Portable Just-in-time Specialization of Dynamically Typed Scripting Languages

Kevin Williams, **Jason McCandless**, David Gregg

Trinity College Dublin

October 5, 2011

embarkinitiative
Investing in People and Ideas

IBM

Introduction

- Portable approach to JIT compilation for dynamically typed scripting languages

Motivation

- Scripting languages growing fast. Run slower than staticly typed languages (type checks)
- Interpreters for scripting languages are portable
- Lua is a popular portable scripting language (ANSI C)

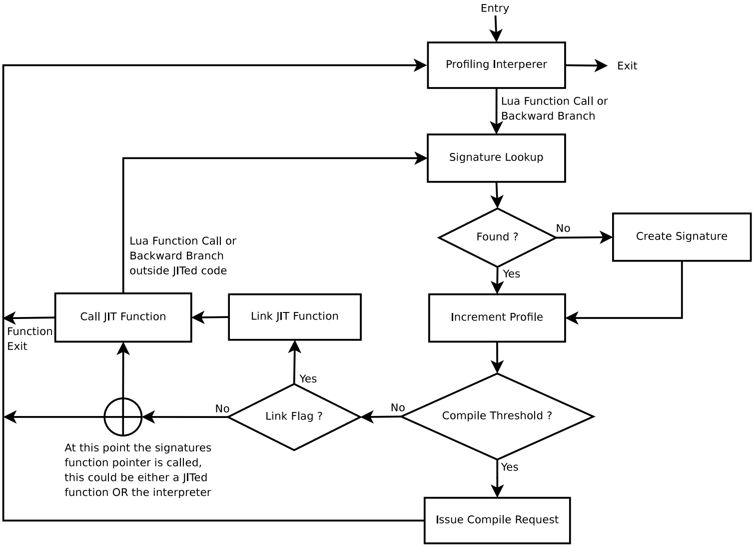
Motivation

- JIT helps speed, breaks portability
- Past JIT for scripting only focused on whole method compilation — *'big bang'* pattern of program performance

Our Aims

- Create a profiling mechanism to profile running programs at multiple levels of granularity
- Allow the use of a native C compiler as a JIT compiler
- Hiding the cost of native C compiler using parallel approach

VM Execution Flow



Profiling Architecture

- Modified version of regular Lua interpreter
- Profile count of each set of types for each function call and backward branch (profile signature)
- Each function stores signature list
- Function call/backward branch creates new entry in list, increments profile count

Profile signature

NEXT SIGNATURE	INSTRUCTION INDEX	NUMBER OF LOCALS	LOCAL TYPES	FUNCTION POINTER	PROFILE COUNT	LINK FLAG

State of signature triggers compilation, linking and dispatch of JIT functions

Profile signature

NEXT SIGNATURE	INSTRUCTION INDEX	NUMBER OF LOCALS	LOCAL TYPES	FUNCTION POINTER	PROFILE COUNT	LINK FLAG

State of signature triggers compilation, linking and dispatch of JIT functions

Profile signature

NEXT SIGNATURE	INSTRUCTION INDEX	NUMBER OF LOCALS	LOCAL TYPES	FUNCTION POINTER	PROFILE COUNT	LINK FLAG

State of signature triggers compilation, linking and dispatch of JIT functions

Thread Communication

- Producer/consumer communication
- Use link flag to signal compilation completed

Thread Communication

- Scripting languages typically implement automatic memory management
- Danger of function being garbage collected
- Implemented a queueing data structure accessible to VM's garbage collector
- Queue managed by both threads

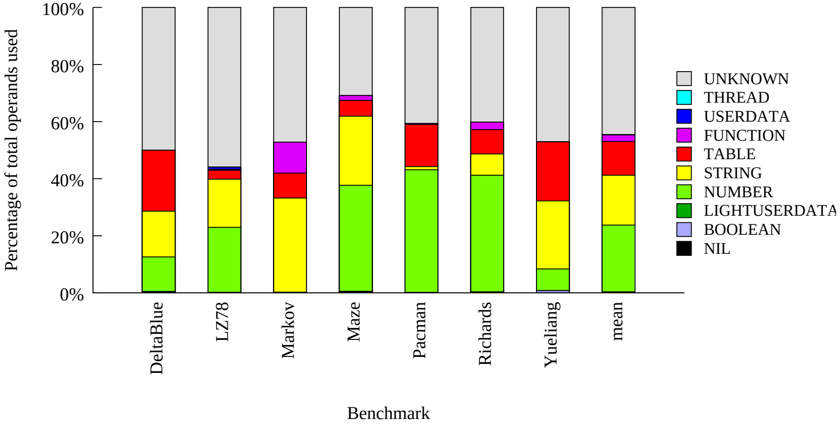
Type Inference

- Types taken from signature
- Simple interprocedural worklist
- Type inference identifies a type for each variable at every execution point from a set of known inputs
- Outputs a flow graph
- Types of any Lua primitive type, or UNKNOWN

