Virtual Machine Showdown: Stack versus Registers

David Gregg

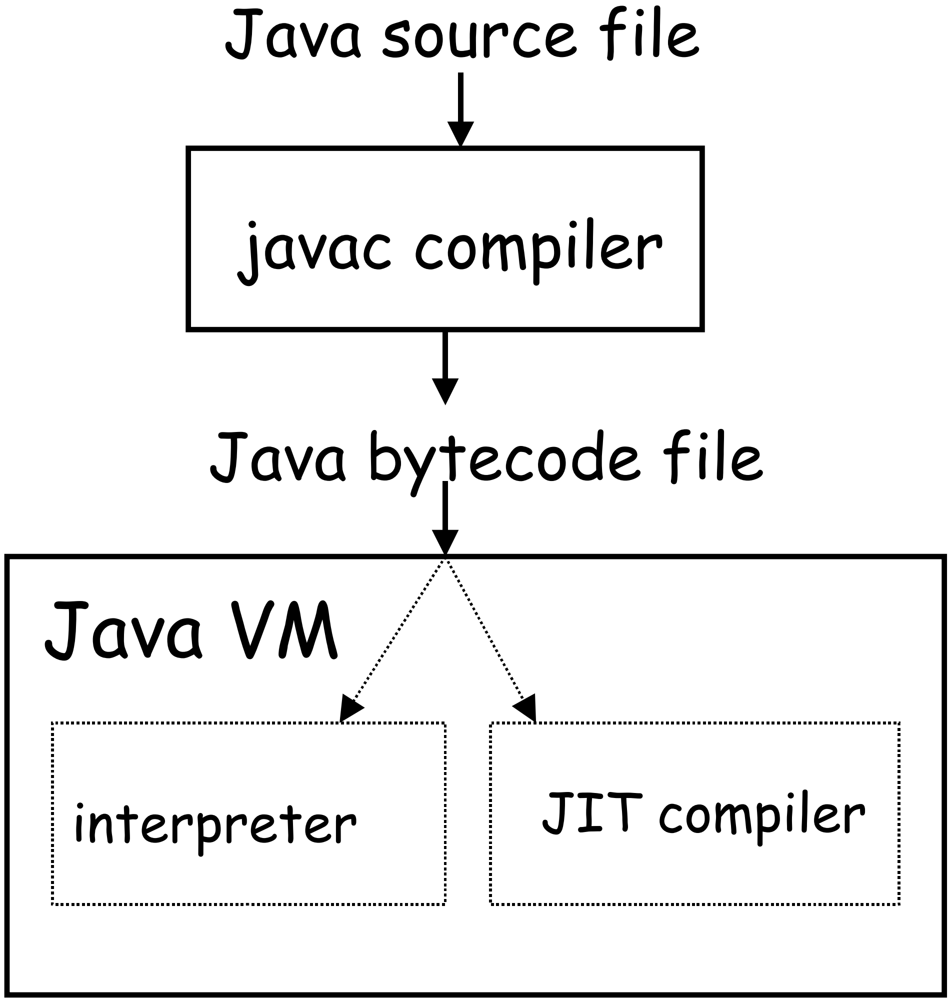
Software Systems Lab

Yunhe Shi, Kevin Casey (TCD), Anton Ertl (TU Wien)

Virtual Machines (VM)

- High-level language VMs
  - Popular for implementing programming languages
    - Java, C#, Pascal, Perl, Lua
- 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

Virtual Machines (VM)



- Interpreter is program that emulates VM
- Just-in-time (JIT) compiler translates bytecode to real machine code

Why use interpreters?

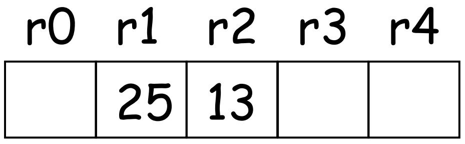
- Huge advances in JIT compilers in last 15 years
  - Faster compilation
  - Better optimization
- But interpreters are still hugely popular for implementing VMs
  - Why do engineers still build interpreters?

Why build interpreters?

