

Optimization of Dynamic Languages Using Hierarchical Layering of Virtual Machines

Alexander Yermolovich Christian Wimmer Michael Franz

Department of Computer Science
University of California, Irvine
{ayermolo, cwimmer, franz}@uci.edu

Abstract

Creating an interpreter is a simple and fast way to implement a dynamic programming language. With this ease also come major drawbacks. Interpreters are significantly slower than compiled machine code because they have a high dispatch overhead and cannot perform optimizations. To overcome these limitations, interpreters are commonly combined with just-in-time compilers to improve the overall performance. However, this means that a just-in-time compiler has to be implemented for each language.

We explore the approach of taking an interpreter of a dynamic language and running it on top of an optimizing trace-based virtual machine, i.e., we run a *guest VM* on top of a *host VM*. The host VM uses trace recording to observe the guest VM executing the application program. Each recorded trace represents a sequence of guest VM bytecodes corresponding to a given execution path through the application program. The host VM optimizes and compiles these traces to machine code, thus eliminating the need for a custom just-in-time compiler for the guest VM. The guest VM only needs to provide basic information about its interpreter loop to the host VM.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors—Compilers, Optimization

General Terms Design, Languages, Performance

Keywords ActionScript, Lua, dynamic languages, trace compilation, hierarchical virtual machines

1. Introduction

In recent years dynamic languages have grown in popularity and acceptance. Applications have become more interactive, and dynamic languages allow a flexible environment that is easy to deploy, modify, and prototype in. Although the number of dynamic languages has increased, the process of their implementation has not changed much.

At first, a simple interpreter is created when designing a new language. Once the language has gained a wider acceptance, further work goes into optimizing interpreter performance. However, at some point applications get too big to be interpreted efficiently

and users demand better performance. The final step in improving the performance of the language is to create a full-blown virtual machine (VM) with a just-in-time (JIT) or ahead-of-time compiler. However, only few dynamic languages reach this stage. It requires much development work and deep knowledge of both compiler optimization techniques and hardware architecture.

With a plethora of already available virtual machines it is questionable whether such a move is necessary for dynamic languages. A simpler approach is to build a VM for a dynamic language on top of an already existing VM, i.e., to build a *guest VM* on top of a *host VM*. The host VM is usually a mature VM that provides optimized services like garbage collection and optimized JIT compilation. Such a hierarchical layering of VMs has been proven to be efficient—take for example the two Python implementations Jython [20], which runs on top of Java, and IronPython [18], which runs on top of .NET.

However, this approach suffers from a fundamental problem: the host VM does not know that it is executing a guest VM, so it optimizes it like any other application. The interpreter of the guest VM is JIT compiled to machine code, instead of the actual workload *executed* by the interpreter. In order to get this workload compiled, current guest VMs use a two-stage approach: the JIT compiler of the guest VM translates the dynamic language to bytecodes of the host VM, and the JIT compiler of the host VM then translates these bytecodes to optimized machine code. This severely limits what features of the dynamic language a guest VM can support.

To overcome this problem, we propose a unified view of such a hierarchical VM setting where the host VM is aware of the guest VM. The two VMs communicate using a well-defined API, which allows the host VM to utilize its JIT compilation facility to directly compile and optimize the workload running on top of the guest VM. Figure 1(a) illustrates this approach. Traditional JIT compilation techniques are not well suited for this approach because we do not want to compile at the granularity of methods, especially we do not want to compile the main interpreter method of the dynamic language's interpreter as a whole.

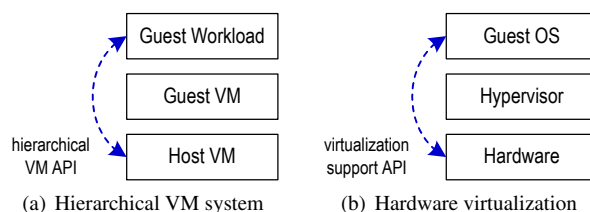


Figure 1. Structure of our hierarchical VM system and analogy with hardware virtualization.

© ACM, 2009. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in the Proceedings of the Dynamic Languages Symposium, pp. 79–88.

DLS'09, October 26, 2009, Orlando, Florida, USA.

<http://doi.acm.org/10.1145/1640134.1640147>

However, the novel *trace-based* JIT compilation techniques are a good fit, because frequently executed paths through a program are recorded and compiled. With only slight modifications of the trace recorder, it is possible to record the path through the guest workload instead of the path through the guest VM. All optimizations applied by the trace compiler of the host VM are then automatically applied to the traces of the guest workload.

Our approach of optimizing hierarchical VMs can be compared to recent advances in hardware virtualization. When a guest operating system is executed on top of a hypervisor, the guest operating system’s hardware-related structures become mere “user data” from the perspective of the underlying hardware. Recently adopted solutions to avoid the overhead of virtualization consist of a specific API (several new hardware instructions [28]) by which certain features of the guest OS can be exposed directly to the underlying hardware. This enables safe and efficient virtualization without requiring the hypervisor to continuously intervene (see Figure 1(b)), i.e., to eliminate the overhead of the hypervisor. Similarly, our optimization approach eliminates the overhead of the host VM.

We propose to execute a guest VM on top of a host VM with a trace-based JIT compiler. In our implementation, the guest VM is the Lua VM [17, 23], and the host VM is Tamarin-Tracing [8], a VM for Adobe’s ActionScript language. The Lua VM provides hints about its interpreter loop to Tamarin-Tracing. This allows the trace-based JIT compiler to capture and optimize the Lua workload. Our experience implementing a prototype of this systems allows us to make the following contributions:

- We present the approach of optimizing hierarchically layered VMs using trace compilation.
- We identify and explore a fundamental set of features that a guest VM needs to expose to the host VM.
- We show what optimizations a trace-based JIT compiler can perform on the workload executed by a guest VM.
- We present an implementation of the approach and evaluate its performance.

The remainder of this paper is organized as follows. Section 2 provides a general overview of trace based compilation that is at the heart of our approach. Section 3 describes our approach to optimize hierarchical VM layering and provides examples of problems as well as our proposed solutions. Section 4 describes the implementation of our system. Section 5 evaluates our approach based on a set of benchmarks. Future work is discussed in Section 6, and related work in Section 7. The paper ends with conclusions in Section 8.

