Dynamic Interpretation for Dynamic Scripting Languages

Kevin Williams, Jason McCandless and David Gregg
{kwilliam, mccandjm, david.gregg}@cs.tcd.ie
Trinity College Dublin

Abstract

Dynamic scripting languages offer programmers increased flexibility by allowing properties of programs to be defined at run-time. Typically, program execution begins with an interpreter where type checks are implemented using conditional statements. Recent JIT compilers have begun removing run-time checks by specializing native code to program properties discovered at JIT time.

This paper presents a novel intermediate representation for scripting languages that explicitly encodes types of variables. The dynamic representation is a flow graph, where each node is a specialized virtual instruction and each edge directs program flow based on control and type changes in the program. The interpreter thus performs specialized execution of whole programs. We present techniques for the efficient interpretation of our representation showing speedups of greater than 2x over *static interpretation*, with an average speedup of approximately 1.3x.

1. Motivation

Scripting language virtual machines (VMs) typically compile high-level source code into an array of low-level opcodes (bytecodes). They then use a standard interpretation loop to execute the array of opcodes. Run-time type checks are implemented with either conditional statements or switch statements. Performing these type checks forms a significant portion of program execution. These costs seem unnecessary especially when we realize the same variable types are inevitably rechecked as loop iterations repeat the execution of the same block of code.

Recent research has shown the advantages of type specialization in JIT compilation [19, 10]. However, little research exists on the specialization of interpreters for scripting languages. We propose a scripting language representation and interpretation technique that performs specialized execution of dynamic code. We present a dynamic intermediate representation (DIR) which explicitly encodes the types of variables at each point of execution. The DIR is a flow graph, where each node is a specialized virtual instruction and each edge directs program flow based on control and type changes in the program. The proposed DIR therefore has a specialized path in the graph for every sequence of control and type changes found during execution.

In this paper we present the initial development of our prototype implementation in the Lua Virtual Machine [12, 13]. We present the design of our dynamic representation as well as the techniques used for its efficient interpretation. We illustrate our representation with a motivating example and present our plans for future interpreter optimizations. We compare the interpretation of our representation to the standard Lua implementation and show early speedups reaching a max of 2.13x and an average of 1.3x. We present reductions in the total number of instructions executed including reductions in conditional branches (used in interpreter type checking) as well as decreases in data and instruction cache accesses.

2. Background

All high level programming languages have type systems. A type system defines a set of valid values and a set of operations that can be applied to those values. The type system allows high level languages to detect and prevent invalid operations such as an arithmetic operation between two strings. The types of variables can be defined statically at compile time, or dynamically at run time. Languages such as C and Java have static type systems; whereas Perl, Python and Lua are dynamically typed. In a statically typed language, the type of a value that a variable can contain is determined at compile time. In dynamically typed languages however, variables can contain values of any type and each time an operation is applied to a variable during program execution the type of the variable must be checked. This typically adds a significant overhead to executing programs in dynamically typed languages. A goal of much research is to reduce or eliminate dynamic type checks in compilers or virtual machines for dynamically typed languages [3, 10, 19, 17].

The Lua VM supports nine variable types: NIL, BOOLEAN, LIGHTUSERDATA, NUMBER, STRING, TABLE, FUNCTION, THREAD, USERDATA. Variables are stored in a *(type,value)* pair called a *'tagged value'*. The type field of this structure is a byte and the value field is a union of values accessed by the VM depending on the value of the type tag.

The Lua VM is a register based machine. Instructions executed by the interpreter access *'virtual registers'* stored in the VM's call stack. There are thirty eight instructions in the Lua instruction set [16]. Instructions are 32 bits wide and contain an opcode and between one and three operands. Operands can either index a register or declare a constant value. Instructions are stored in arrays. The executing interpreter accesses this array through a program counter index. The index is updated by each instruction. Standard instructions simply increment the counter to the next index while conditional branch and loop instructions increment the counter by an offset value. The interpreter then *dispatches* the instruction indexed by the program counter. In interpretation, dispatching is the process of sending interpretation to the correct location for the implementation of the program's opcode. In this paper we will refer to this representation as the static intermediate representation (or SIR) and to the interpretation technique as static interpretation.

3. Dynamic Intermediate Representation

In this section we present the DIR we propose for the specialized interpretation of Lua programs. Our representation is a flow graph where nodes represent specialized instructions and edges represent either control-flow or type-flow. Type-flow provides a path for each variable type defined by the node's operation. These paths provide the opportunity for type specialization. As every type change results in a new path, all variable types are known at every point in execution. The interpreter therefore achieves complete specialization of whole programs.

Paths in the graph are built on demand, meaning only those control flow and type flow paths which occur during execution are ever created. Nodes on paths are built one-by-one. As complete knowledge of local types is available, creation of a new node and selection of its specialized instruction is directed by the types of the instruction's operands. The program's structure and behavior is defined by the SIR generated by the script compiler (the Lua compiler in our case). Hence construction of the dynamic flow graph is guided by the SIR. The rest of this section describes the structure of nodes forming our DIR followed by a discussion of some of the specializations performed on the Lua instruction set.

3.1 Standard Node

The most basic node in our representation contains three fields: (1) the opcode of the specialized instruction, (2) a pointer to the array of live types and (3) a pointer to the next node in the path. Instructions which use this basic node structure always result in the same control and type flow. Examples of such instructions include loading of constant values, arithmetic instructions and direct branches. Figure 1(a) sketches the structure of this node and Listing 1(e) shows a pseudo-implementation of its dispatch.

