit1
    jbe     exit1
    fstp    st(0)
    //      (60) = Conv_I4 i
    cvttsd2si ebx,mmword ptr [ebp-1Ch]
    //      guard (60) Clt_Un (69)
    mov     esi,dword ptr [ebp-16Ch]
    cmp     ebx,esi
    jae     exit2
    //      (72) = (71) [(60)]
    mov     edi,dword ptr [ebp-164h]
    fld     qword ptr [edi+ebx*8+0Ch]
    //      (121) = i Add 1.0D
    fld     qword ptr [ebp-1Ch]
    fadd    qword ptr ds:[3D80008h]
    fstp    qword ptr [ebp-194h]
    //      (127) = Conv_I4 (121)
    mov     dword ptr [ebp-180h],ebx
    cvttsd2si ebx,mmword ptr [ebp-194h]
    //      guard (127) Clt_Un (69)
    cmp     ebx,esi
    jae     exit3
    //      (139) = (71) [(127)]
    fld     qword ptr [edi+ebx*8+0Ch]
    //      (71) [(60)] = (139)
    mov     ecx,dword ptr [ebp-180h]
    fstp    qword ptr [edi+ecx*8+0Ch]
    //      (71) [(127)] = (72)
    fst     qword ptr [edi+ebx*8+0Ch]
    //      loop tail state: i = (121), tmp = (83)
    fld     qword ptr [ebp-194h]
    fstp    qword ptr [ebp-1Ch]
    //      (81)=update default(.).wrappedDouble=(72)
    //      (82)=update (81).typeKind = Double
    //      (83)=update (82).wrappedObject = null
    lea     ebx,[ebp-2Ch]
    fst     qword ptr [ebx+8]
    mov     dword ptr [ebx+4],2
    xor     eax,eax
    mov     dword ptr [ebx],eax
    fstp    st(0)
    //      book-keeping how often loop gets executed
    inc     dword ptr [ebp-12Ch]
    jmp     loop
```

Figure 7. The optimized trace loop body X86 code.

The unoptimized trace recorded for this loop, inlining all helper methods, has 210 instructions, including guards, and is not shown.

After optimization, 37 loop invariant instructions and 13 loop variant instructions remain, some of which can be delayed until an exit is triggered. Figure 6 shows the optimized trace intermediate representation, omitting delayed instructions. In the first part, the hoisted loop invariant computations are shown. Numbers in parentheses, e.g. (2), represent SSA values that are defined along the hoisted or loop body code. The instructions shown include field accesses ((x) = (y).f), guards (guard (x) op (y)) where the operator op can be equality (Ceq), disequality (Cne), inequality (Clt for signed integers/ordered floating point values, Clt_Un for unsigned integers/unordered floating point values), type casts that yield null on failure ((x) as T), calls to native methods (call (x).f()), conversion from unsigned integers to floating point (Conv_R_Un), conversion from floating point to signed integer (Conv_I4), arithmetic operations ((x) op (y)) where the operator op can be subtraction (Sub), addition (Add), minimum of unsigned integers (Min_Un), struct updates (update (x).f = (y)), and default values (default(T)), i.e. the struct value where are fields are filled with a zero bit pattern.

The Min_Un instruction was not present in the originally recorded trace, but resulted from a speculative guard strengthening optimization, in which two separate guards were combined. The accesses to type1, dynamicPropertyOffset1, and property1 are related to the access of a.length, which refers to the access of the array length property in this calling context.

Figure 7 shows the resulting machine code for the loop body. We show the loop body here without loop unrolling, but SPUR will actually unroll the loop body at least once. The generated machine code is not optimal, and there is room for further improvement: The value (60) is spilled and later reloaded, and the temporary tmp of type ValueWrapper is fully constructed and written back at the end of the loop; this is because SPUR currently does not perform a liveness analysis of local variables typed as ValueWrapper.