2. Trace Compilation

Trace compilation uses a different paradigm than most other just-in-time (JIT) compilation techniques. Instead of adhering to principles derived from traditional static compilers, trace compilation is only applicable for dynamic compilers because it is based on runtime profiling information. Execution of a method starts in the interpreter. While the interpreter executes bytecodes, it keeps track of frequently executed “hot” loops of the program. In the trace-based approach it is assumed that backward branches represent a control flow transfer from the end of the loop to the beginning of the loop. When such branches becomes hot, i.e., when the execution count crosses a certain threshold, traces are recorded and compiled.

The example in Figure 2(a) shows the control flow graph of a loop that contains an *if* condition in the loop body. Assume that block 4 is executed more frequently than block 5. When the loop is detected as hot, trace recording starts at the loop header (block 2). This block is the *anchor* of the trace. The first trace, called *trunk*

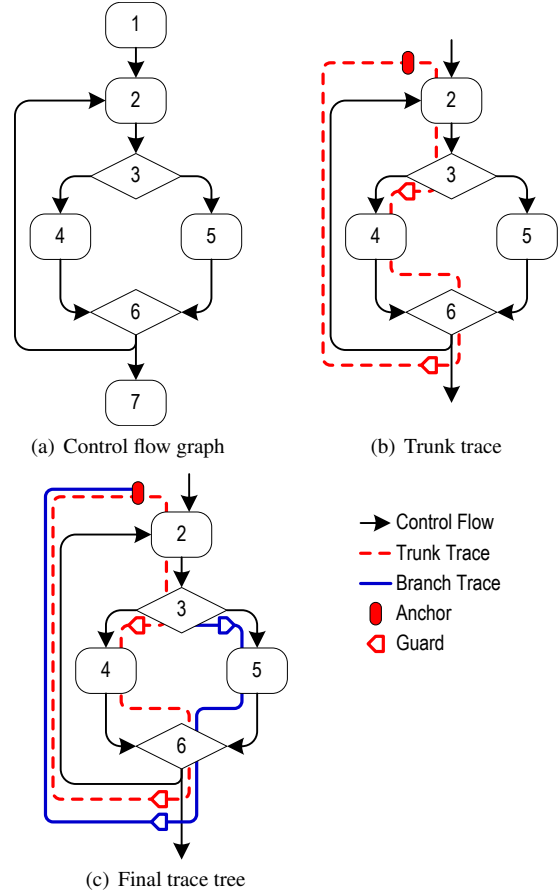


Figure 2. Example for trace compilation.

trace, covers the most frequently executed path through the loop, i.e., the blocks 2, 3, 4, and 6 (see Figure 2(b)). This trace is recorded while actually interpreting a loop iteration. In addition to executing a bytecode, the trace recorder also emits the necessary instructions of the intermediate representation (IR). Standard compiler optimizations are done on this IR and then machine code is generated.

At control flow split points, the trace follows only one control flow path. A special instructions, called *guard*, is inserted to check that the control flow during execution is consistent with the control flow during recording, i.e., that the condition that was recorded still holds. If not, the guard fails and a *side exit* code is executed that returns control back to the interpreter. In our example, the trace contains two guards: one at the end of block 3 and the second at the end of block 6. The first guard checks the branch condition, and the second guard checks the loop exit condition.

The transition from trace execution back to the interpreter is an expensive operation, and thus should not be a common case. If a loop contains multiple hot paths, i.e., if a guard fails frequently, it is necessary to record and compile this path. The initial trace is expanded to a *trace tree* by recording a *branch trace*. The next time the guard fails, the side exit does not return to the interpreter, but branches to the new trace. It is either possible to compile the whole tree as one unit, or every trace independently. Since the first approach leads to repeated compiles of the trunk trace, we use the latter approach. This is called *trace linking* in [8].

In our example, assume that block 5 is also executed frequently enough to be compiled, i.e., the guard of the trunk trace at the end of block 3 fails frequently. The recording of the second trace starts

at block 5, and leads back to the loop header through block 6. At the end of block 6, again a guard is inserted to check for the loop exit condition. Since this guard fails infrequently, no trace recording is started there. The final trace tree consists of two traces that cover the two paths through the loop. Figure 2(c) shows these traces. The blocks 1 and 7, which are not part of the loop, are never compiled.

3. Hierarchical VM Layering

This section presents our approach of optimizing the workload executed by the guest VM using the infrastructure provided by the host VM. We first present an example that is used throughout the section, and illustrate how it would be executed without our optimizations. Next we look at how the trace compiler of the host VM identifies the interpreter loop of the guest VM as frequently executed and compiles it, leading to a big trace tree with many short branch traces. Finally, we show the behavior of the system when the guest VM provides a *hint* about its interpreter loop to the host VM. With this hint, the trace compiler identifies loops of the actual workload as frequently executed and compiles them, leading to small trees without branches that only cover the important parts of the workload.

3.1 Example

Figure 3 shows the computation of the factorial written in a dynamic language. The variables are not typed, and the `for` loop uses an easy to understand syntax. We use the syntax and the bytecode format of Lua for our example, because our implementation is based on this system. However, the concepts are applicable to any interpreted dynamic language.

```
function factorial(n)
  local result = 1;
  for i = 2, n do
    result = result * i;
  end
  return result;
end
```

Figure 3. Lua source code of factorial example.

Before execution, the source code is parsed and converted to an internal bytecode representation. Figure 4 shows the bytecodes generated by Lua. To allow efficient interpretation, a compact binary representation is used where each bytecode has a fixed length of 4 bytes. The instruction set is register based and consists of untyped and high-level bytecodes. In the example, the register R0 stores the value of the parameter `n` passed to the method, and R1 stores the value of local variable `result`. The first four bytecodes load constants into registers and move values between registers.

The `for` loop is directly converted to high-level bytecodes. The registers R2, R3, and R4 store the loop counter variable `i`, the loop limit, and the loop increment, respectively. For technical reasons, these three registers must have consecutive numbers. The `FORPREP` bytecode performs all necessary pre-checks of the loop that must be executed only once, e.g., it checks if all three registers contain numbers, and then jumps to the `FORLOOP` bytecode. The `FORLOOP` bytecode modifies the loop variable and checks whether the loop end condition has been reached. If not, then the loop body is executed by jumping to the `MUL` bytecode in our example. The final `RETURN` bytecode returns the value stored in the register R1. This bytecode structure allows an efficient interpretation of the loop: only two bytecodes are executed frequently.

```
parameter n passed in R0
(1) LOADK 1 -> R1
(2) LOADK 2 -> R2
(3) MOVE R0 -> R3
(4) LOADK 1 -> R4
(5) FORPREP R2 -= R4; jump to (7)
(6) MUL R1, R2 -> R1
(7) FORLOOP R2 += R4; if R2 <= R3 then jump to (6)
(8) RETURN R1
```

Figure 4. Lua bytecodes of factorial example.