3.2 Conditional Node

Conditional nodes have all the fields of a standard node and an additional pointer to a target node. They are used to implement instructions that can result in two paths of control flow but never result in any type changes. Examples of such instructions include conditional instructions such as *equal to*, *greater than* and *less than*, as well as loop instructions which control the flow of loops. Figure 1(b) sketches the structure of this node and Listing 1(f) shows a pseudo-implementation of its dispatch. Here we can see a condition statement defining the direction of execution.

3.3 Type-Directed Node

Type-directed nodes dispatch the next node based on the type of the operation's result variable. This node has three fields, the first two fields are the same as the standard node fields. The third field is an array of target nodes. Each entry in the array is a pointer to the first node of a new path. During execution, the type of the result of the operation is used to index the target array. The node selected is dispatched and execution continues down that path.

These nodes are used at any point where there is a single change of variable type and no change in control flow (i.e. all paths exiting the node execute the same instructions, the difference between paths is the types they are specialized to). The Lua language has nine types and so the dispatch array for our implementation has nine entries. Figure 1(c) sketches the structure of this node and Listing 1(g) shows a pseudo-implementation of its dispatch. Here we can see that the instruction defines some variable in *dest_reg*. This register has two fields, a value and a type. The type is used to index the dispatch array.

3.4 Call Node

Recall that each node in the DIR is an instruction specialized to a set of local variable types. Therefore, call nodes represent a call site and a known set of parameter types. A key/value mapping structure maps the call node to a function and the set of parameter types. The key for this mapping is the address of the called function. The value returned is the entry node of the function in the dynamic representation. The effect of this mapping is a flow of types across function boundaries.

In dynamic languages, the function called at a given point may change from execution to execution as functions are treated as first-class values. In practice, our experiments show that the majority of call points have a single called function, with a few having more than one. As a result of this

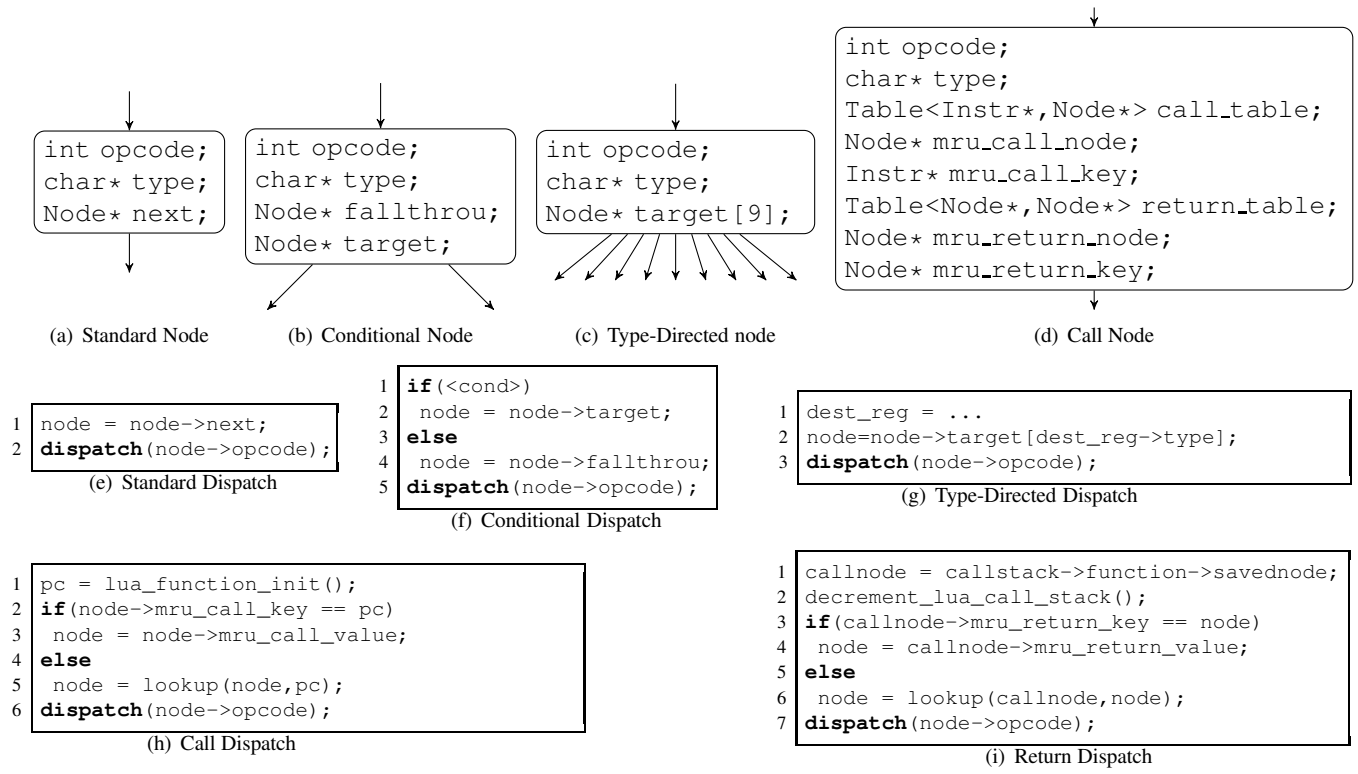


Figure 1. Node Structures and interpreter dispatch techniques.

profile information, call node mappings are implemented efficiently using two arrays. One array to store keys of functions and the second to store the value of nodes. A linear search of the key array will find the index of the correct node in the value array. Once found, this node can be dispatched in the usual way. Our implementation further improves the efficiency of this search by storing the most recently used key and value in 'cached' fields of the node. In practice these cached fields are usually a correct match, so linear searches only occur in a small number of cases.