- Speed
  - Hybrid JIT/interpreter implementations
  - Real-time systems
- Memory
  - Size of VM and generated code
  - Hybrids, embedded systems, code compression
- Software engineering
  - Portability
  - Simplicity: cost, correctness
  - Safety: sandboxing of executing code
  - Tools: debugger, profiler
  - Dynamic language features

Stack Architecture

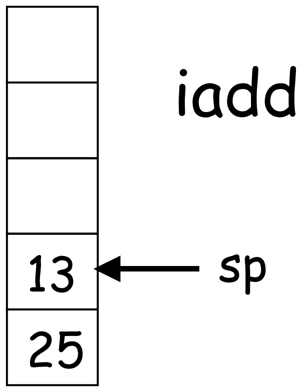
- Almost all real computers use a register architecture
  - Values loaded to registers
  - Operated on in registers



r0	r1	r2	r3	r4
	25	13		

iadd r0, r1, r2

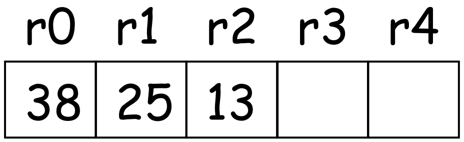
- But most popular VMs use stack architecture
  - Java VM, .NET VM, Pascal P-code, Perl 5



iadd

Stack Architecture

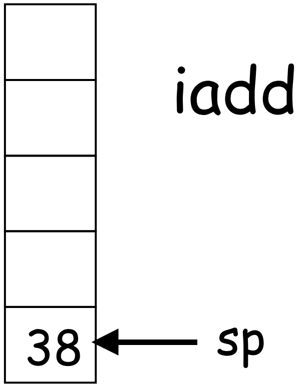
- Almost all real computers use a register architecture
  - Values loaded to registers
  - Operated on in registers



r0	r1	r2	r3	r4
38	25	13		

iadd r0, r1, r2

- But most popular VMs use stack architecture
  - Java VM, .NET VM, Pascal P-code, Perl 5



iadd

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

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

- 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

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
  - Usually 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

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;	iadd a, b, c
	iload c;	
	iadd;	
	istore a;	

