Dynamic Interpretation for Dynamic Scripting Languages

Kevin Williams, Jason McCandless, David Gregg

Trinity College Dublin

April 28, 2010

embarkinitiative
Investing in People and Ideas

Introduction

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

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

Background

- 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

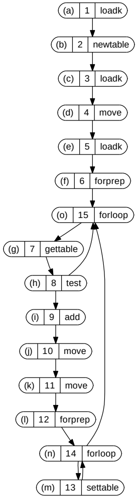
Static IR

```
local N = 100
local flags = {}
for i = 2, N do
  if not flags[i] then
    for k = i+i, N, i do
      flags[k] = true
    end
  end
end
```

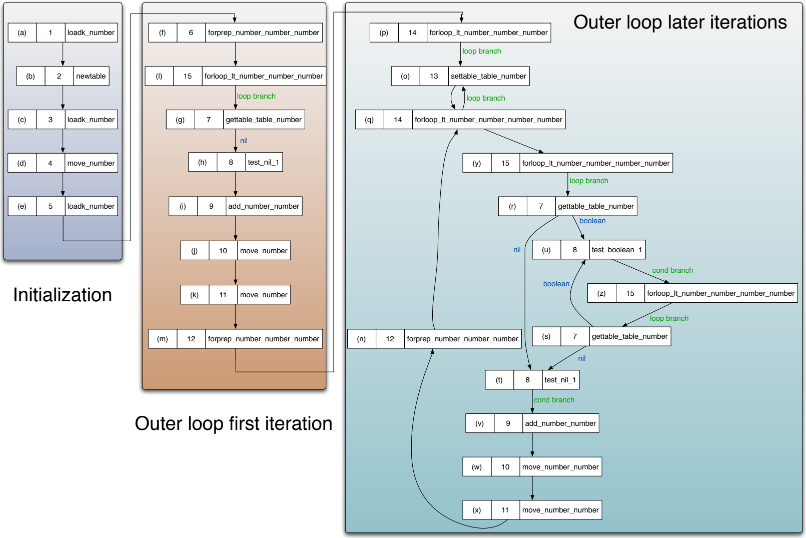
Static IR

```
 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
14 forloop  r6 L13  ; if loop, goto[13] else goto[15]
15 forloop  r2 L7   ; if loop, goto[7]  else [end]
```

‘Imaginary’ Static IR graph



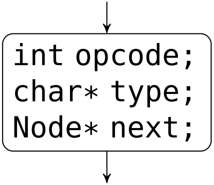
Dynamic IR graph



Dynamic IR

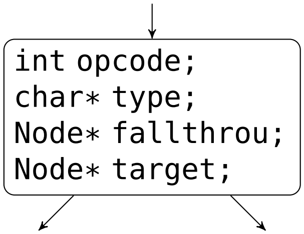
- Dynamic IR a flow graph
    - Nodes represent specialized instructions
    - Edges represent either control-flow or type-flow
- Every type change results in a new path -> all variable types are known at every point in execution
- Construction of the dynamic flow graph guided by the Static IR

Standard Node



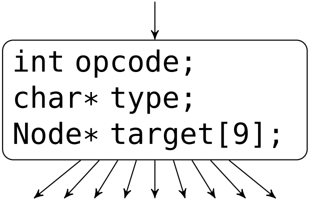
- Always result in same control and type flow

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

```
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;
```