3.2 Interpretation

At the heart of the bytecode based interpreter is a dispatch loop. Figure 5 shows a simplified fragment that covers the bytecodes used in our example. It iterates over the bytecodes, analyzes one at a time, and uses the bytecode number in a `switch` statement to determine how to execute the bytecode. For bytecodes that do not change control flow, the program counter `pc` is just advanced by one to the next bytecode. Bytecodes that change control flow compute the new `pc` based on the bytecodes' semantics.

```
void interpret(Method m) {
  int pc = 0;
  while (true) {
    Bytecode bc = m.bytecodes[pc];
    pc++;
    switch (bc) {
      case LOADK: {...}
      case MOVE: {...}
      case ADD: {...}
      case MUL: {...}
      case FORPREP: {... pc = ...}
      case FORLOOP: {... pc = ...}
      case RETURN: {...}
      ...
    }
  }
}
```

Figure 5. Interpreter dispatch loop.

Because the bytecodes are untyped and at a high level, the implementation of the bytecodes requires type checks, value conversions, and error handling code for type mismatches. Therefore, the implementation of nearly all bytecodes contains several `if` statements and method calls, and some of them contain loops. This means that the execution of one bytecode requires a possibly complex control flow. When illustrating the interpreter in subsequent figures, we still collapse this control flow to one block to simplify the structure.

3.3 Trace Behavior Without Hinting

The interpreter introduced in the previous section is part of the guest VM. With the hierarchical layering of VMs, it is executed by the host VM. The trace-based JIT compiler of the host VM (as discussed in Section 2) records frequently executed paths and compiles them to machine code. Although this approach is effective for a regular workload, it is inefficient when the workload of the host VM is the guest VM. After the initial setup, the hot part of the guest VM is the interpreter dispatch loop. Therefore, this part of the guest VM is recorded and optimized, instead of capturing what the dispatch loop is executing.

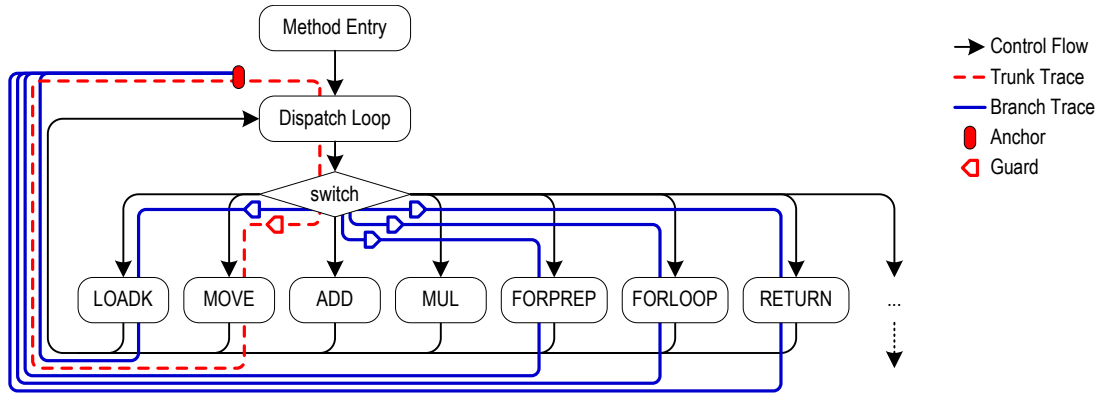


Figure 6. Traces recorded for factorial example.

In our factorial example, assume that the method is called to compute the factorial of 1. Therefore, the computation loop is not entered (the MUL bytecode is never executed), i.e., we have a sequential workload without any repeated execution. However, without any hint from the guest VM to the host VM, this code still causes the host VM to generate a trace tree. The host VM starts in the interpreter mode and executes the guest VM like any other application. When the host VM reaches the interpreter dispatch loop of the guest VM, it cannot distinguish between backward branches in the different `switch` cases because they all point to the header of the loop. This causes the threshold to be reached quickly, and all the backward branches, leading from `switch` cases to the loop header, to become hot. Assume that the threshold for compilation is two iterations, i.e., after two iterations the loop is considered hot so that the third iteration is recorded and compiled.

In the example, the loop is hot after the first two `LOADK` bytecodes are executed. On the third iteration of the loop, the trace recorder is invoked. It starts at the header of the loop, records through the instructions leading to the `switch` statement, and then records through the `MOVE` implementation. This initial trace covers only the code for the `MOVE` bytecode. At the condition point of the `switch` statement, a guard is inserted to check this. The dashed line in Figure 6 illustrates this trace.

During recording, the implementation of the `MOVE` bytecode is recorded. When the backward branch to the header of the loop is reached, the recording is stopped and the trace is compiled. All the traces are stored based on the destination of a backward branch. When the backward branch is taken the next time, a lookup is done to see if there is already a compiled trace in the cache. Because of this, the newly compiled trace is executed immediately after it is compiled. The next bytecode to be executed is the `LOADK` bytecode. Since this path has not been recorded before, it causes the guard at the `switch` condition to fail, and control returns to the interpreter of the host VM.

Like backward branches, all the side exits from traces are also tracked. Assume that the threshold for determining if a side exit is hot is set to 1, so recording of the `LOADK` bytecode begins right away. When the backward branch to the header of the dispatch loop is reached again, the recording is stopped and the trace is compiled. This process is repeated for each new bytecode that is encountered.

When the guest VM finishes execution of the workload, the trace-based JIT compiler has created traces for the implementation of the `MOVE`, `LOADK`, `FORPREP`, `FORLOOP`, and `RETURN` bytecodes. When the factorial of 1 is computed and thus the loop is never executed, none one of these traces are ever executed completely. The guard for the `switch` of the bytecode number always fails. Because of the overhead of recording and compiling traces, the

performance actually suffers with the trace-based JIT compiler compared to if the dispatch loop was purely interpreted.

Figure 6 illustrates all these traces. Note that the blocks for the bytecode implementations are abstract. Because each implementation requires control flow itself, the real traces contain more blocks and more guard instructions. Most of the control flow is for type checking and error handling, so these details are not important.