8. Evaluation

Figure 8 shows steady-state execution times for the SunSpider benchmarks in milliseconds, on a Intel Core2 Quad CPU, Q9650 @ 3.00 GHz, 8 GB Ram, Windows 7 Enterprise, 64-bit version, running 32-bit versions of all browsers. Each benchmark was embedded in a loop iteration 30 times; the lowest reported execution time is quoted in the table. As a result, jitting or tracing overhead is effectively not included in the quoted times. Note that we used abbreviated benchmark names to reduce the size of the table. V8 refers to V8 in Chrome 4.0.249.89 (38071), TM refers to TraceMonkey in Firefox 3.6, IE8 refers to Internet Explorer 8, which runs JavaScript with an interpreter without jitting. IE9 refers to an early version of Internet Explorer 9 (Platform Preview v1.9.7745.6019), which interprets JavaScript bytecodes with selective jitting. SPUR with tracing is the SPUR engine as described in this paper, SPUR without tracing is using SPUR's JIT, but without tracing, and SPUR-CLR is using

	V8	TM	SPUR with traces	SPUR CLR	SPUR w/o traces	IE8	IE9 (Pre-view)
3d-cube	13	25	14	42	63	112	37
3d-morph	19	36	10	26	47	103	42
3d-raytrace	15	43	16	50	72	165	24
acc-bin-tree	1	28	22	37	50	128	19
acc-fannkuch	10	47	15	84	174	280	13
acc-nbody	11	13	6	36	56	148	28
acc-nsieve	2	8	4	13	33	90	8
bitops-3bit	2	1	0	5	7	77	1
bitops-bits	6	7	6	6	8	79	5
bitops-and	6	2	1	15	12	186	3
bitops-nsiev	12	17	4	32	57	135	15
control-rec	2	31	9	17	22	106	2
crypto-aes	75	15	14	40	75	113	12
crypto-md5	7	2	2	16	20	70	10
crypto-sha1	7	3	2	15	21	70	11
date-tofte	20	56	49	67	92	165	42
date-xparb	26	69	66	65	67	161	41
math-cordic	13	21	6	48	66	143	2
math-partial	15	11	13	71	105	100	31
math-spectr	5	3	2	16	22	99	16
regexp-dna	12	37	616	528	585	204	31
string-b64	14	7	10	32	38	560	19
string-fasta	21	42	64	75	112	163	39
string-tagcl	22	45	123	109	160	129	43
string-unpac	43	58	382	294	397	124	68
string-valid	22	19	50	114	82	113	31

Figure 8. Steady-state execution times for SunSpider benchmarks in milliseconds

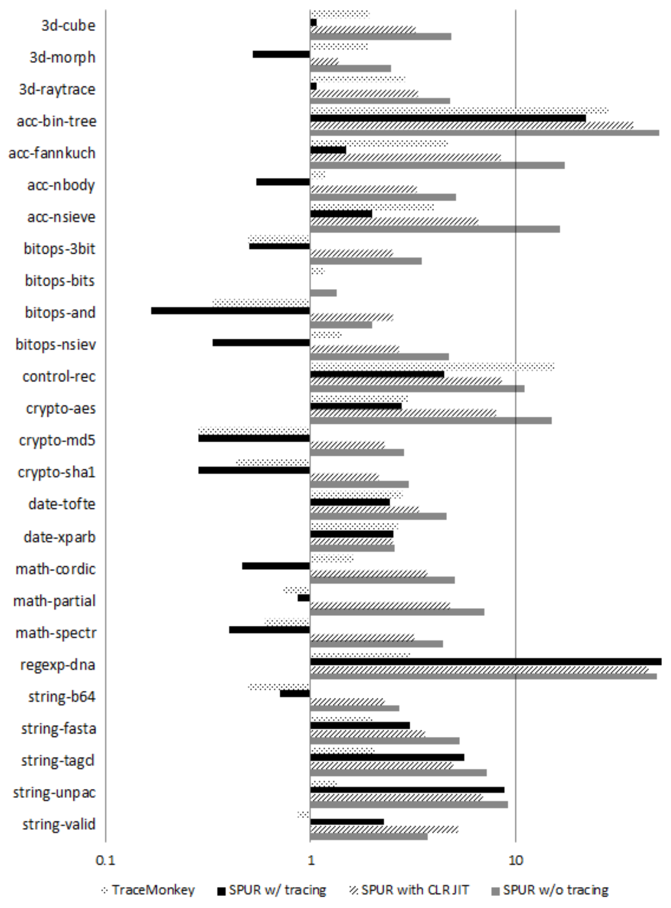

Figure 9. Normalized execution time over V8 4.0.249.89.