3.5 Return Node

Having completed execution of a function, control flow must return to the calling function. As is the case for call nodes, return nodes represent a return instruction and a set of return types. Hence, an equivalent table mechanism using *(key,value)* pairs to find the next node is also used in return nodes. The return table is located in the calling node, as functions may regularly be called from different call sites but individual call sites rarely call multiple functions. Storing the return table in the call site therefore reduces the number of *(key,value)* pairs in a table and improves the search time for the correct key. Call nodes are stored in stack frames on the Lua stack and accessed from the stack at the end of each function.

The combination of our call/return mappings results in the inter-procedural type profiling of all function calls for any number of parameter and return variables (a powerful asset for JIT compilation).

3.6 Instruction Specialization

This section provides a brief overview of the set of instructions in the Lua VM and the range of specialization our interpreter performs.

1. Register Loads: Several different instructions in the Lua instruction set implement different types of loads. A specialization applicable to all of these instructions is the assignment of type. A register in Lua stores a value and a type. Commonly a load instruction loads a variable of the same type from a source register to a destination register. In these cases we have removed the redundant assignment of type to the destination register.
2. Arithmetic Operations: The usual set of arithmetic instructions are available in the Lua instruction set. Static implementations of these operations first check that both operands are of type NUMBER before proceeding with the operation. Having full type knowledge of variables, our specialized implementations of arithmetic instructions do not require type checking and simply perform the desired operation. The same is true for string operations like concatenation.
3. Table Access: Tables are the sole data structuring mechanism in the Lua language. They are used to implement a

wide range of data structures. They are associative structures and can be indexed by any value and can store values of any type. They contain two separate parts (1) a hash part, for storing values indexed by hashed values and (2) an array part for storing values indexed by integer keys. Our implementation provides specialized table access based on the type of the key. The wide use of tables in Lua makes this specialization an important optimization for overall performance.

4. Conditional Branches: The Lua instruction set has several different conditional instructions. These instructions compare two values and direct control flow based on the result of these comparisons. Valid comparisons can only be made between two values of the same type. Each conditional instruction in the instruction set is therefore specialized to the type of its operands. The value of this specialization is large because type checking in these instructions is performed by an expensive ANSI C switch statement.

4. Motivating Example

4.1 SIR

Figure 2 shows an example of a Lua implementation of the well known Sieve of Eratosthenes algorithm. The program calculates all prime numbers in the first *N* natural numbers. The implementation presented in Figure 2(a) marks all numbers which are not prime in the *flags* table with the boolean value *true*. At the end of this program all prime numbers will remain unmarked in the *flags* table.

Figure 2(b) is the SIR generated by the Lua compiler. Execution of this opcode results in the control flow pattern depicted in Figure 2(c). This graph is presented to illustrate the structure of a static flow graph to the reader and is never constructed at any point in static interpretation.

4.2 DIR

Figure 3 is an illustration of the graph that is constructed and interpreted by our dynamic interpreter for the Sieve of Eratosthenes program. The graph has twenty-six nodes in total. Each node has three labels, the first is a letter (a–z) which the authors will use for referencing nodes during discussion. The second is a number (1–15) which identifies the equivalent static instruction that the node specializes (found in Figure 2(b)). The last label is the opcode name. Opcodes in our DIR are specialized to the types of variables they are operating on. For example, node *v* is an add operation operating on two numbers, therefore the opcode selected is *add_number_number*.

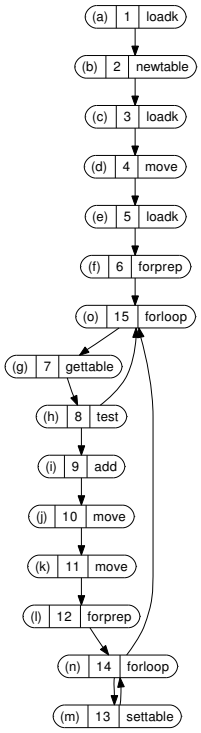
Edges in the graph represent program flow. Edges with no labels represent fall-through flow, i.e. instructions which always dispatch to the same next instruction in the program. Control flow edges are labeled *loop branch* and *cond branch*. These edges represent control flow dispatches for branch instructions. The remaining edges are type flow edges. In

```
1  local N = 100
2  local flags = {}
3  for i = 2, N do
4    if not flags[i] then
5      for k = i+i, N, i do
6        flags[k] = true
7      end
8    end
9  end
```

(a) Lua source code, a high-level representation.

```
1  loadk    r0 k0    ; reg0 = constant0
2  newtable r1 0 0   ; reg1 = new table(0,0)
3  loadk    r2 k1    ; reg2 = constant1
4  move     r3 r0    ; reg3 = reg0
5  loadk    r4 k2    ; reg4 = constant2
6  forprep  r2 L15   ; perform forloop prep, goto[15]
7  gettable r6 r1 r5; reg6 = reg1[reg5]
8  test     r6 L15   ; if reg6, goto[9] else goto[15]
9  add      r6 r5 r5; reg6 = reg5 + reg5
10 move     r7 r0    ; reg7 = reg0
11 move     r8 r5    ; reg8 = reg5
12 forprep  r6 r1    ; perform forloop prep, goto[14]
13 settable r1 r9 k3; reg9[reg1] = constant3
14 forloop  r6 L13   ; if loop, goto[13] else goto[15]
15 forloop  r2 L7    ; if loop, goto[7]  else [end]
```

(b) Lua opcode, a low-level representation.



