Dynamic Interpretation for Dynamic Scripting Languages

Kevin Williams, Jason McCandless, David Gregg

Trinity College Dublin

April 28, 2010
Introduction

- Novel intermediate representation for scripting languages that explicitly encodes types of variables.
Outline

1. Motivation
2. Background
3. Motivating Example
4. Dynamic Intermediate Representation
5. Experimental Evaluation
6. Scope for Optimization
7. General Applicability
8. Conclusions
Motivation

- Scripting languages growing really fast!
- Run slower than statically typed languages (type checks)
- Lua: a popular portable scripting language amongst many developers
Motivation

- Standard interpretation loop often used to execute array of opcodes
- Run-time type checks a significant portion of program execution
- Research shows advantages of type specialization in JIT, little research on specialization of interpreters
Our Idea

- A Dynamic Intermediate Representation that explicitly encodes types of variables at each execution point
- Dynamic IR is a flow graph, each edge directs program flow based on control and type changes in the program
  - Therefore specialized path in the graph for every sequence of control and type changes
This Paper

- Present the initial development of our prototype implementation in the Lua VM
- Design of dynamic representation as well as techniques used for its efficient interpretation
Background

- High level languages have type systems. They enable detection and prevention of invalid operations...
- Dynamically typed languages – each time an operation is applied to a variable during program execution, the type of the variable must be checked
- Goal of much research to reduce or eliminate dynamic type checks in compilers or virtual machines
• Lua VM has nine variable types: \{nil, boolean, lightuserdata, number, string, table, function, thread, userdata\}
• Register based machine
• Thirty-eight instructions
• Interpreter accesses array of instructions through program counter
Static IR

```plaintext
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
```
Static IR

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
15. forloop r2 L7 ; if loop, goto[7] else [end]
‘Imaginary’ Static IR graph

An ‘imaginary’ control flow graph for the static interpretation of the Sieve of Eratosthenes algorithm.
Initialization

(a) 1 loadk_number
(b) 2 newtable
(c) 3 loadk_number
(d) 4 move_number
(e) 5 loadk_number

Outer loop first iteration

(f) 6 forprep_number_number_number
(g) 7 gettable_table_number
(h) 8 test_nil_1
(i) 9 add_number_number
(j) 10 move_number
(k) 11 move_number
(l) 12 forprep_number_number_number

Outer loop later iterations

(m) 14 forloop_lt_number_number_number
(n) 15 forloop_lt_number_number_number
(o) 14 forloop_lt_number_number_number
(p) 14 forloop_lt_number_number_number
(q) 13 settable_table_number
(r) 7 gettable_table_number
(s) 8 test_boolean_1
(t) 8 test_nil_1
(u) 8 test_boolean_1
(v) 9 add_number_number
(w) 10 move_number_number
(x) 11 move_number_number
(y) 15 forloop_lt_number_number_number
(z) 15 forloop_lt_number_number_number

loop branch
loop branch
loop branch
loop branch
cond branch
cond branch
cond branch
cond branch

cond branch
cond branch
cond branch
cond branch
cond branch
cond branch
cond branch
cond branch
cond branch
cond branch
cond branch
cond branch
cond branch
cond branch
cond branch
cond branch
cond branch
cond branch
cond branch
cond branch
cond branch
cond branch
Dynamic IR

- Dynamic IR a flow graph
  - Nodes represent specialized instructions
  - Edges represent either control-flow or type-flow
- Every type change results in a new path -> all variable types are known at every point in execution
- Construction of the dynamic flow graph guided by the Static IR
Standard Node

- Always result in same control and type flow
Conditional Node

- Instructions that result in two paths of control flow but no type changes
Type-Directed Node

- Dispatch the next node based on the type of the operation’s result variable
Call Node

- Represent a call site and known set of parameter types

```c
int opcode;
char* type;
Table<Instr*, Node*> call_table;
Node* mru_call_node;
Instr* mru_call_key;
Table<Node*, Node*> return_table;
Node* mru_return_node;
Node* mru_return_key;
```
Return Node

- Represent a return instruction and a set of return types
Instruction Specialization

1. Register Loads
2. Arithmetic Operations
3. Table Access
4. Conditional Branches
Benchmarks

- No real standard for scripting language benchmarks
We use a set of kernel benchmarks taken from CLBG and GWCLS.
Experimental Setup

- Intel Xeon dual core 2.13GHz
Performance
Performance

![Performance Speedup Graph]