SPUR's JavaScript compiler, but using the production Microsoft CLR v3.5 JIT (and runtime) instead of SPUR's JIT, again without tracing.

Comparing SPUR without tracing to SPUR-CLR shows that the code generated by SPUR's JIT is on average 1.4 times slower than the CLR JIT. The same JIT is used when running SPUR with tracing, and yet — with exception of certain string-heavy benchmarks — it consistently performs faster than SPUR-CLR, faster than TraceMonkey, and quite often faster than V8. Excluding the string-related benchmarks regexp-dna, string-b64, string-fasta, string-tagcl, string-unpack, string-valid, then the code generated by SPUR with tracing only takes 59% percent of the time to execute compared to the code generated by TraceMonkey. When further excluding the recursion-heavy benchmarks acc-bin-tree, control-rec, then SPUR's code completes execution in 87% of the time it takes V8.

In its current implementation, SPUR does not have an optimized implementation of the string functions, and no optimized regular expression library. This prevents SPUR from performing better on the string-related benchmarks.

Figure 9 compares the execution time of TraceMonkey, SPUR with tracing, SPUR-CLR, and SPUR without tracing code against V8. Execution times are normalized against V8's time. Thus, bars extending to the right of 1, are factors slower than V8. Bars extending to the left of 1 are fractions of execution time of V8 code, thus faster.

The graph illustrates several points. First, code produced by SPUR without tracing runs only slightly slower than SPUR-CLR. This validates the code generated by our unoptimized JIT as relatively competitive to a commercial implementation. In light of that, the graph shows that tracing does dramatically improve the performance of code generated by SPUR over SPUR without tracing, often by over a factor of 10. Thus our speedups really do come from tracing, and not just a better back-end or other effect.

Observe that where TraceMonkey beats V8, SPUR does so too and with similar magnitude. Vice-versa, where TraceMonkey is slower than V8, SPUR usually is as well. The graph thus substantiates our hypothesis that trace optimizations that work well for a dynamic source language tracer like TraceMonkey, also work well when tracing the code obtained by translating the dynamic language and its runtime to CIL.

Overall, V8 is still a formidable engine to beat. In 14/26 benchmarks V8 wins, often dramatically, whereas SPUR wins in 11/26 benchmarks.

	Trees	Loops	Traces	Traces /Tree	Loop Traces /Loop	Instrs. /Trace
3d-cube	96	8	156	1.6	5.1	863
3d-morph	9	3	10	1.1	1.3	380
3d-raytrace	62	8	102	1.6	4.0	1360
acc-bin-tree	38	0	47	1.2	-	650
acc-fannkuch	20	7	47	2.4	2.1	161
acc-nbody	15	4	21	1.4	1.8	2542
acc-nsieve	6	3	12	2.0	1.0	128
bitops-3bit	3	1	3	1.0	1.0	218
bitops-bits	6	1	15	2.5	8.0	55
bitops-and	3	1	3	1.0	1.0	376
bitops-nsiev	7	3	17	2.4	2.3	112
control-rec	45	0	56	1.2	-	260
crypto-aes	81	26	143	1.8	2.7	455
crypto-md5	16	4	17	1.1	1.3	4254
crypto-sha1	17	4	26	1.5	3.0	898
date-tofte	38	2	43	1.1	1.0	1345
date-xparb	41	3	63	1.5	3.0	1754
math-cordic	5	1	9	1.8	5.0	654
math-partial	8	1	9	1.1	2.0	1980
math-spectr	21	4	25	1.2	2.0	317
regexp-dna	3	2	3	1.0	1.0	2010
string-b64	7	3	8	1.1	1.3	2566
string-fasta	14	3	18	1.3	2.3	668
string-tagcl	19	2	30	1.6	3.0	460
string-unpac	42	6	78	1.9	6.3	1221
string-valid	16	3	34	2.1	3.0	416

Figure 10. Tracing statistics of SPUR on SunSpider benchmarks

The current overhead of SPUR's trace optimization and compilation is significant. The numbers reported in Figure 8 effectively do not include jit or tracing overhead for any JavaScript engine. We have not yet attempted to reduce the overhead. For example, in the crypto-md5 benchmark, the main loop results in a trace tree with only a single trace, consisting of 60891 instructions; total processing of this trace, including trace recording, optimization, and compilation is around 1.5 seconds.