(c) An *'imaginary'* control flow graph for the static interpretation of the Sieve of Eratosthenes algorithm.

Figure 2. A Lua implementation of the Sieve of Eratosthenes algorithm.

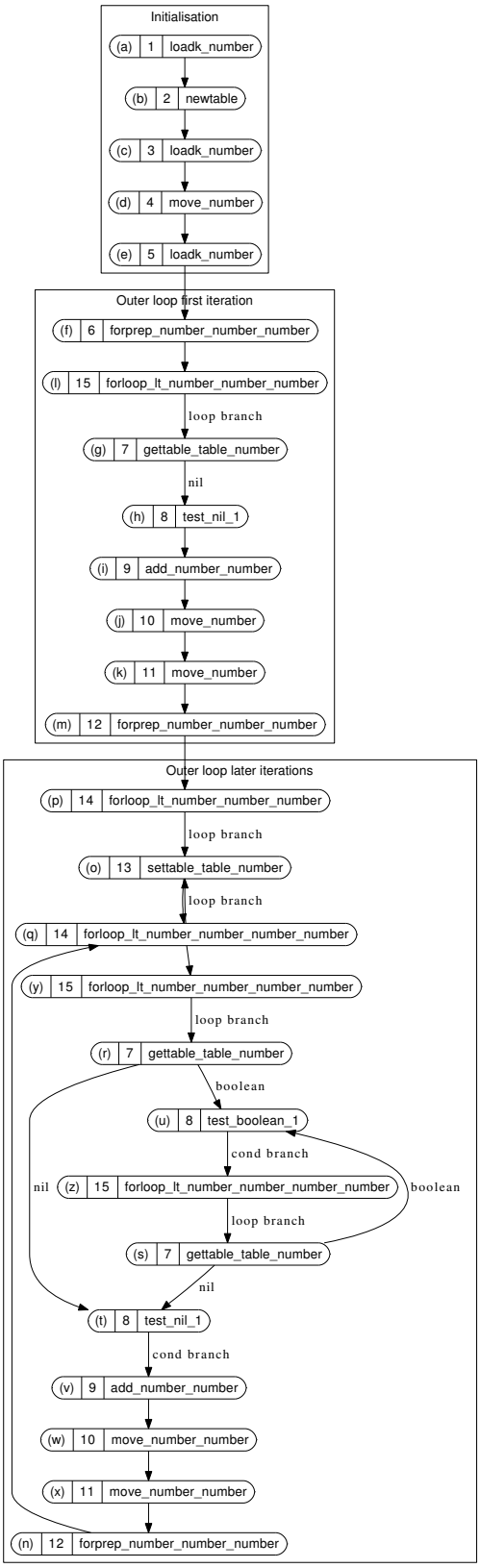


Figure 3. The Sieve of Eratosthenes dynamic representation built and executed by our interpreter.

this graph, they are labeled either *nil* or *boolean*. An example of type flow can be found in nodes *g* and *r*. These nodes are table fetch instructions and return a value from a table. They therefore dispatch based on the type of the return value. In node *g* there is only a single path, for type *nil*, as only one type value is ever returned at this point. In contrast, node *r* has two paths where values of type *nil* and *boolean* are returned. The two paths exit to the same program instruction, however the next and all subsequent instructions are specialized to the returned type.

The Sieve of Eratosthenes program is implemented with fifteen static Lua instructions. When these instructions are specialized to the local variable set found during execution, twenty-six specialized instructions are generated. In Figure 3 we have clustered these nodes into three sets. The first cluster represents the instructions that initialize the variables of the program. The second cluster is the first iteration of the for-loop spanning the static instructions 5–15. The final cluster is all subsequent iterations of the same loop. Two separate paths are generated for iterations of the outer loop as there are uninitialized variables in the first iteration. Only after the first iteration completes does the graph become stable.