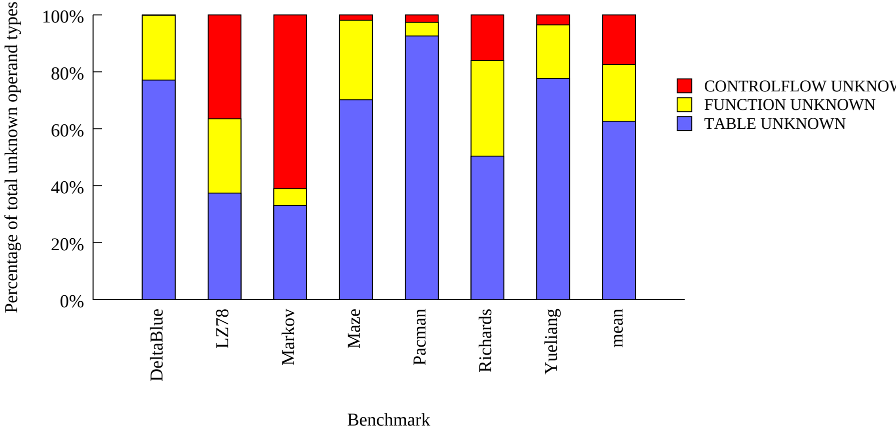
Type Inference

- Iterative algorithm infers types from predecessor nodes
- Any conflicts in the sets results in a type becoming UNKNOWN
- Table structure values always unknown

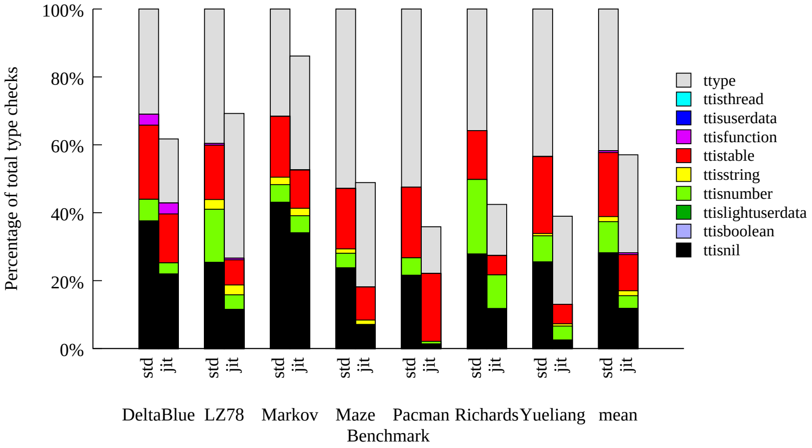
Type Inference Performance



Type Inference Performance



Type Inference Performance



Optimizations

- Code generator performs linear pass
- Specialized implementation for each virtual instruction
- Specialization performed using operand values decoded from opcode

Control Flow and Interpreter Overhead

- Instruction operards access using bit operations, virtual registers referenced after decode
- Code generation outputs decoded implementations of all instructions
- Control flow generation using direct branching (`goto`)

Type Check Removal

- 3 kinds of type check:
    - Artithmetic/string operations $\text{A} = \text{B OP C}$
    - Conditional branches
    - Table accesses — table+key

Lua Register Variables to Native C Variables

- Lots of overhead in accessing Lua variable
- Code generator generates code to directly access C variables

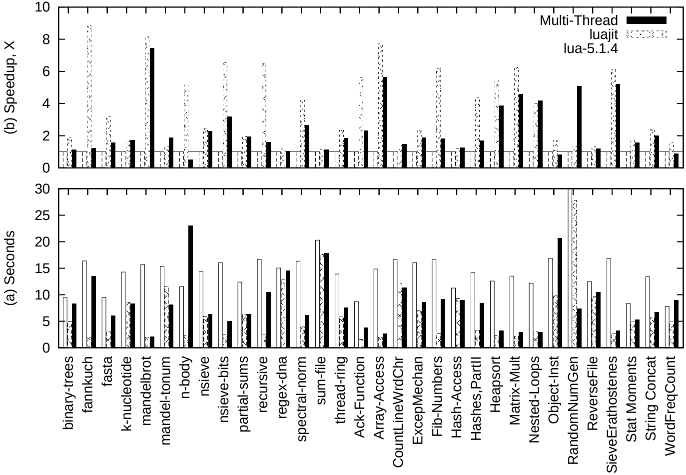
Lua Function Calls

- Naively, call from a JITed Lua function to another JITed Lua function requires lookup
- Code generator generates direct calls to compiled Lua functions with necessary guards

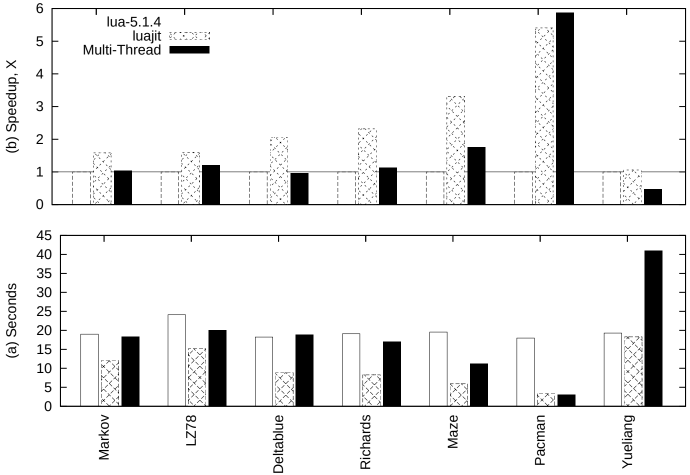
Benchmarks

- No real standard for scripting language benchmarks
- Compared against LuaJIT — extremely fast x86 JIT compiler

Micro Benchmarks Performance



Application Performance



Conclusions

- Type analysis at these levels of granularity produces a large number of known types
- Type data collected has shown variables fetched from tables to be the most common source of unknown types
- Demonstrated the practicalities of using external compilers in JIT systems

Discussion

Three weaknesses with this system:

- Cost of gathering types expensive
- Not knowing what comes out of table
  - Could keep one type for the table, updating type on store
- Calls are expensive

Future Work

- Dynamic intermediate representation
- Full type knowledge of all live local variables
- Interpreter optimisation as well as code generation

Thanks

Questions?