When the factorial of a large number is computed, the `MUL` bytecode is also traced. After this, no trace recording is necessary anymore. Starting with the second iteration of the workload loop, only compiled traces are executed and thus the performance is better compared to pure interpretation. Still, the overhead was overly high since several never executed traces were recorded and compiled. Additionally, the code for the `switch` statement that selects the correct bytecode trace is executed twice per loop iteration, i.e., for the `FORLOOP` and the `MUL` bytecode. Our optimization eliminates this overhead.

3.4 Hinting Mechanism

By providing directions, or *hints*, the guest VM allows the host VM to better understand the workload that is executed. A key insight into this is that a trace-based JIT compiler by default does not trace loops of the workload, but only loops of the guest VM. Because one loop iteration of the workload consists of multiple bytecodes, it still requires multiple loop iterations when the interpreter loop is compiled. To detect loops, the host VM looks at its program counter for executing the guest VM. The guest VM has a different program counter for executing the workload.

Passing the program counter of the guest VM (the *guest PC*) to the host VM and combining it with the original program counter of the host VM (the *host PC*) results in a new virtual program counter (the *virtual PC*) that accurately captures the guest workload. Both program counters are combined such that the guest PC goes into the higher order bits and the host PC uses the lower order bits of the virtual PC. Because of this, the dispatch loop of the interpreter is no longer recognized as a loop by the trace recorder.

In our sequential workload example where the factorial of 1 is computed, the guest PC is always increased at the loop header (remember that the `LOADK`, `LOADK`, `MOVE`, `LOADK`, `FORPREP`, `FORLOOP`, and `RETURN` bytecodes of Figure 4 are executed). The repeating sequences of the host PC values is dominated by the increasing values of the guest PC values. Therefore, no backward branch is detected in the workload and thus no bytecodes are traced and compiled.

When the factorial of a large number is computed, the `LOADK` to `FORPREP` bytecodes are still executed only once. The virtual PC only increases, so these bytecodes are not traced. The `MUL` and `FORLOOP` bytecodes then form the loop of the guest workload.

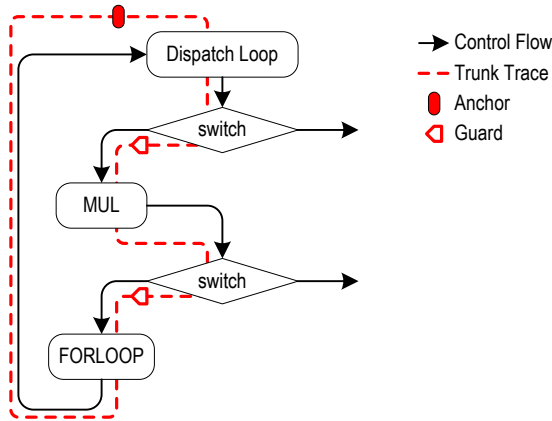


Figure 7. Trace of factorial function with hinting.

Because both the guest PC (the MUL bytecode) and the host PC (the top of the interpreter dispatch loop) are the same for each iteration, a backward branch is detected. The loop is recorded after a certain number of iterations. The recorded trace covers the MUL and FORLOOP bytecodes (see Figure 7). Only one trace is recorded. While the guards for the switch statement are still present, they do not fail until the factorial loop needs to be exited. Note that the boxes for the MUL and FORLOOP bytecodes represent multiple basic blocks since the implementation of these bytecodes requires some control flow for type checks and exception handling. However, it is also sufficient to trace one path through this control flow as the behavior is the same for each loop iteration.

A first natural approach would be to use only the guest PC in the host VM for trace creation. However, this would have unexpected consequences on the trace recorder because the granularity is too coarse, i.e., the PC then does not change throughout a whole iteration of the dispatch loop. While we do not want to record the dispatch loop, all other control flow necessary for the implementation of a bytecode must be correctly recorded. This control flow can contain loops itself, and we want such loops to be detected and recorded.

For example, the implementation of the VARARG bytecode contains such a loop (see Figure 8). It is necessary for the handling of functions that take a variable number of arguments. The VARARG bytecode reads them into registers so that values can be used inside a function. When using just the guest PC for trace recording, the trace recorder would not notice that a loop is being executed since the guest PC remains the same. Therefore, the recorder would follow all the backward branches and record each loop iteration, i.e., the loop would be completely unrolled. This would lead to excessive code duplication and is thus not feasible.

3.5 Guest VM Guarantees

The guest VM supplies information about its interpreter loop and its program counter to the host VM. Apart from this information, no other guarantees are necessary. The traces recorded by the host VM still contain all elements of the dispatch loop. As shown in Figure 7, the code for the switch of the interpreter dispatch loop precedes each bytecode implementation. Even if the information about the interpreter loop was incorrect, the resulting trace would be suboptimal but still valid. However, when the guest VM makes additional guarantees, the host VM can optimize the traces more aggressively.

The key insight for optimizations is that the bytecodes of the guest VM are immutable. Once the bytecodes of a method were

```

case VARARG: {
    ...
    for (j = 0; j < numArgs; j++) {
        storeValue(baseReg + j, callInfo.argument[j]);
    }
    break;
}

```

Figure 8. Implementation of the VARARG bytecode.

created by parsing the source code, they are not changed anymore. This means that memory loads from the bytecode array always yield the same results, i.e., the guard that checks that the first switch of our factorial example branches to the implementation of the MUL bytecode never fails. The host VM cannot eliminate this load without support from the guest VM because the host VM cannot distinguish between data structures of the guest VM that are immutable and data structures of the guest workload that are mutable.

Therefore, the guest VM needs to specify which regions of the memory contain the immutable bytecodes. When the trace recorder sees a load from such an immutable memory region, it replaces the load with a constant. The value of the constant is the value loaded during recording. Subsequent constant folding and dead code elimination based on conditions that can be evaluated statically can eliminate most parts of the interpreter dispatch loop.

Immutable bytecodes have the implication that self-modifying code is not allowed. Instead of modifying an existing method, it is necessary to deprecate the old method and put the modified code in a new method, which is then compiled and optimized anew. Note that the basic hinting mechanism described in the previous section allows self-modifying code. If code is modified after trace recording, the guard created for the switch of the interpreter dispatch loop fails. The modified code is first interpreted and then optimized.

Figure 9 shows the trace of the factorial example when the bytecode array is immutable. The switch blocks are eliminated, so only the code for the actual implementation of the MUL and FORLOOP bytecodes remain. The guard that checks the loop exit condition is part of the FORLOOP bytecode: this bytecode compares the loop counter variable and the loop limit and takes a side exit to the interpreter when the loop has reached the upper bound. There is no guard necessary checking that a FORLOOP bytecode comes after the MUL bytecode; the bytecode sequence was detected during trace recording, and the guest VM guarantees that it does not change in the future.