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

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

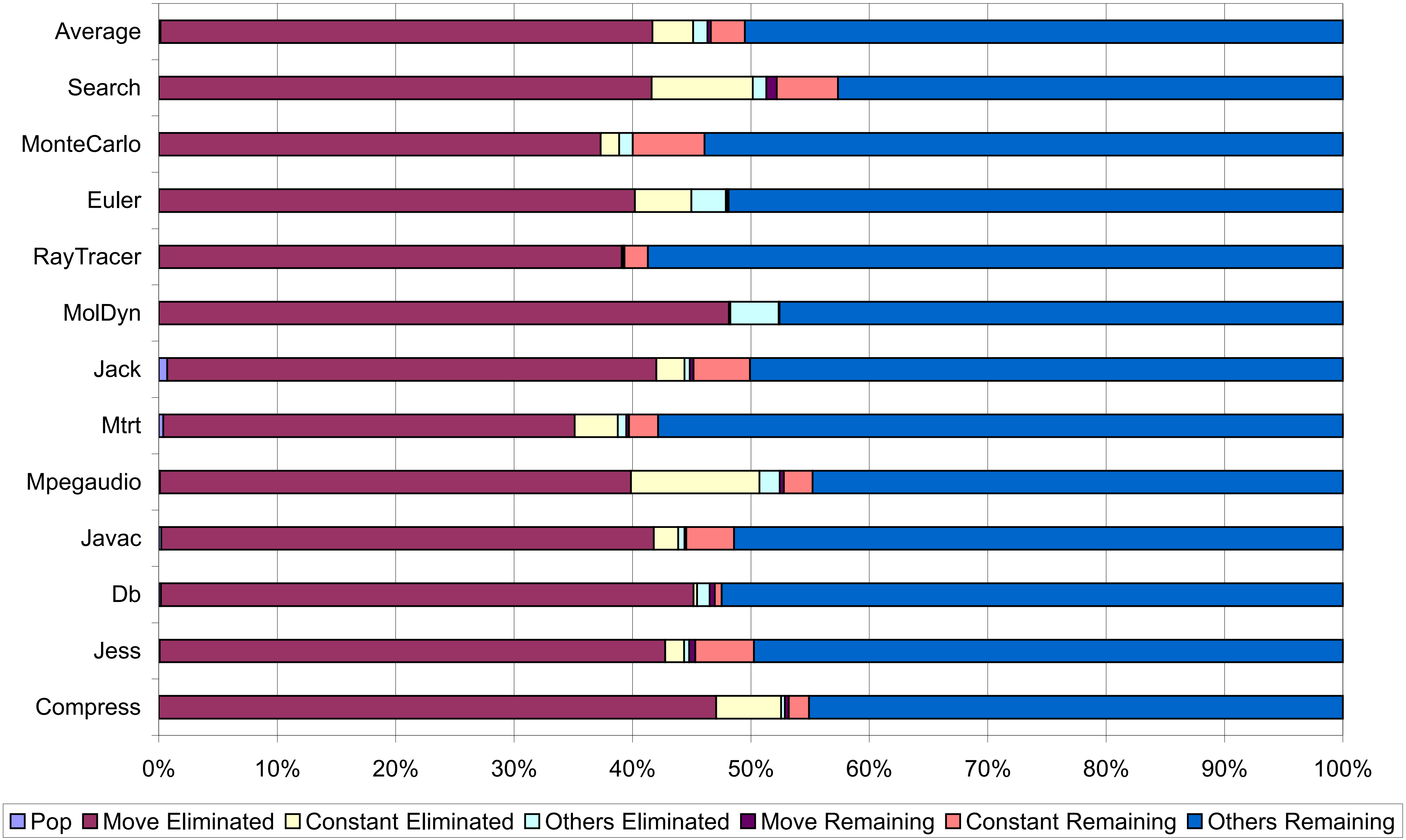
From Stack to Register

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

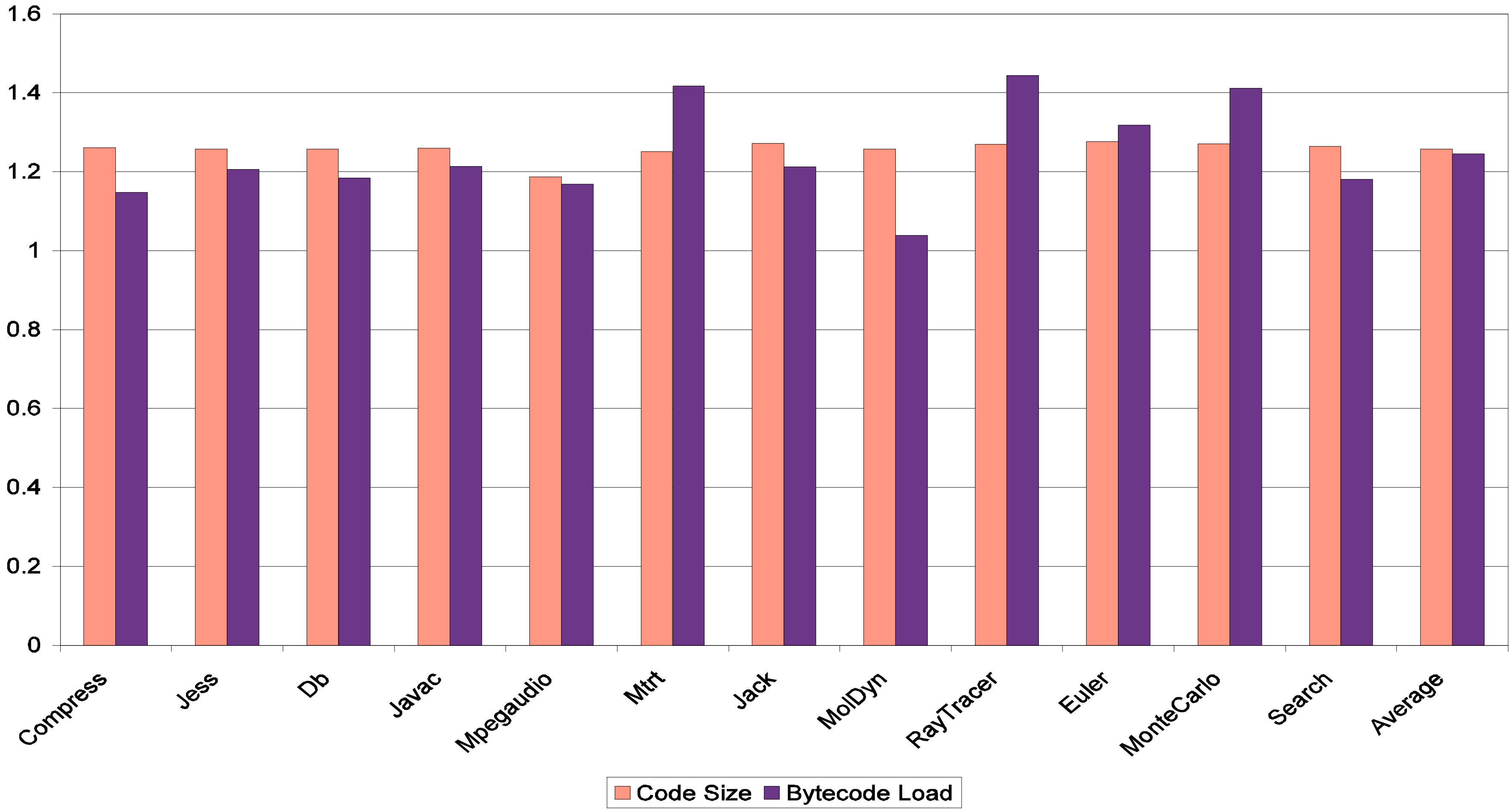
Experimental Setup

- 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