- Represent a call site and known set of parameter types

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

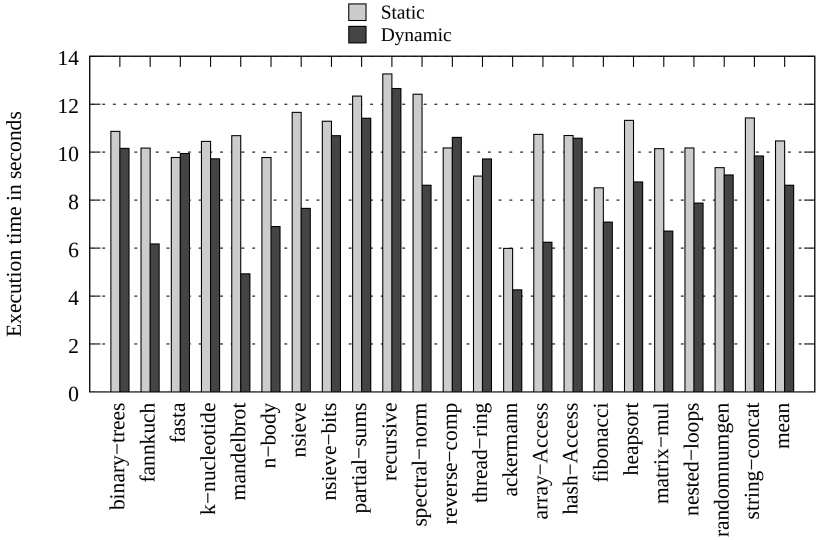
Micros

- We use a set of kernel benchmarks taken from CLBG and GWCLS

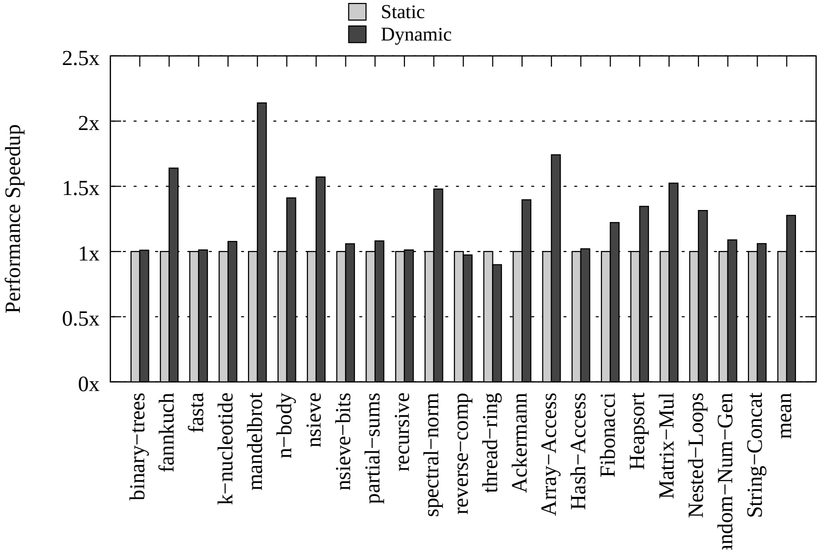
Experimental Setup

- Intel Xeon dual core 2.13GHz

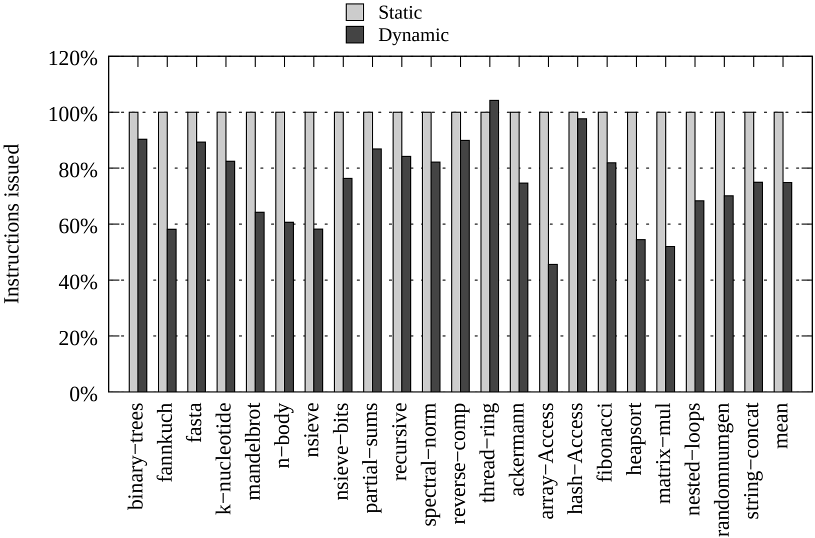
Performance



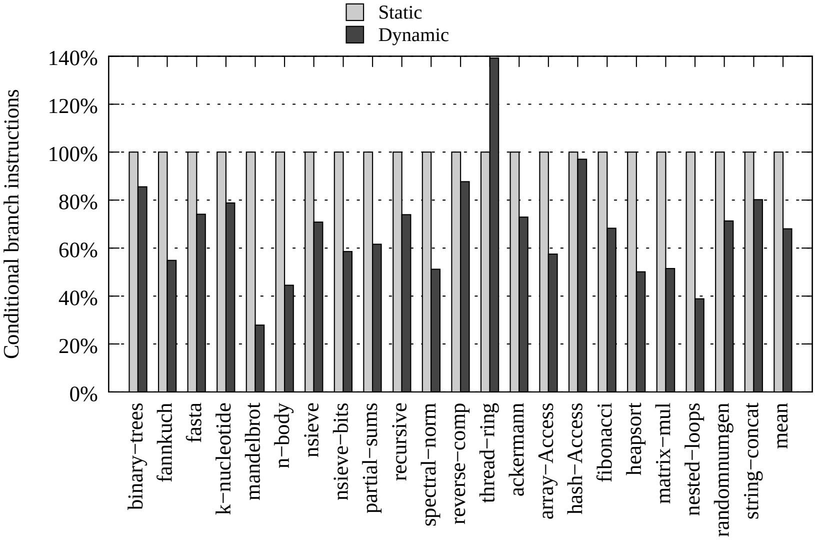