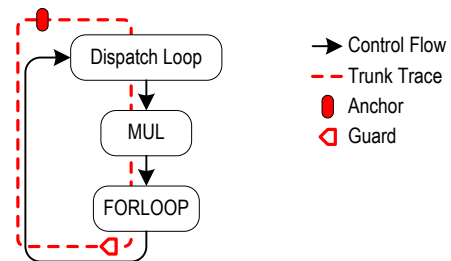


Figure 9. Trace without dispatch overhead.

3.6 Optimizations

Without our hinting mechanism, only individual bytecodes are captured in traces. The result is a large number of short traces. This limits the amount of global optimizations that the compiler of host

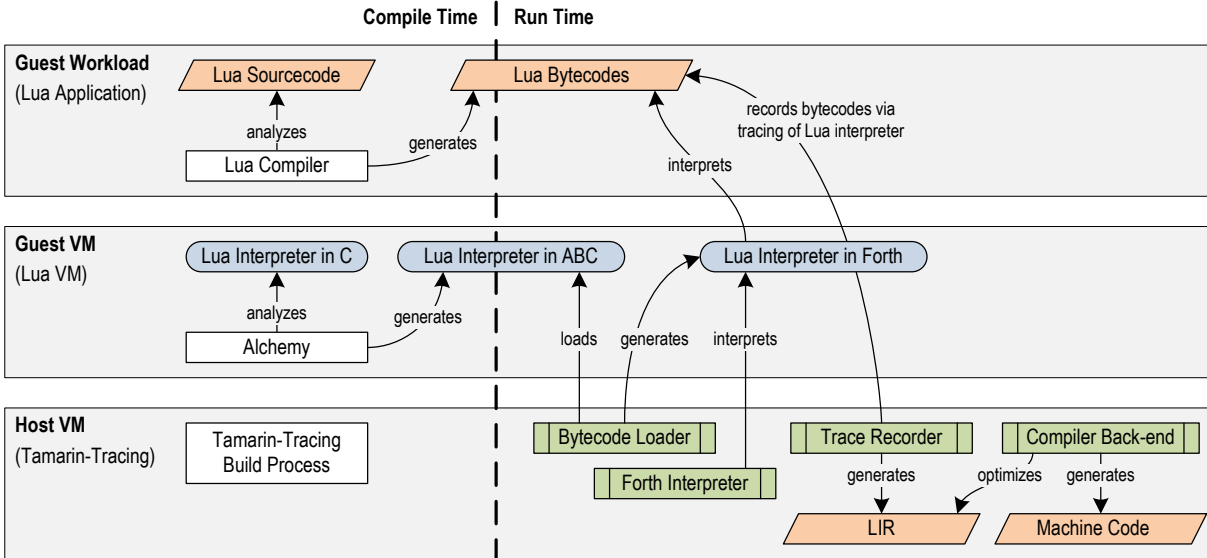


Figure 10. System structure of compile-time and run-time components.

VM can apply. Although it can still apply optimizations like constant folding, dead code elimination, common subexpression elimination, and type specialization on compiled traces, the scope is too limited for these optimizations to be effective.

For example, consider the computation loop of our factorial example (see Figure 4). Both the MUL bytecode and the FORLOOP bytecode access R2, i.e., the implementation of both bytecodes contains the same code that loads this value from the memory data structure that manages the local variables of a method. Without hinting, these two bytecodes end up in different traces and cannot be optimized together. When hinting is enabled, the two loads are in the same trace. Common subexpression elimination is able to discover that they load the same memory location and can eliminate the second one.

Similarly, the increased scope can be used by optimizations that eliminate unnecessary guard and type check instructions. Since Lua is a dynamic language, all type checks must be performed at run time. For example, an arithmetic operation must first check that all its input operands are numbers. As long as each bytecode implementation is traced and compiled separately, all of these checks are necessary. With hinting, type information from one bytecode can be used to eliminate type checks of subsequent bytecodes.

4. Implementation

Our implementation uses Tamarin-Tracing as the host VM. Tamarin is the code name of Adobe’s virtual machines that implement ActionScript 3 (AS3) [1], a flavor of ECMAScript [13]. 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. Tamarin does not directly operate on AS3 source code, but instead executes a bytecode intermediate format known as ActionScript bytecode (ABC) [2]. AS3 source files are converted to ABC using the Adobe Flex compiler [4].

Tamarin-Tracing [24] is a research virtual machine that is based on the virtual machine shipping with Adobe Flash. In contrast to the product version, it uses a trace-based compiler to achieve performance improvements. It is partially a self-hosting compiler, with many portions of the system itself written in ActionScript 3.

For the guest VM, we use the Lua VM [17, 23], which is written in C. Unfortunately, there is no appropriate VM for an established dynamic language available that is written directly in ActionScript 3. However, nearly arbitrary C and C++ code can be compiled to ActionScript using Adobe Alchemy [5], formerly known as FlaCC [3]. Alchemy is an extension of the LLVM Compiler Infrastructure project [21]. The ActionScript bytecode resulting from this somewhat involved build process is more complicated because it simulates raw memory and CPU instructions on a type-safe virtual machine. However, the system is fully functional.

The Lua VM can either directly operate on Lua source code, or it can execute code that was already compiled into the Lua bytecode format. We use the latter in our system configuration to eliminate the overhead of Lua source code parsing for the benchmark execution. Lua source code is converted to bytecodes using the Lua compiler that is part of the Lua codebase. The left side of Figure 10 illustrates the compile-time components of our hierarchical VM system.

4.1 Tamarin-Tracing Internals

During its startup, Tamarin-Tracing loads a set of libraries written in AS3 and compiled into ABC format. These libraries provide functionality, such as common math functions, to the AS3 programmer and provide a glue between the functions and their native implementation. Internally, Tamarin-Tracing uses the Forth language as an intermediate representation. This is a stack based language and has an instruction set of about 200 Primitive Words. Multiple Primitive Words can be combined to Super Words, which are interpreted as one unit in order to reduce the interpreter dispatch overhead.

ABC is translated to Forth at the time bytecodes are loaded, so Tamarin-Tracing is essentially a Forth virtual machine. In our system configuration, this means that the ABC bytecodes of the Lua interpreter are converted to Forth at startup time. Forth is executed internally by Tamarin-Tracing using an automatically generated interpreter. The interpreter profiles backward branches and triggers trace recording when a certain threshold is crossed. When a trace is recorded, it not only executes Primitive Words or Super Words, but also emits instructions of a low-level intermediate representation (LIR).

