Virtual Machine Showdown: stack versus registers

Yunhe Shi[1], Kevin Casey[1], Anton Ertl[2], **David Gregg**[1]

[1]Department of Computer Science
Trinity College Dublin

[2]Institut für Computersprachen
Technische Universität Wien

This talk is based on a paper in *ACM TACO* 4(4), January 2008

Stack Architecture

- Almost all real computers use a register architecture
  - Values loaded to registers
  - Operated on in registers
- But most popular VMs use stack architecture
  - Java VM, .NET VM, Pascal P-code, Perl 5

Virtual Machines (VM)

- High-level language VMs
  - Popular for implementing programming languages
    - Java, C#, Pascal, Perl
- Program is compiled to virtual machine code
  - Similar to real machine code
  - But architecture neutral
- VM implemented on all target architectures
  - Using interpreter and/or JIT compiler
  - Same VM code then runs on all machines

Why stack VMs?

- Code density
  - No need to specify register numbers
- Easy to generate stack code
  - No register allocation
- No assumptions about number of registers
  - ????
- Speed
  - May be easier to JIT compile
  - May be faster to interpret
    - Or maybe not…

Which VM interpreter is faster?

- Stack VM interpreters
  - Operands are located on stack
  - No need to specify location of operands
  - No need to load operand locations
- Register VM interpreters
  - Fewer VM instructions needed
    - Less shuffling of data onto/off stack
  - Each VM instruction is more expensive

VM Interpreters

- Emulate a virtual instruction set
- Track state of virtual machine
  - Virtual instruction pointer (IP)
  - Virtual stack
    - Array in memory
    - With virtual stack pointer (SP)
  - Virtual registers
    - Array in memory
    - No easy way to map virtual registers to real registers in an interpreter

Which VM interpreter is faster?

- Question debated repeatedly over the years
  - Many arguments, small examples
  - No hard numbers
- Some are confident that answer is obvious
  - But which answer?

VM Interpreters

```
while ( 1 ) {
  ip++;
  opcode = *ip;
  switch ( opcode ) {
    case IADD:        *(sp-1) = *sp + *(sp-1); sp--; break;
    case ISUB:        *(sp-1) = *sp – *(sp-1); sp--; break;
    case ILOAD_0:     *(sp+1) = locals[0]; sp++; break;
    case ISTORE_0:    locals[0] = *sp; sp--; break;
    ………
  }
}
```

VM Interpreters

- Dispatch
  - Fetch opcode & jump to implementation
  - Most expensive part of execution
  - Unpredictable indirect branch
  - Similar cost for both VM types
  - But register VM needs fewer dispatches
- Fetch operands
  - Locations are explicit in stack machine
- Perform the operation
  - Often cheapest part of execution

Operand Access

- Stack machine
  - Virtual stack in array
  - Operands on top of stack
  - Stack pointer updates
- Register machine
  - Virtual registers in array
  - Must fetch operand locations (1-3 extra bytes)
    - More loads per VM instruction

Stack versus registers

- Our register VM
  - Simple translation from JVM bytecode
  - One byte register numbers

Source code	**Stack code**	**Register code**
a = b + c;	iload b; iload c; iadd; istore a;	iadd a, b, c

From Stack to Register

- Translated JVM code to register VM
- Local variables mapped directly
  - Local 0 → Register 0
- Stack locations
  - Mapped to virtual registers
  - Height of stack is always known statically
  - Assign numbers to stack locations

From Stack to Register

Stack Code	Register Code	Comment
iload 4	imove r10, r4	; load local variable 4
bipush 57	biload r11, 57	; push immediate 57
iadd	iadd r10, r10, r11	; integer add
istore 6	imove r6, r10	; store TOS to local 6
iload 6	imove r10, r6	; load local variable 6
ifeq 7	ifeq r10, 7	; branch by 7 if TOS==0