5. Experimental Evaluation

5.1 Experimental Setup

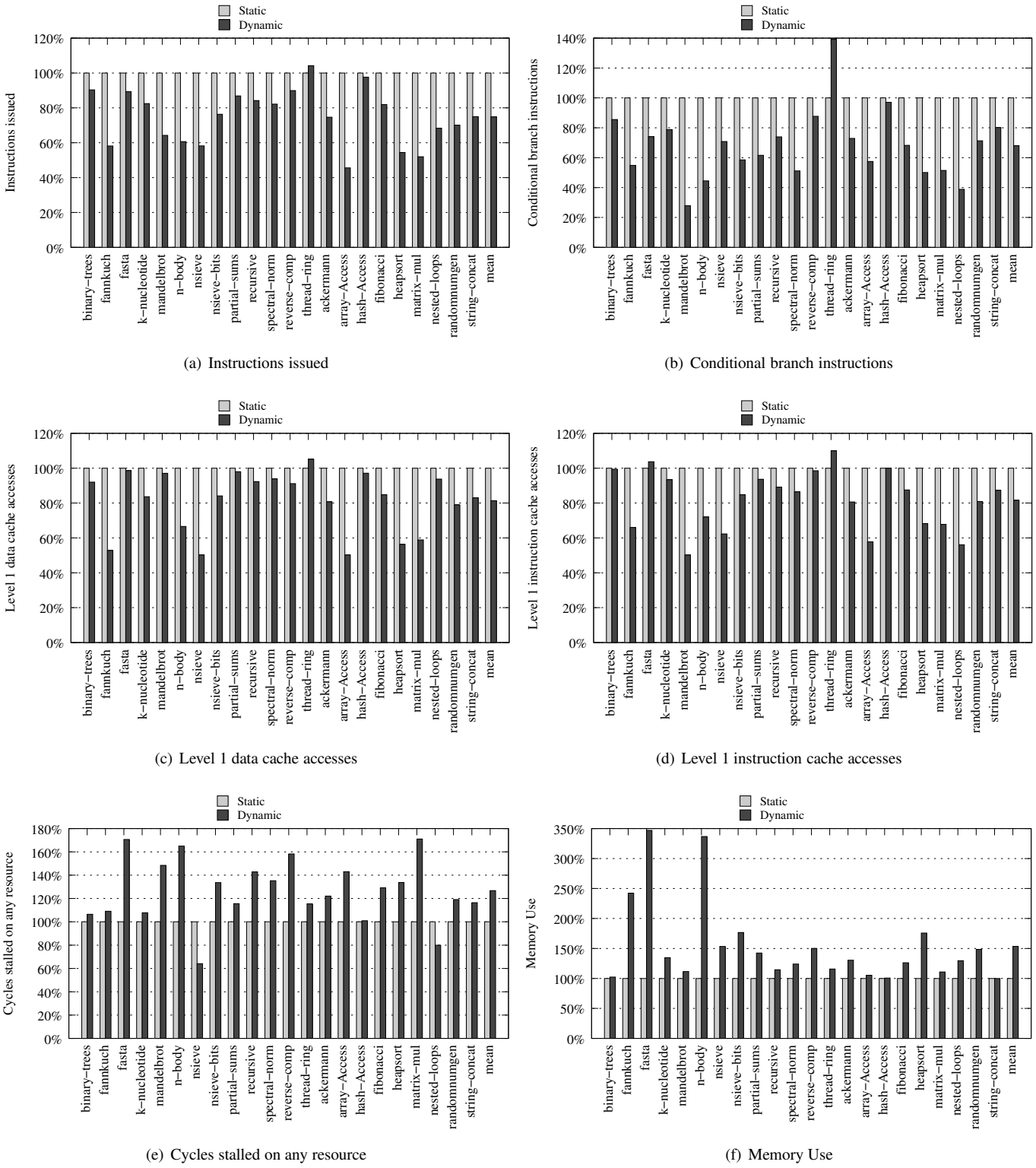
The following sections provide a comparison of hardware performance and running times between static and dynamic interpretation. The test machine used to run these experiments has two Intel Xeon Dual Core 2.13Ghz processors each with 4MB caches and 12GB of memory. The operating system is Ubuntu 9 with x86_64 GNU/Linux kernel 2.6.29.2. Lua source code is version 5.1.4. We have used the GNU gcc compiler version 4.3.3. Both the static and dynamic versions of code were compiled with the optimization options '-O3 -fomit-frame-pointer'. The hardware performance counters presented were collected using the PAPI profiling tool.

Our micro-benchmark set is taken from the Computer Language Benchmarks Game [7] and the Great Win32 Computer Language Shootout [11]. A lack of formal benchmark sets for scripting languages make these suites a common source of benchmarking for scripting language implementations [10, 3, 6, 17].

Figures 4(a) to 4(e) present hardware performance counters. Results are presented relative to the absolute result for the static interpreter. Hence, in all these figures, static bars are at 100% and the dynamic versions show percentage increases or decreases relative to the absolute static numbers.

5.2 Machine Instructions Issued

We first look at the total number of machine instructions issued to execute both static and dynamic interpreters (see Figure 4(a)). With the exception of one benchmark, all show a decrease in the number of machine instructions issued. This shows that the combination of dynamic interpreter dispatch



(a) Instructions issued

(b) Conditional branch instructions

(c) Level 1 data cache accesses

(d) Level 1 instruction cache accesses

(e) Cycles stalled on any resource

(f) Memory Use

Figure 4. (a–e) present hardware performance counters collected using the PAPI profiling tool. All results are presented relative to the absolute result for static interpretation. (f) presents memory use in the heap and stack, collected by polling the process status information from the Linux proc file-system.

and instruction specialization results in fewer instructions than the standard (and cheaper) static interpreter dispatch and (more expensive) run-time conditional type-check.

The benchmark that does not achieve a decrease in instructions issued is the *thread-ring* benchmark. This program benchmarks the Lua *coroutine* library, executing many coroutine yield and resumes. At the time of experimentation, our prototype interpreter had no efficient implementation for resuming from coroutines after yields and hence each resume requires a lookup of the correct resume node (removing this lookup is a future goal of the project).

5.3 Branch Instructions

The effect of interpreter specialization can be best seen in the dramatic decrease of conditional branch machine instructions (again across all benchmarks except *thread-ring*). By far the best performing benchmark in this experiment is the *mandelbrot* benchmark. This benchmark executes many arithmetic instructions. And the greater than 75% reduction in conditional branches is achieved largely from specializing those arithmetic instructions. Other notable results are for *array-access* and *matrix-mul* both of which have many number-key table accesses. All of these accesses are specialized to the key type.

5.4 Cache Access

Figures 4(c) and 4(d) present Level 1 cache accesses for both the data cache and instruction cache. Interpreter dispatch of nodes requires more loads compared to an equivalent bytecode dispatch. Because of this, an overall increase in data cache accesses would be expected. Surprisingly, dynamic interpretation reduces the overall number of data cache accesses. The reduction in data cache accesses is a result of removing type accesses for run-time type checking. The instruction cache has an overall reduction in cache accesses as would be expected after a reduction in instructions issued.