Performance



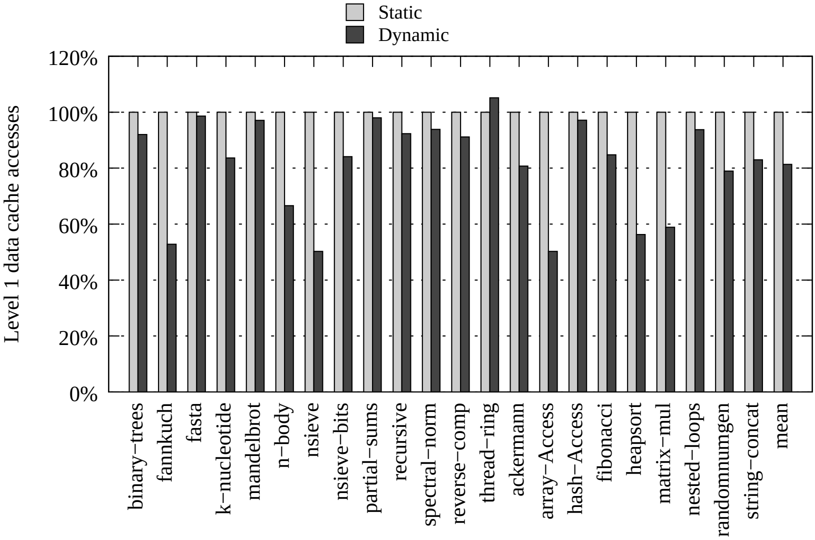
x86 Machine Instructions Issued



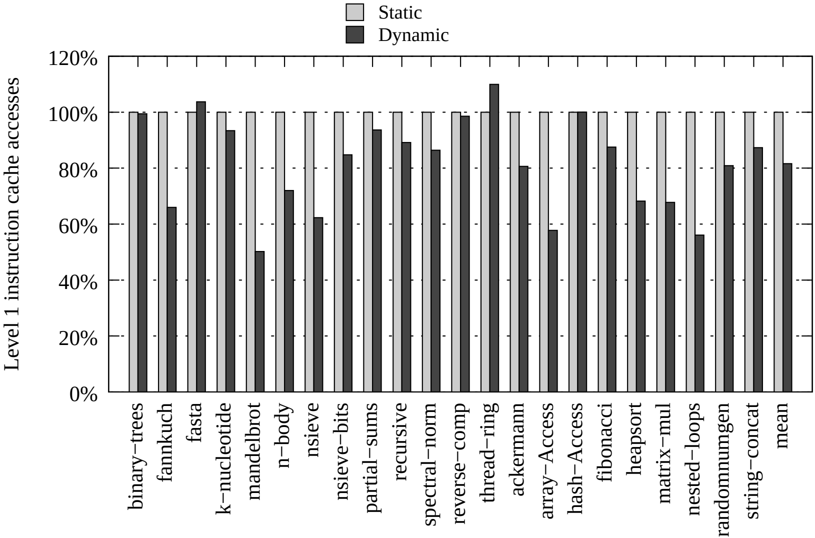
Conditional Branch Instructions



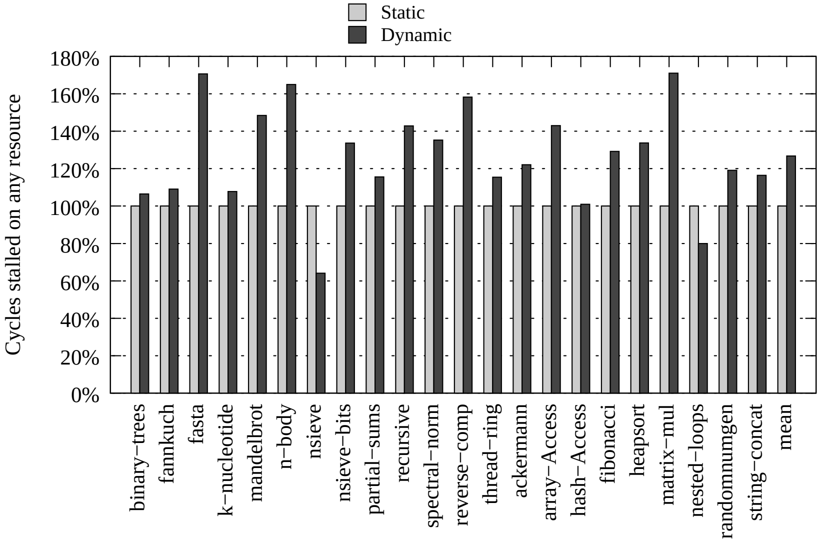
L1 Data Cache



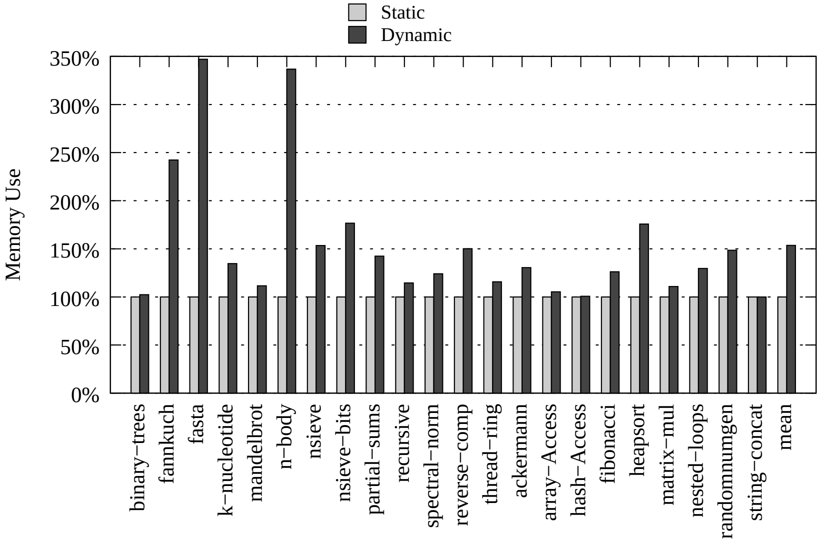
L1 Instruction Cache



Cycles Stalled On Any Resource



Memory Use


Static
Dynamic
Memory Use
350%
300%
250%