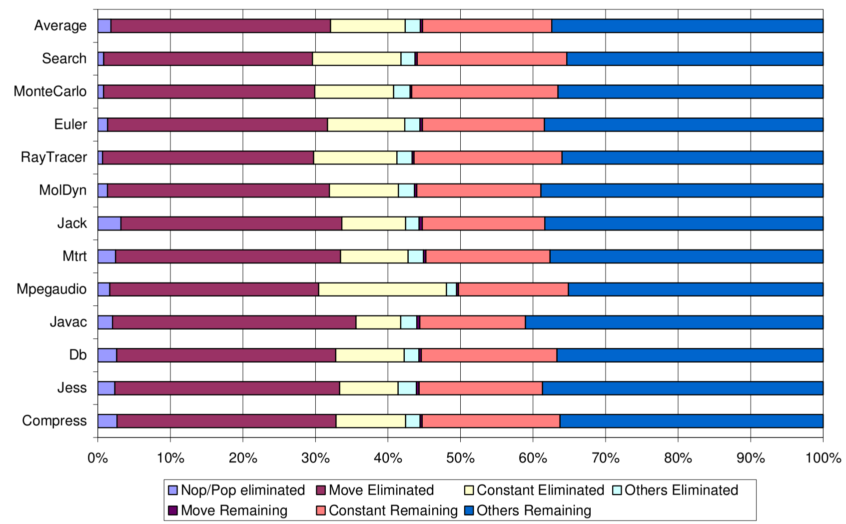
From Stack to Register

- Clean up register code with classical optimizations
  - Copy propagation to remove unnecessary move operations
  - Partial redundancy elimination
    - Re-use constants already in registers
    - Stack VM consumes its operands so must load constants every time it uses them

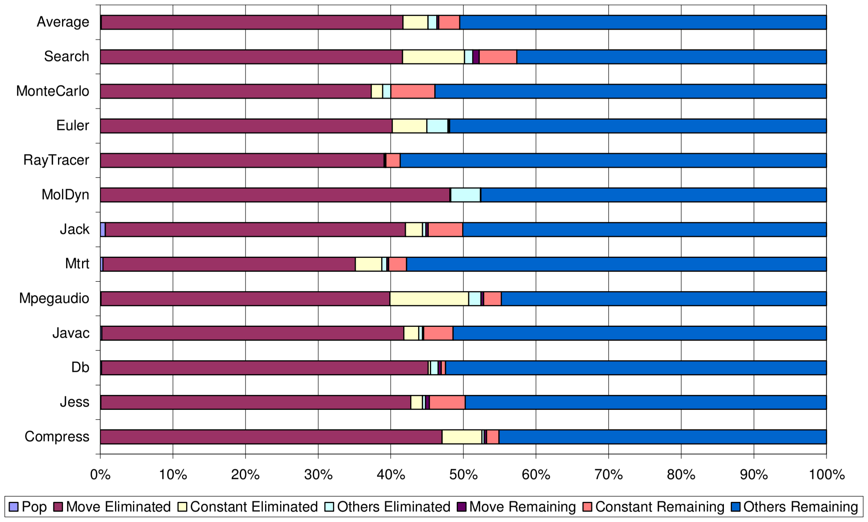
Experimental Setup

- Implemented in Cacao VM
- Method is JIT compiled to register code on first invocation
  - Results include only executed methods
- Standard benchmarks
  - SPECjvm98, Java Grande
- Real implementation wouldn't translate
  - Better generate register code from source
  - But translation allows fairer comparison
    - Except for translation time