5.5 Processor Stalls

Despite the reduction in cache accesses, Figure 4(e) shows that the DIR implementation leads to many more processor stalls across almost all benchmarks. There are several factors contributing to these stalls. An increase in data cache misses is recorded, caused by the inefficient allocation of nodes. Memory for individual nodes is currently allocated on a node-by-node basis. The result of this allocation is many nodes scattered inefficiently around memory. The allocation also contributes to an increase in TLB misses. Implementing specialized versions of interpreter instructions leads to an increase in instruction size which leads to a corresponding increase in instruction cache misses. A more compact allocation of nodes is expected to improve data cache performance in future iterations of our interpreter — where DIRs will be stored in contiguous blocks of memory, managed by the VM.

5.6 Memory

DIRs require a separate node for each specialization of an individual instruction. Programs which contain many changes in control and type flow will result in large graphs and an increase in memory use. Figure 4(f) shows this behavior in a couple of our benchmarks — *fasta* and *n-body*. They both contain large increases in memory use as the data sets they are operating on are small and hence the DIR is relatively large.

5.7 Performance

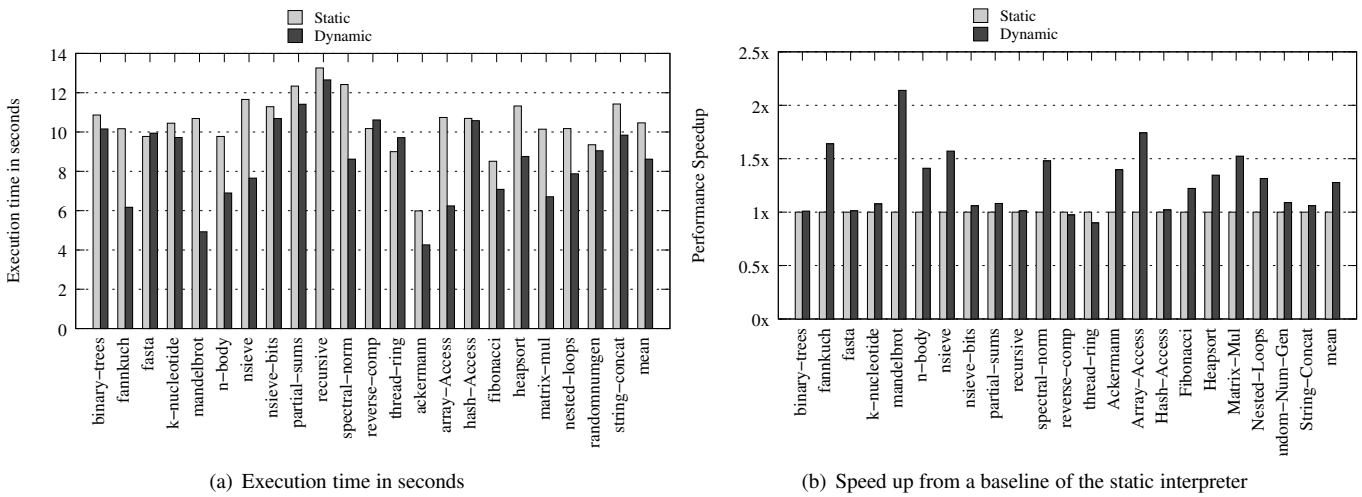
Figures 5(a) and 5(b) show running times and speedups for our benchmark set. Benchmarks whose bottleneck is the execution of instructions that perform lots of type checking achieve a very favorable speedup. Other benchmarks whose bottleneck is system library calls and non-specializable Lua instructions see a smaller increase in performance. The interpreter dispatch overhead of our DIR approach is greater than that of bytecode dispatch. Despite this increased cost, the only benchmark which achieves a significant slow down is *thread-ring* and only because our current implementation lacks an efficient coroutine resume mechanism.

6. Scope for Optimization

This paper has presented a dynamic intermediate program representation that encodes both dynamic control flow and variable type flow through a whole program. It has shown interpreter dispatch techniques which enable the efficient execution of the representation. It finally presented analysis of the technique; comparing its performance to that of a static equivalent. In order to present a fair comparison of both techniques this paper has so far neglected the possibilities of optimization of the dynamic representation. In this section we present some thoughts on our future work and the types of optimization we plan to improve performance in the future.

Interpreter Dispatch: The Lua 5.1 implementation is designed to be as portable as possible. For these reasons, the Lua authors have used a switch based interpreter dispatch. Our current representation copies this dispatch technique. More efficient dispatch techniques have been established in previous work. Threading techniques [2] such as token threading and direct threading are equally applicable to our representation. As specialization reduces the bottleneck of type checks, instruction dispatch becomes more and more important to overall performance.

Register Caching: A popular optimization among stack based virtual machines is to cache the top value(s) of the stack in machine registers [8]. An equivalent approach to caching virtual registers in machine registers is to date un-



(a) Execution time in seconds

(b) Speed up from a baseline of the static interpreter

Figure 5. A comparison of run-time performance.

established[1]. Using our dynamic representation we plan to specialize instructions with implicit register operands. Due to the number of virtual registers it is not advisable to cache all registers. The authors plan to create a mapping of machine registers to virtual registers. Region boundaries will be given header and footer nodes for loading and storing cached registers. Caching will be implemented by need, meaning only the most frequently used registers will be cached.