- Static
- Dynamic

Performance Speedup

- 0x
- 0.5x
- 1x
- 1.5x
- 2x
- 2.5x
Experimental Evaluation

Machine Instructions Issued

x86 Machine Instructions Issued

<table>
<thead>
<tr>
<th>Function</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary-trees</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fannkuch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fasta</td>
<td></td>
<td></td>
</tr>
<tr>
<td>k-nucleotide</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mandelbrot</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n-body</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nsieve</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nsieve-bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>partial-sums</td>
<td></td>
<td></td>
</tr>
<tr>
<td>recursive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spectral-norm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reverse-comp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread-ring</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ackermann</td>
<td></td>
<td></td>
</tr>
<tr>
<td>array-Access</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hash-Access</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fibonacci</td>
<td></td>
<td></td>
</tr>
<tr>
<td>heapsort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>matrix-mul</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nested-loops</td>
<td></td>
<td></td>
</tr>
<tr>
<td>randomnungen</td>
<td></td>
<td></td>
</tr>
<tr>
<td>string-concat</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conditional Branch Instructions

Static
Dynamic
0%
20%
40%
60%
80%
100%
120%

binary−trees
fannkuch
fasta
k−nucleotide
mandelbrot
n−body
nsieve
nsieve−bits
partial−sums
recursive
spectral−norm
reverse−comp
thread−ring
ackermann
array−Access
hash−Access
fibonacci
heapsort
matrix−mul
nested−loops
random−mung
string−concat
mean
L1 Data Cache

![Bar chart showing L1 data cache accesses for various benchmarks. The x-axis represents different benchmarks such as binary-trees, fannkuch, fasta, k-nucleotide, mandelbrot, n-body, nsieve, recursive, spectral-norm, reverse-comp, thread-ring, ackermann, array-Access, hash-Access, fibonacci, heapsort, matrix-mul, nested-loops, random-numgen, string-concat, mean. The y-axis represents the level of data cache accesses ranging from 0% to 120%. There are two bars for each benchmark, one for static and one for dynamic, indicating the percentage of cache accesses. The chart highlights the comparison between static and dynamic interpretations.]
L1 Instruction Cache

- Static
- Dynamic

Level 1 instruction cache accesses
Cycles Stalled On Any Resource

Static
Dynamic

0%
20%
40%
60%
80%
100%
120%
140%
160%
180%

Cycles stalled on any resource

- binary−trees
- fannkuch
- fasta
- k−nucleotide
- mandelbrot
- n−body
- nlsieve
- nsieve−bits
- partial−sums
- recursive
- spectral−norm
- reverse−comp
- thread−ring
- ackermann
- array−Access
- hash−Access
- fibonaccι
- heapsort
- matrix−mul
- nested−loops
- randomnumgen
- string−concat
- mean
Memory Use

- Static
- Dynamic

Graph showing memory use for various programs.
Interpreter Dispatch

- Switch based currently used in Lua. Other techniques equally applicable to our representation.
- As specialization reduces the bottleneck of type checks, instruction dispatch becomes more important to overall performance.
Register Caching

- Popular among stack based virtual machines
Super Nodes

- Concatenate common pairs of instructions to form a single instruction
Loop Optimizations

- Leverage the type knowledge of variables inside loops to eliminate array bounds checking and move memory allocation outside loop body
- Specialize loop counters to integers when loop bound is constant
Dead Node Removal

- Specialization can lead to redundant nodes.
- Can be removed and all entry nodes directed to target exit node
Dynamic IR graph

(a) 1 loadk_number
(b) 2 newtable
(c) 3 loadk_number
(d) 4 move_number
(e) 5 loadk_number

(f) 6 forprep_number_number_number

(g) 7 gettable_table_number

(h) 8 test_nil_1

(i) 9 add_number_number

(j) 10 move_number

(k) 11 move_number

(m) 12 forprep_number_number_number

(n) 12 forprep_number_number_number

(p) 14 forloop_it_number_number_number_number

(q) 14 forloop_it_number_number_number_number

(r) 7 gettable_table_number

(s) 8 test_boolean_1

(t) 8 test_nil_1

(u) 8 test_boolean_1

(v) 9 add_number_number

(w) 10 move_number_number

(x) 11 move_number_number

(y) 7 gettable_table_number

(z) 15 forloop_it_number_number_number_number

Outer loop later iterations

Outer loop first iteration

Initialization
Library Nodes

- Stack incrementing and decrementing required for executing standard library operations
JIT Compilation

- Mixed-mode
- Dynamic IR will improve current model of static interpretation followed by speculated specialization
General Applicability

- DIR implemented in Lua
- Applicable to other scripting languages
Conclusions

- Presented a novel approach to scripting language specialization
- First stage of a project to bring specialization to all levels of interpretation
- Not addressed some potential scaling issues with representation
Thanks

Questions?