The LIR is a register-based intermediate representation that uses static single assignment (SSA) form [10]. Traces are linear sequences of instructions where branch traces can split off, but without control flow merges. Therefore, phi functions for the SSA form are only necessary at the loop header, i.e., at the start of the trunk trace. During recording, an unlimited number of virtual registers is used. After the trace is recorded, standard compiler optimizations are performed on the LIR using pipelined optimization filters. Final assignment to machine registers is done by a register allocator before machine code is emitted by the compiler back-end. The right side of Figure 10 shows these run-time components of our system.

4.2 Trace Recording and Execution

While interpreting Forth code, all backward branches are tracked and the number of times they are taken is counted. Branches are defined as *backward branches* when the PC is decremented by Forth branch instructions. When a backward branch is identified, the current execution state is saved and a helper function is invoked to handle it.

The helper function can either start the trace recording process, execute a trace, or do nothing if the backward branch is not hot. It performs a lookup in the trace cache to see if a trace already exists. The lookup is done using the PC that was passed into the function as part of the saved interpreter state.

If a compiled trace does not exist and the branch is hot, trace recording for a trunk trace is initiated and all the data structures necessary for recording of the trace are initialized. A flag is set so that the interpreter switches into trace recording mode and starts recording after the method returns. In this mode, LIR code is generated while Forth instructions are interpreted. Trace recording continues until an end condition is encountered. The main reasons to exit from recording are a backward branch to the beginning of the trace, a return bytecode without a corresponding call that was already traced, or an excessively long trace. After finishing recording, the trace is compiled, the data structures used during recording are discarded, and the trace is inserted into the trace cache.

If the trace cache lookup yields an already compiled trace, the current saved interpreter state is passed to it and the trace is executed. Upon the exit from a trace, the saved interpreter state is updated, and a lookup is done for the destination PC of the side exit. Tamarin-Tracing implements the trace linking optimization discussed in Section 2. When a side exit is frequently taken, a branch trace is created. The steps are the same as if the trace recording was done for a hot backward branch. The branch trace is compiled separately without affecting the trunk trace. On the next exit from this side exit, the exit code is patched with a jump to the branch trace so that execution remains in compiled code for all subsequent executions of this side exit.

4.3 Hinting Mechanism

The objective for our hinting mechanism is twofold. In addition to improving performance, it needs to be easy to use by the programmer who is instrumenting the guest VM, and when a hint is not provided it should not affect the host VM. We achieve these objectives by taking advantage of the already existing AS3 library that a programmer can use to access native functions. To this library, we add our own custom class `HintClass`. The class has `set` and `get` functions for passing the PC of the guest VM to and from the host VM. These two functions do not have an actual implementation in the AS3 class, but are marked with the `native` keyword. This tells the host VM that they invoke native functions implemented in it. We compile the class into a library that the host VM loads during startup, and provide the functionality of these two functions in the auto-generated `.h` and `.cpp HintClass` files. These functions set or

get a value in a variable that can be accessed internally from different parts of the host VM.

The use of our hinting mechanism varies depending on whether the guest VM is written directly in AS3, or in C/C++ and then compiled to AS3 using the Alchemy toolchain. If the guest VM is written in AS3, the programmer needs to import the `HintClass` using a standard import command, and invoke the `set` method in the appropriate places in the dispatch loop:

```
import avmplus.HintClass
...
HintClass.hintPC = pc
```

If the guest VM is written in C/C++, the programmer needs to instruct the Alchemy toolchain to insert the import command and the assignment command into the AS3 file that gets created:

```
__asm__("import avmplus.HintClass");
...
__asm__("HintClass.hintPC = %[pc]");
```

The `__asm__` keyword tells Alchemy toolchain to insert the specified AS3 code into the files it generates, replacing `%[pc]` with the actual program counter variable. The first command specifies the AS3 code to import our custom class. The second command specifies the AS3 code that invokes the setter method of our class and passes in the PC of the guest VM.

By implementing the hinting functionality as calls to native functions, we make sure that the `hintPC` gets updated not only when host VM is in the interpreter or recording a trace, but also when the compiled trace is executed. This has the advantage of not needing to pass the value in the saved execution state between mode transitions, and eliminates the overhead of needing to save another value when a trace exits from its execution.

With `hintPC` being updated, we can now use it during trace lookup. We modify the helper routines that do trace lookup to use a virtual program counter, consisting of the `hintPC` of the guest VM and the actual PC of the host VM. In our implementation, the virtual PC is a 64 bit value where the most significant 32 bits hold the `hintPC` value and 32 least significant bits hold the host PC. When no hint is specified, only the lower 32 bits hold a value, and the upper 32 bits are 0's. This allows for trace storage and lookup even when the hint is not provided. This is the case during the startup of the host VM, or when it executes a part of the guest VM outside the dispatch loop. Thus a hinting mechanism does not alter the host VM's behavior for a regular workload.

5. Evaluation

To evaluate the performance of our approach, we use several Lua benchmarks available from [22]. All benchmarks are executed on a system with a dual socket Intel Xeon X5140 2.33 GHz processor, with 32 GB of RAM, using Ubuntu Kernel version 2.6.28. We compare the performance of Lua running on top of Tamarin-Tracing without our hinting mechanism (referred to as *baseline*) and with our hinting (referred to as *optimized*). We do not provide the number for Lua running natively without Tamarin-Tracing. As pointed out in Section 4, the Lua interpreter, written in C, must first be converted to ActionScript. This adds a significant overhead because an unsafe language must be simulated in a safe environment. Therefore, the native version of Lua is orders of magnitude faster for some benchmarks. This paper does not deal with this overhead because it is not related to the hierarchical VM approach.