Super Nodes: Another common interpreter optimization is to concatenate common pairs of instructions together to form a single instruction [18]. This technique is equally applicable to dynamic representations. The authors plan to use established techniques in the field to implement super node optimizations [9, 5].

Loop Optimizations: Loops often provide opportunities for optimization in compilers — as is the case in our interpreter. We plan to leverage the type knowledge of variables inside loops to eliminate array bounds checking and move memory allocation outside the loop body. Table access nodes can be further specialized to 'cache' the array pointer; we believe this will lead to a massive performance gain. The number type in Lua is implemented using 'doubles'. We plan to optimize loop counters by specializing them to integers when we know the loop bound is constant.

Dead Node Removal: Specialization of operations can lead to redundant nodes. For example in Figure 3, node *t* specializes the TEST instruction to an operand of type NIL. The result of this operation is always false and control flow will always follow the FALSE path. This is illustrated in Figure 3 as there is only a single path leaving the TEST node, which is a conditional node. The TRUE path is never built as it is never executed. Nodes specialized to this level can be removed and all entry nodes can be directed to the target exit node.

Library Nodes: Scripting languages often provide standard libraries to perform complex operations like STRING, MATH and IO functions. The authors plan to inline these operations into the interpreter. This will remove the stack incrementing and decrementing required for their execution and will improve performance of the functions themselves as they are heavily inlined into the interpreter.

JIT Compilation: Mixed-mode VMs begin executing programs in a profiling interpreter. When a *'hot'* region of code is identified a JIT compiler compiles the opcodes to a native machine code representation of the same program. Individual JIT compilers often favor a single level of granularity as a compilation unit (e.g. functions, loops or traces). Optimization and specialization of a region of code can be based on: (1) analysis performed at JIT time [17] or (2) recording a single run of execution and compiling based on behavior found during that execution [10]. Future executions of the same region of code may or may not follow the same type and control flow as the compiled region and accordingly will remain in native code or request the compilation of further paths. This approach to mixed-mode execution can cause common program flow structures such as nested loops to make multiple calls to the JIT compiler. While modern JIT compilers have established workarounds to this problem, the behavior remains undesirable.

We suggest that the use of our DIR will improve the current model of static interpretation followed by speculated specialization. When a hot region of code is discovered, a JIT compiler could compile to native code guided by the

[1] It is in-fact a goal of the JavaScript Squirrel-fish VM, http://webkit.org/blog/189/announcing-squirrelfish/

verbose profile of the DIR. Optimizations of control-flow paths and specialization of operations could be performed with greater precision, while granularity levels could be determined based on the needs and behavior of the hot region. For example, a loop containing an if-then-else statement as a body may be compiled at either a loop or trace granularity. A trace would be chosen when either of the conditional paths were executed more often than the other. However, in the case where both paths are equally likely, the JIT compiler could choose to compile the whole loop and avoid further compilation and eliminate expensive entering and exiting of native regions. In summary, we claim our proposed representation has two advantages over current models: (1) specialization can be achieved at all levels of execution from interpretation to native execution of JIT code and (2) more effective JIT shapes and levels of specialization could be achieved as compilation is directed by a full program profile created by the dynamic interpreter.

7. General Applicability

The DIR presented in this paper is implemented in the Lua VM. However, the technique is applicable to other scripting languages — some require minor changes to their variable structures and some would specialize instructions in different ways to Lua. For example, in Python, arithmetic instructions can overflow from one type (*int*) into another (*long*). Arithmetic behavior like this would be modeled with a type-directed node (see Figure 1(c)). In general, dynamic scripting language implementations have some *'tagged value'* structure to define *(type,value)* pairs. In Lua, these tagged values have a byte which encodes one of the nine possible types. Our dynamic interpreter uses this byte to index the array of targets for node dispatch. In Python, tagged values use a pointer to a type object; however objects of 'built-in' types can be augmented with extra information. The data structure for objects of these types would need to be extended with a type indicator byte. In PHP, tagged values called *ZVals* have a type byte which encodes one of eight basic types. Javascript implementations differ in their bytecode format. In the SpiderMonkey implementation, the tagged value, termed *jsval*, is a machine word with three bits representing the type, thus some decoding would be required.

8. Related Work

Scripting language specialization has been the topic of only a few recent research publications. All of these research efforts have been directed at either ahead-of-time compilation and analysis or just-in-time compilation and specialization of run-time variables.

Gal et al. wrote a tracing JIT compiler for the JavaScript VM running in the Firefox web browser [10]. They successfully showed that compiling at a trace granularity allowed for the specialization of JIT compiled code. They used *type guards* at trace entry and *side-exits* at selected points in the trace to guarantee type-safe execution. They used SSA-based *trace trees* to perform aggressive optimizations and presented trace formations to handle the problem of nested control flow. Their interpreter, which begins the execution of all programs and performs the profiling of control flow (but not type) uses *'static interpretation'* techniques. Zaleski *et al.* [23] presented a tracing JIT for the Java language. They used direct calling dispatch techniques in their interpreter to gradually develop a trace JIT of a hot region of code. Rigo's psyco [19] is a run time specialization technique for the Python programming language. It performs run time specialization *by need* using a mixed execution/specialization phase of execution to specialize JIT code fragments.