Figure 10 shows some statistics about the trace trees SPUR builds for the SunSpider benchmarks. "Trees" shows the number of trace trees created from distinct trace anchors, "Loops" represents the number of trace trees which have looping paths, "Traces" represents the number of recorded traces, or in other words, the number of paths through all trace trees. "Traces/Tree" shows the average number of traces per tree, "Loop Traces/Loop" shows the average number of looping paths per tree with any looping path. "Instrs/Trace" shows the average number of instructions in recorded traces, where we count the number of instructions in our trace intermediate representation, before any optimization such as constant folding is applied. Note that "acc-bin-tree" and "control-rec" do not contain any loops. Compared to the statistics reported for TraceMonkey (Figure 13, "Detailed trace recording statistics" in [16]), SPUR tends to record more trees and loops. There are two main reason for this tendency: SPUR's JavaScript compiler specializes methods according to types for arguments that can be inferred statically at call sites. This causes the generation of different type-specialized method bodies, which in turn get separate trace trees. Also, SPUR does not only record traces for proper loops, but SPUR also starts tracing from trace anchors placed after all potential exit points from existing traces, and at entry points of potentially recursive methods. However, SPUR records fewer loops than TraceMonkey for some benchmarks. The reason for this phenomenon is likely to be found in the different approaches for nested loops, which SPUR does not support directly. The number of looping traces per loop also varies widely between SPUR and TraceMonkey, which is probably due to differences in the tracing systems' runtimes, whose internal branching over different object types are responsible for a significant number of traces.

9. Related Work

Several state-of-the-art JavaScript implementations employ just-in-time compilation, e.g. Apple's SquirrelFish Extreme and Google's V8.

Compiling multiple specialized code versions was pioneered by [8] in the context of the SELF language. Dynamic trace compilation was attempted systematically by Dynamo [2], which operates at the machine code level. It interprets machine code, records traces, optimizes them, and maintains a *trace cache* of optimized traces. Other binary optimization systems include [7, 13].

YETI [29] is a simple TJIT for Java, focussing on an efficient integration with an interpreter. HotPathVM [19] is another TJIT for Java. It attaches secondary (child) traces to primary traces, in order to form trace trees [18]. Traces are represented in SSA form. Child traces may connect different loops, effectively supporting loop nests. The goal of the HotPathVM was to provide a small footprint JVM with better performance than an interpreter. Compared to a state of the art JVM, HotPathVM still trails far behind in performance.

Tamarin-Tracing [10], and TraceMonkey [16] are two JavaScript TJITs based on the ideas in the HotPathVM. They use an interpreter to gather traces of high-level JavaScript operations and optimize these traces directly before compiling them to machine code. To save time recompiling trace trees, these TJITs use trace stitching, where optimized branch traces are stitched onto the main trunk (or side branch) without recompiling the entire tree. This approach saves compilation time, but does not permit aggressive optimizations like global register allocation (described in [15]).

Suganuma et. al. [27] propose an inter-procedural region selection scheme similar to our tracing approach. They use profiling and instrumentation to drive region selection and rely on on-stack replacement (OSR) to transfer out of compiled regions. Unlike traces, regions are arbitrary subsets of methods. They often contain join nodes, and thus require correspondingly more complex profiling and optimization algorithms.