Figure 11 shows the execution time with our optimization (the second column, *O*) normalized to the baseline (the first column, *B*). The numbers above the second column show the speedup, i.e., the reciprocal of the reduced execution time. When the hinting

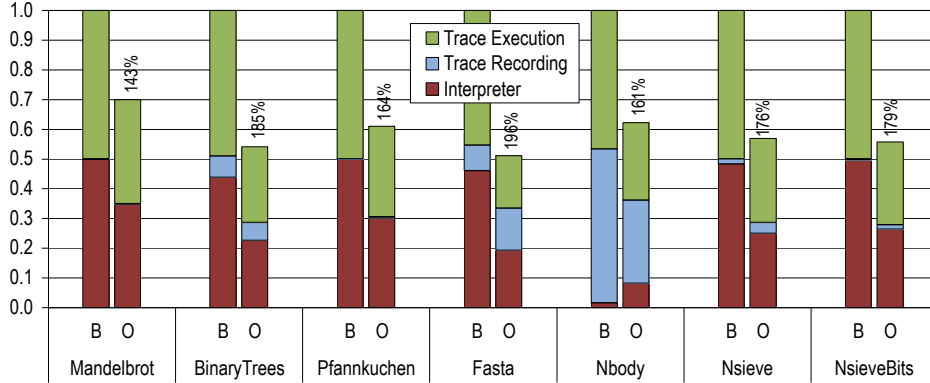


Figure 11. Breakdown of time spent in different parts of the host VM for baseline (*B*) and optimized (*O*) configuration (lower bars are better). The numbers show the overall speedup of the optimized version relative to the baseline.

	Trunk Traces		Branch Traces		Branches per Trunk	
	Baseline	Optimized	Baseline	Optimized	Baseline	Optimized
Mandelbrot	18	78	48	34	2.67	0.44
BinaryTrees	63	263	311	588	4.94	2.24
Pfannkuchen	37	282	133	257	3.59	0.91
Fasta	117	573	757	1290	6.47	2.25
Nbody	25	367	134	230	5.36	0.63
Nsieve	31	92	132	175	4.26	1.90
NsieveBits	35	139	150	200	4.29	1.44

Figure 12. Number of trunk traces and branch traces compiled.

mechanism is used, all benchmarks show a speedup, which varies between 143% and 196%. The most notable speedup is seen in *Fasta*, *BinaryTrees*, *NsieveBits*, and *Nsieve*. In these benchmarks, the host VM is able to identify hot regions of the guest VMs workload quicker and capture them in traces.

The execution time of each benchmark can be split into three parts: the time spent in the interpreter, in the trace recorder, and executing compiled code. Figure 11 also contains the breakdown of our benchmark results for these parts. For all benchmarks, our optimization reduces the time spent executing traces. Also, the overall time spent in the interpreter, either interpreting without or with trace recording, is reduced for all benchmarks.

For benchmarks like *Pfannkuchen*, the time necessary for trace recording is nearly negligible. The Lua code contains multiple small loops that are executed frequently and iterate a large number of times. For such benchmarks, the optimized host VM is able to quickly capture the hot parts of the guest VM into traces and subsequently execute them natively. With the hint enabled the host VM spends less time in the interpreter compared to when the hint is disabled.

The *BinaryTrees* benchmark uses recursive functions to construct binary trees of different depths, and to perform checks on these trees. To increase the workload, these functions are executed repeatedly in a loop. With the hint enabled, the host VM is able to capture the recursive calls in the guest VM workload, and then reuse these traces for trees of different depths. Without the hint more time is spent interpreting these recursive calls.

An exception to the general behavior is the *Nbody* benchmark. This benchmark has a few loops that execute long sequence of instructions. With the hint enabled, the host VM remains in the

interpreter longer, executing these loops until they get hot. Without the hint, the host VM spends less time in the interpreter, but this is offset by it recording different parts of the *Nbody* benchmark, thus requiring more time for trace recording than with the hint enabled.

In all cases, the time spent executing traces is less with the hint enabled than when it is disabled. With the hint enabled the host VM is able to capture the guest VMs workload into longer traces. This allows optimizations discussed in Section 3.6, which result in faster executing traces.

Traces that are compiled with the hint enabled on average also have less branches than traces that are compiled when the hint is disabled. Figure 12 shows the number of trunk traces and the number of branch traces recorded with and without optimizations. The number of trunk traces increases by a factor of 3 to 15 when enabling the hints. In contrast, the number of branch traces increases at most by a factor of 2 or even decreases. Therefore, the ratio of average branches per trunk traces (the last two columns of Figure 12) also decreases. These numbers match our expectations of the optimization: The overall number of traces increases because many workload loops are traced instead of the interpreter loop. However, the workload loops are much simpler and less “branchy” than the interpreter loop, so the average complexity of the traces decreases.

6. Future Work

One of the main advantages of dynamic languages is their type system flexibility where variables and object fields can change types at any time. Some languages, like JavaScript, go further by allowing fields to be dynamically added to objects. These are prototype-based languages. The flexibility dictates that a programmer implements these objects in the guest VM using dynamic collections and

other functionalities of the host VM. This approach adds additional complexity to the guest VM development and results in a considerable overhead both in space and time. In future work, we plan to extend the host VM to support the object model of the dynamic languages. The host VM provides a well defined set of APIs that allows the guest VM to dynamically create host VM classes. This not only simplifies a guest VM implementation, but also improves the performance by eliminating the overhead of field accesses.

7. Related Work

Bolz et al. [7] developed the idea of hierarchical VMs simultaneously to us. In their PyPy system [25], an interpreter for a new language is written in a reduced subset of Python. This *language interpreter* is executed by a *tracing interpreter*, which is responsible for recording a trace of the application executed by the language interpreter. The trace is then compiled to optimized machine code. PyPy compiles the language interpreter and the trace compilation system to C, and relies on a C compiler to create machine code. As a result, the guest VM and the host VM are compiled together by the C compiler, so the host VM is not visible as a separate entity. This mixing of C with a jit compiler complicates their system, e.g., they need a special fallback interpreter that handles guard failures. Our system does not need this because we use a self-contained and fully functional host VM (Tamarin-Tracing).

Trace recording is a well established technique for dynamic profile guided optimization of native binaries. Bala et al. [6] introduced it as a method for runtime optimization in their Dynamo system, which was then subsumed by DynamoRIO. In this system, machine code is optimized at run time, which differs from compilation of bytecodes because machine code misses all of the high-level information that bytecodes provide. Frequently executed basic blocks are compiled and added to the code cache. If the block is present in the cache, DynamoRIO retrieves and executes it instead of re-interpreting it. Blocks that are frequently executed sequentially are linked together, thus creating trees of basic blocks.

Sullivan et al. [27] optimize interpreters running on top of DynamoRIO. Their approach of optimizing the interpreter is similar to ours: DynamoRIO is aware of the interpreter loop and uses an abstract program counter that consists of the logical PC of the interpreter and the native PC of DynamoRIO. The major difference to our work is that DynamoRIO is not a VM, but a native code optimization system. Their approach is much more limited because they must not make any assumptions about the program, including register usage, calling conventions, or the stack layout. Our approach of integrating two full-fledged VMs allows far more aggressive optimizations.