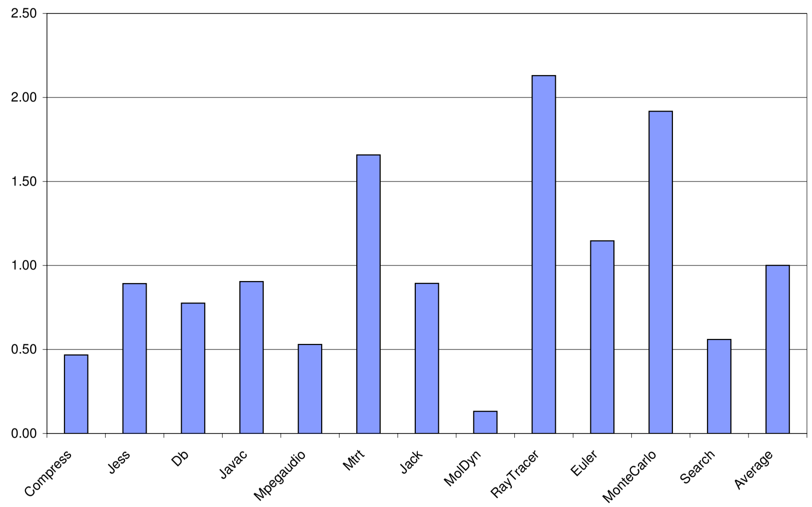
Static VM Instructions



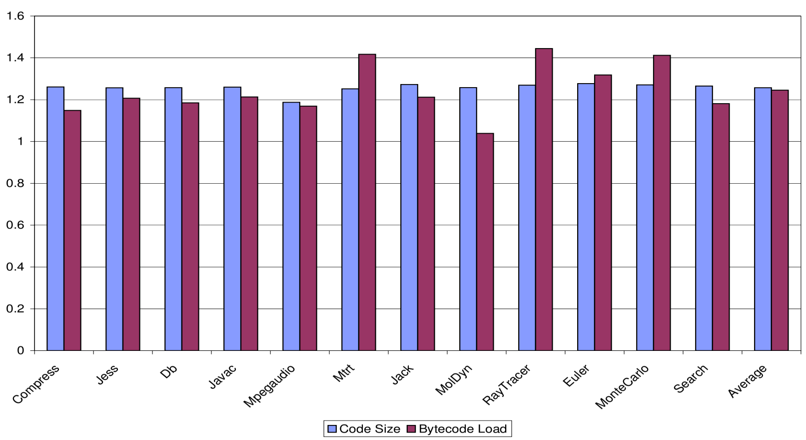
Dynamic VM Instructions



Ratio of additional loads to eliminated instructions



Increase in bytecode loads



Real machine memory ops

Source Code

```
a = b + c;
```

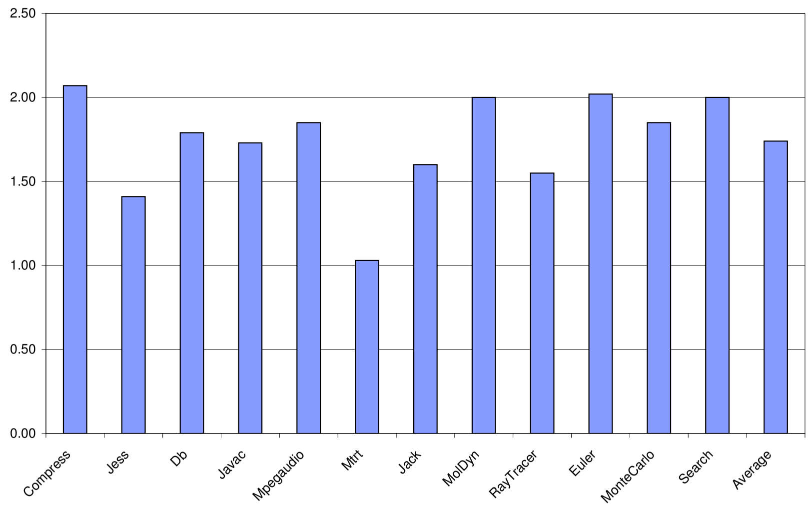
Register Code

```
/* iadd a, b, c */
reg[a] = reg[b] + reg[c];
```

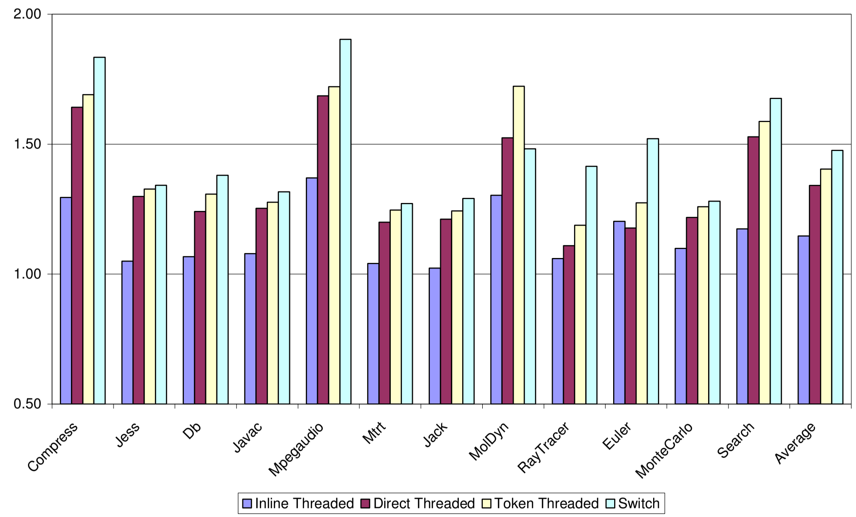
Stack Code

```
/* iload c */
*(++sp) = locals[c];

/* iload b */
*(++sp) = locals[b];

/* iadd */
*(sp-1) = *(sp-1) + *sp;
sp--;

/* istore a */
locals[a] = *(sp--);
```