Closely related to our translation of JavaScript into CIL is earlier work by Chang et.al. [9], where they embed JavaScript in Java, combined with a TJIT. Their system still contains a Java interpreter, whereas SPUR is based on jitting only. Our JavaScript translation can take advantage of value types in the native runtime.

More general trace-optimizations of hierarchically layered virtual machines (VMs) have been described recently [5, 26, 28]. However, while our approach is based on compilation at all levels (JavaScript to CIL, and CIL to machine code), these approaches are based on interpreters at all levels. As the interpreter loop of the guest VM would appear to be the hot loop and optimized, instead of the loops of the interpreted programs, they propose special hints to guide the trace optimization of the host VM, essentially to unroll the guest interpreter loop. While we do not have a general problem with an interpreter loop, we do employ similar hints to unfold loops in the JavaScript runtime that perform dynamic type lookups at runtime.

The *Dynamic Language Runtime* (DLR) [25] is a library on top of .NET which simplifies the implementation of dynamic languages. Our JavaScript runtime can be seen as a specialized implementation of the DLR. The DLR does not provide any support for tracing.

Cuni et.al. [11] describe how to use the built-in dynamic code generation features of the Microsoft CLR, namely the `Reflection.Emit` library, to realize a TJIT. As a consequence, their approach incurs a relatively high overhead for entering and leaving traces, as their optimized traces are regular methods. SPUR has the concept of trace methods, which inherit the stack frame of the originating method, and SPUR supports a `transfer` instruction that allows reconstructing inlined method calls.

Merrill et.al [24] present a system which compiles each method into two equivalent binary representations: a low fidelity region with counters to profile hot loops and a high-fidelity region that has instrumentation to sample every code block reached by a trace. When a hot loop has been identified, the low-fidelity code transfers control to the high-fidelity region for trace formation. Upon conclusion of a trace, execution jumps back to the appropriate low-fidelity region.

We have earlier reported on the static analysis based on abstract interpretation which SPUR performs to infer types of local variables and method arguments [22].

The SPUR system, written mainly in C#, having a static compiler (Bartok) and a JIT at runtime, is similar in nature to the Maxine system for Java [23], which was recently extended for tracing [4]. In contrast to the tracing Maxine system, SPUR's optimized traces seamlessly integrate with other precompiled or jitted code, without the need for adapter stack frames to mediate between different code versions, and SPUR supports any .NET language, including JavaScript.

10. Conclusion and Future Work

We presented a tracing just-in-time compiler (TJIT) for CIL, which significantly improves the runtime performance of JavaScript programs on the .NET platform. Our approach is novel in that it does not employ an interpreter at any language level. We confirmed our hypothesis that the performance benefits of tracing for dynamic languages is not lost when the tracing is performed on an intermediate level of the compiled dynamic language source, rather than at the source level. The key to make our approach work in practice is the ability to trace not only the target code produced by the dynamic language compiler, but also the runtime code of the dynamic language.

In future work, we want to reduce the overhead of trace optimization and compilation by revisiting our algorithms to avoid any non-linear computations (currently, our alias analysis, invariant code motion, and guard strengthening are potentially quadratic in the number of instructions). We also want to defer that workload into a parallel thread, so that the main program execution can continue undisturbed, similar to [3, 20]. As an alternative, the code generation process could be performed offline, leveraging previously recorded traces. We also want to garbage collect the jitted code when it is no longer needed.

SPUR can currently detect implied guards by checking built-in patterns over equalities, disequalities, and inequalities. We plan to use symbolic execution [21] along the traces combined with an SMT solver such as Z3 [12] to detect all implied guards, and to perform further optimizations. We also plan to investigate if precise symbolic knowledge about a loop body at runtime can be leveraged for automatic parallelization [6].

Some virtual machines, including Google's V8, use tagged pointers to compactly represent 31-bit integers without boxing. We want to investigate if such a value representation can further improve performance.

We plan to improve our static analysis on JavaScript code, to track liveness of local variables, and to perform an interprocedural analysis when possible.