Gal et al. [15] introduced trace compilation and trace trees for JIT compilation. They integrate trace recording and trace compilation into the JamVM, a lightweight Java VM for embedded devices. Subsequently, trace compilation was successfully used in JIT compilers for JavaScript [14, 16] and ActionScript [8]. All these implementations target only the optimization of one language, without the support for hierarchical VM settings. Our implementation is based on the trace compiler for ActionScript.

With the increased popularity of dynamic languages, several commercial projects try to make it easier for developers to integrate guest VMs with an existing host VM. One example is the Java Specification Request (JSR) 223 [19], a mandatory part of Java 6. It specifies an API to integrate code written in arbitrary other languages into Java source code. Using this API, more than 30 dynamic languages can be easily integrated into Java applications [26]. However, the API does not provide any integration of the guest VMs into the Java VM; they remain regular applications from the point of view of the Java VM.

The developers of guest VMs on top of Java frequently complain that Java bytecodes and the Java VM specification make it difficult to compile dynamic languages directly to Java bytecodes. To resolve this issue, Sun Microsystems has proposed the *Da Vinci Machine Project* [11] to simplify the execution of dynamic languages on a Java VM. The goal is to add small extensions to the Java VM that eliminate the most complicated workarounds in current implementations of dynamic languages. Examples of proposed extensions are a new bytecode for dynamic method invocation, method handles that allow references to methods in a manner simpler than reflection, new means for lightweight bytecode loading without the need to generate a complete class file, and support for continuations and tail calls in the Java VM. The integration of the guest VM and host VM remains loose, the host VM still only loads bytecodes generated by the guest VM.

The *Dynamic Language Runtime* (DLR) [12] developed by Microsoft aims at simplifying the implementation of dynamic languages on top of the Common Language Runtime (CLR). It is a library that provides commonly needed functionality, for example a compiler that translates an abstract syntax tree into bytecodes for the CLR, dynamic method calls, and a dynamic type system [9]. It is used by several projects, for example IronPython. In contrast to our proposed approach, it is only a library without any integration into the underlying host VM.

8. Conclusions

In this paper we explored an innovative approach of improving the performance of dynamic languages using a hierarchical VM framework. Instead of creating a unique JIT compiler in the guest VM for the dynamic language, we leverage the host VM with an already existing trace-based JIT compiler.

Normally, the host VM would only optimize the interpreter of the guest VM, but not the workload executed on top of the guest VM. To address this issue, the guest VM provides the host VM with hints about its interpreter loop, i.e., it specifies which parts of the guest VM should not be traced for its own but only in conjunction with the guest workload. With these hints the host VM can implicitly capture and optimize the workload executed by the guest VM by recording multiple iterations of the guest VMs interpreter dispatch loop.

The evaluation shows that this approach leads to a considerable speedup when executing the Lua VM on top of Tamarin-Tracing, i.e., when a well-established interpreter-based dynamic language is executed on top of a VM that already uses a trace-based JIT compiler.

Acknowledgments

We want to thank all of the Tamarin developers at Adobe, especially Scott Petersen, for the continuous help and support with the implementation of this project.

Parts of this effort have been sponsored by the National Science Foundation (NSF) under grants CNS-0615443 and CNS-0627747. Further support has come from generous unrestricted gifts from Sun Microsystems, Google, and Mozilla, for which the authors are immensely grateful.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and should not be interpreted as necessarily representing the official views, policies, or endorsements, either expressed or implied, of the NSF, any other agency of the U.S. Government, or any of the companies mentioned above.

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 FlaCC*, 2008. http://llvm.org/devmtg/2008-08/Petersen_FlashCCompiler.pdf.
- [4] Adobe Systems Inc. *Adobe Flex SDK*, 2008. <http://www.adobe.com/products/flex/flexdownloads/>.
- [5] Adobe Systems Inc. *Adobe Alchemy*, 2009. <http://labs.adobe.com/technologies/alchemy/>.
- [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] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: PyPy’s tracing JIT compiler. In *Proceedings of the Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25. ACM Press, 2009.
- [8] 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 *Proceedings of the International Conference on Virtual Execution Environments*, pages 71–80. ACM Press, 2009.
- [9] B. Chiles. CLR inside out: IronPython and the dynamic language runtime. *MSDN Magazine*, October 2007.
- [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] *The Da Vinci Machine Project*, 2008. <http://openjdk.java.net/projects/mlvm/>.
- [12] *Dynamic Language Runtime*, 2008. <http://www.codeplex.com/dlr/>.
- [13] ECMA. *Standard ECMA-262: ECMAScript Language Specification*, 3rd edition, 1999. <http://www.ecma-international.org/publications/standards/Ecma-262.htm>.
- [14] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. 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 *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 465–478. ACM Press, 2009.
- [15] 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.
- [16] J. Ha, M. R. Haghighat, S. Cong, and K. S. McKinley. A concurrent trace-based just-in-time compiler for JavaScript. Technical Report TR-09-06, University of Texas, Austin, 2009.
- [17] R. Ierusalimsky, L. H. de Figueiredo, and W. C. Filho. Lua - an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.
- [18] *IronPython*, 2009. <http://www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython>.
- [19] *Java Specification Request 223: Scripting for the Java™ Platform*, 2006. <http://www.jcp.org/en/jsr/detail?id=223>.
- [20] *The Jython Project*, 2009. <http://www.jython.org/Project/>.
- [21] *The LLVM Compiler Infrastructure*, 2009. <http://llvm.org/>.
- [22] *Lua Benchmarks*, 2009. <http://shootout.alioth.debian.org/gp4/luaphp>.
- [23] *Lua Programming Language*, 2009. <http://www.lua.org/>.
- [24] D. Mandelin. *Tamarin Tracing Internals, Part I to V*, 2008. <http://blog.mozilla.com/dmandelin/2008/05/>.
- [25] *PyPy*, 2009. <http://codespeak.net/pypy/>.
- [26] *Scripting Engines for Java*, 2008. <https://scripting.dev.java.net/>.
- [27] G. T. Sullivan, D. L. Burening, I. Baron, T. Garnett, and S. Amarasinghe. Dynamic native optimization of interpreters. In *Proceedings of the Workshop on Interpreters, Virtual Machines, and Emulators*, pages 50–57. ACM Press, 2003.
- [28] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.