Reduction in "real machine" loads/stores compared with dispatches eliminated



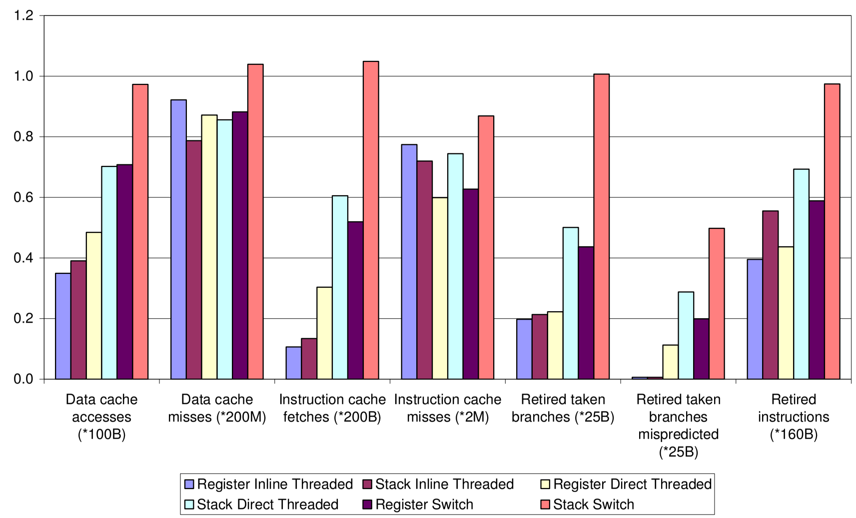
Speedup of Register VM - AMD64



Real Running Times

- Interpreter Dispatch
  - Switch dispatch
  - Token Threaded dispatch
  - Direct threaded dispatch
  - Inline threaded dispatch
- Hardware platforms
  - AMD 64
  - Intel P4
  - Intel Core 2 Duo
  - Digital Alpha
  - IBM PowerPC

AMD64 Event Counters - Compress



Eliminating more redundant expressions

- Stack operations consume their operands
  - So very difficult to re-use existing values
  - Stack machine must load constants, loop invariants repeatedly
  - Register machine can store constants, simple loop invariants in registers
- What about more complex invariants
  - Repeated loads from the heap
  - Requires very sophisticated pointer analysis
    - But what if we could do it?

Eliminating more redundant expressions – speedup on AMD 64

Java VM Summary

- Detailed quantitative results
  - 46% reduction in executed VM instructions
  - 26% increase in bytecode size
  - 25% increase in bytecode loads
- Speedup depends on dispatch scheme
  - Speedup 1.48 with switch dispatch on AMD64
  - Even with the most efficient dispatch, 1.15 speedup can still be achieved

What about Forth?

- Forth usually uses stack VM
- But execution profile very different
- Java instructions:
  - 42% load & stores of locals
  - 6% loads of constants
  - 0-2% stack manipulation
- Very many local load/store
  - Almost all disappear in register VM

What about Forth?

- Forth VM instructions
  - Stack manipulation instructions
    - over, dup, swap, drop, 2dup, ?dup, r>, >r, i
    - maybe 10%-15% ???
  - Literal instructions
    - lit, var
    - maybe 15%-25% ???
  - Local variable instructions
    - >l, @local
    - maybe 2%-5% ???

What about Forth?

- There is no huge block of instructions that will easily disappear using a register VM
  - Apart from literals
- But some speedup is probably possible by using a register VM