References

[1] Andrew Clinick. Introducing JScript .NET, 2000. `http://msdn.microsoft.com/en-us/library/-ms974588.aspx`.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2000. ACM.

[3] M. Bebenita, M. Chang, A. Gal, and M. Franz. Stream-based dynamic compilation for object-oriented languages. In *TOOLS (47)*, pages 77–95, 2009.

[4] M. Bebenita, M. Chang, K. Manivannan, G. Wagner, M. Cintra, B. Mathiske, A. Gal, C. Wimmer, and M. Franz. Trace based compilation in interpreter-less execution environments. Technical Report ICS-TR-10-01, University of California, Irvine, March 2010.

[5] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *ICOOOLPS '09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25, New York, NY, USA, 2009. ACM.

[6] B. J. Bradel and T. S. Abdelrahman. Automatic trace-based parallelization of java programs. In *ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing*, page 26, Washington, DC, USA, 2007. IEEE Computer Society.

[7] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. pages 265–275, 2003.

[8] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *PLDI'89*. ACM Press.

[9] M. Chang, M. Bebenita, A. Yermolovich, A. Gal, and M. Franz. Efficient just-in-time execution of dynamically typed languages via code specialization using precise runtime type inference. Technical Report ICS-TR-07-10, University of California, Irvine, 2007.

[10] M. Chang, E. Smith, R. Reitmaier, M. Bebenita, A. Gal, C. Wimmer, B. Eich, and M. Franz. Tracing for web 3.0: trace compilation for the next generation web applications. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 71–80, New York, NY, USA, 2009. ACM.

[11] A. Cuni, D. Ancona, and A. Rigo. Faster than C#: Efficient implementation of dynamic languages on .NET. In *ICOOOLPS'09*. ACM Press.

[12] L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

[13] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, E. Klaiber, and J. Mattson. The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. pages 15–24. IEEE Computer Society, 2003.

[14] ECMA. International standard ECMA-355, Common Language Infrastructure, June 2006.

[15] A. Gal. *Efficient bytecode verification and compilation in a virtual machine*. PhD thesis, Long Beach, CA, USA, 2006. Adviser-Franz, Michael.

[16] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI'09*.

[17] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 465–478, New York, NY, USA, 2009. ACM.

[18] A. Gal and M. Franz. Incremental dynamic code generation with trace trees. Technical Report ICS-TR-06-16, University of California, Irvine, 2006.

[19] A. Gal, C. W. Probst, and M. Franz. HotpathVM: an effective jit compiler for resource-constrained devices. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 144–153, New York, NY, USA, 2006. ACM.

[20] J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *OOPSLA*, pages 155–174, 2009.

[21] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[22] F. Logozzo and H. Venter. RATA: Rapid atomic type analysis by abstract interpretation application to JavaScript optimization. In *Compiler Construction*, volume 6011 of *LNCS*, pages 66–83. Springer-Verlag, 2010.

[23] B. Mathiske. The maxine virtual machine and inspector. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 739–740, New York, NY, USA, 2008. ACM.

[24] D. Merrill and K. Hazelwood. Trace fragment selection within method-based jvms. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 41–50, New York, NY, USA, 2008. ACM.

[25] Microsoft. Dynamic Language Runtime, 2010. `http://www.codeplex.com/dlr/`.

[26] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *OOPSLA Companion 2006*. ACM Press.

[27] T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for dynamic compilers. *ACM Transactions on Programming Languages and Systems*, 28(1):134–174, 2006.

[28] A. Yermolovich, C. Wimmer, and M. Franz. Optimization of dynamic languages using hierarchical layering of virtual machines. In *DLS '09: Proceedings of the 5th symposium on Dynamic languages*, pages 79–88, New York, NY, USA, 2009. ACM.

[29] M. Zaleski, A. D. Brown, and K. Stoodley. Yeti: a gradually extensible trace interpreter. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 83–93, New York, NY, USA, 2007. ACM.