200%
150%
100%
50%
0%
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
randomnumgen
string−concat
mean

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 commons pairs of instructions to form a single instruction

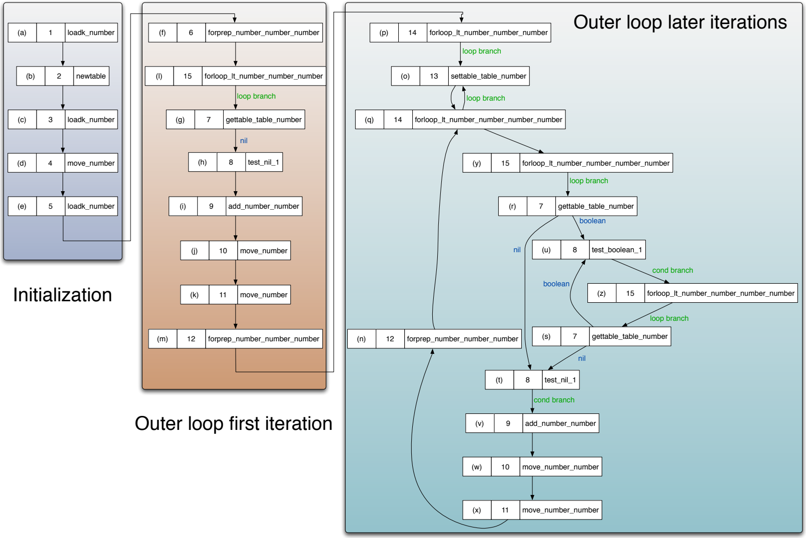
Loop Optimizations

- 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



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?