Biggar et al. presented *phc*, an ahead-of-time compiler for the PHP language. They performed static analysis of PHP programs and developed a compiler technique that links with the language's interpreter to enable correctness through future iterations of the language [3]. Jensen et al. present a static analysis infrastructure for the JavaScript language. Their technique uses type inference and pointer analysis to achieve precision rates greater than 90% in a large number of benchmarks. Their technique, while powerful, has some shortcomings. In their worst case they achieve 61% precision. These results show that the use of static analysis alone is not sufficient to infer large numbers of types in all programs. The dynamic representation presented in this paper guarantees 100% precision for all run-time local variables.

Bruening and Duesterwald [4] investigated strategies for finding optimal JIT compilation unit shapes for Java programs. They explored strategies for minimizing compiled code size and maximizing time spent in JIT code. They concluded that using multiple levels of granularity in JIT compilation could lead to greater performance. Work by Whaley [22], and later by Suganuma et al. [20, 21] used dynamic profiling inside the Java Virtual Machine's interpreter to reduce compilation time by selecting and compiling smaller sections of code they called *partial methods* (i.e. loops). Larus presented whole program paths which capture a program's complete control flow [15]. An outcome of this work was effective discovery of *hot subpaths* for programmers and compilers to optimize. In Ammons' and Larus' retrospective piece [1] they observe the success of program path optimizations in the JIT commuinity. The DIR in this paper similarly builds program paths to improve interpreter efficiency.

Kistler and Franz [14] pioneered the concept of using tree structures in the Java VM. The advantage of their tree represnetation over the original bytecode was a more compact representation that contained more high-level information that improved JIT compilation.

9. Conclusions

This paper has presented a novel approach to scripting language specialization. The approach is the first stage of a

project which plans to bring program specialization to all levels of execution from interpretation to post-JIT native execution. Our experiments have shown our interpretation technique to be more efficient than existing techniques for scripting languages. In future work we plan to bring further efficiency to both interpretation and JIT compilation.

While our approach has shown performance to encourage adoption of our technique in the wider community, we have not addressed some of the potential scaling issues with our representation. For the most part scripting languages are in general type-stable, but the possibility still exists for a dynamic representation to grow to an unsuitably large size. A study of potential profiles is required to analyze the risk of programs which are pathological in type and/or control flow. Possible solutions to any such programs could be to (1) limit the number of paths allowed by creating unspecialized paths in problem regions, (2) remove paths which may have become obsolete or (3) require the existence of a static interpreter as a fall back processing unit.

We have claimed that our dynamic representation will bring improvements to the area of JIT specialization. Our future work includes plans to build an experimental JIT compiler which will make run-time decisions about compilation units. Leaving these compilation decisions to run-time should improve the quality of JIT compilation as whole program profiles are available to guide appropriate compilation units.

References

[1] G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. *SIGPLAN Not.*, 39(4):568–582, 2004.

[2] J. R. Bell. Threaded Code. *Commun. ACM*, 16(6):370–372, 1973.

[3] P. Biggar, E. de Vries, and D. Gregg. A practical solution for scripting language compilers. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied computing*, pages 1916–1923, New York, NY, USA, 2009. ACM.

[4] D. Bruening and E. Duesterwald. Exploring optimal compilation unit shapes for an embedded just-in-time compiler. In *In Proceedings of the 2000 ACM Workshop on Feedback-Directed and Dynamic Optimization FDDO-3*, pages 13–20, 2000.

[5] K. Casey, D. Gregg, and M. A. Ertl. Tiger – an interpreter generation tool. *Compiler Construction*, pages 246–249, 2005.

[6] 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.

[7] CLBG. The Computer Language Benchmarks Game. *Available at http://shootout.alioth.debian.org/*, 2008.

[8] M. A. Ertl. Stack Caching for Interpreters. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 315–327, New York, NY, USA, 1995. ACM.

[9] M. A. Ertl, D. Gregg, A. Krall, and B. Paysan. Vmgen: a generator of efficient virtual machine interpreters. *Softw. Pract. Exper.*, 32(3):265–294, 2002.

[10] 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.

[11] GWCLS. The Great Win32 Computer Language Shootout. *Available at http://dada.perl.it/shootout/*, 2008.

[12] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The Implementation of Lua 5.0. *Journal of Universal Computer Science*, 11(7):1159–1176, July 2005.

[13] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The evolution of lua. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 2–1–2–26, New York, NY, USA, 2007. ACM.

[14] T. Kistler and M. Franz. A tree-based alternative to java byte-codes. *International Journal of Parallel Programming*, 27(1):21–33, 1999.

[15] J. R. Larus. Whole program paths. *SIGPLAN Not.*, 34(5):259–269, 1999.

[16] K.-H. Man. A No-Frills Introduction to Lua 5.1 VM Instructions. In *http://chunkspy.luaforge.net/*. Lua Chunkspy Project, 2003.

[17] M. Pall. The LuaJIT Project. *Available at http://luajit.org*, 2008.

[18] T. A. Proebsting. Optimizing an ansi c interpreter with superoperators. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 322–332, New York, NY, USA, 1995. ACM.

[19] A. Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 15–26. ACM Press, 2004.

[20] T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for a java just-in-time compiler. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 312–323, New York, NY, USA, 2003. ACM.

[21] T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for dynamic compilers. *ACM Trans. Program. Lang. Syst.*, 28(1):134–174, 2006.

[22] J. Whaley. Partial method compilation using dynamic profile information. In *OOPSLA '01*, pages 166–179, New York, NY, USA, 2001. ACM.

[23] M. Zaleski, A. D. Brown, and K. Stoodley. Yeti: a gradually extensible trace interpreter. In *VEE '07*, pages 83–93, New York, NY, USA, 2007. ACM.