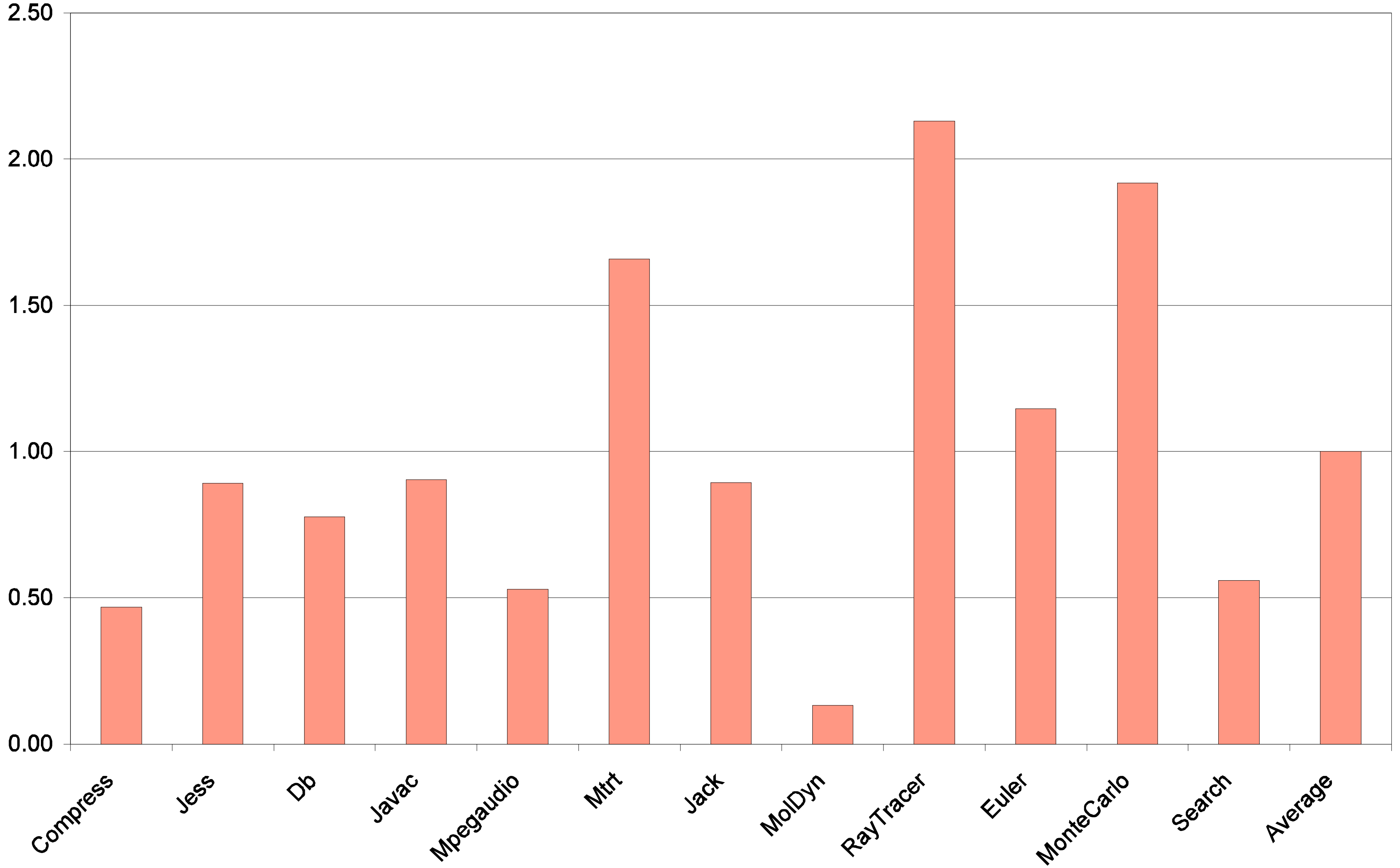
Executed VM Instructions



Increase in bytecode loads



Ratio of additional loads to eliminated instructions



Real machine memory ops

Source Code

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

Register Code

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

Stack Code

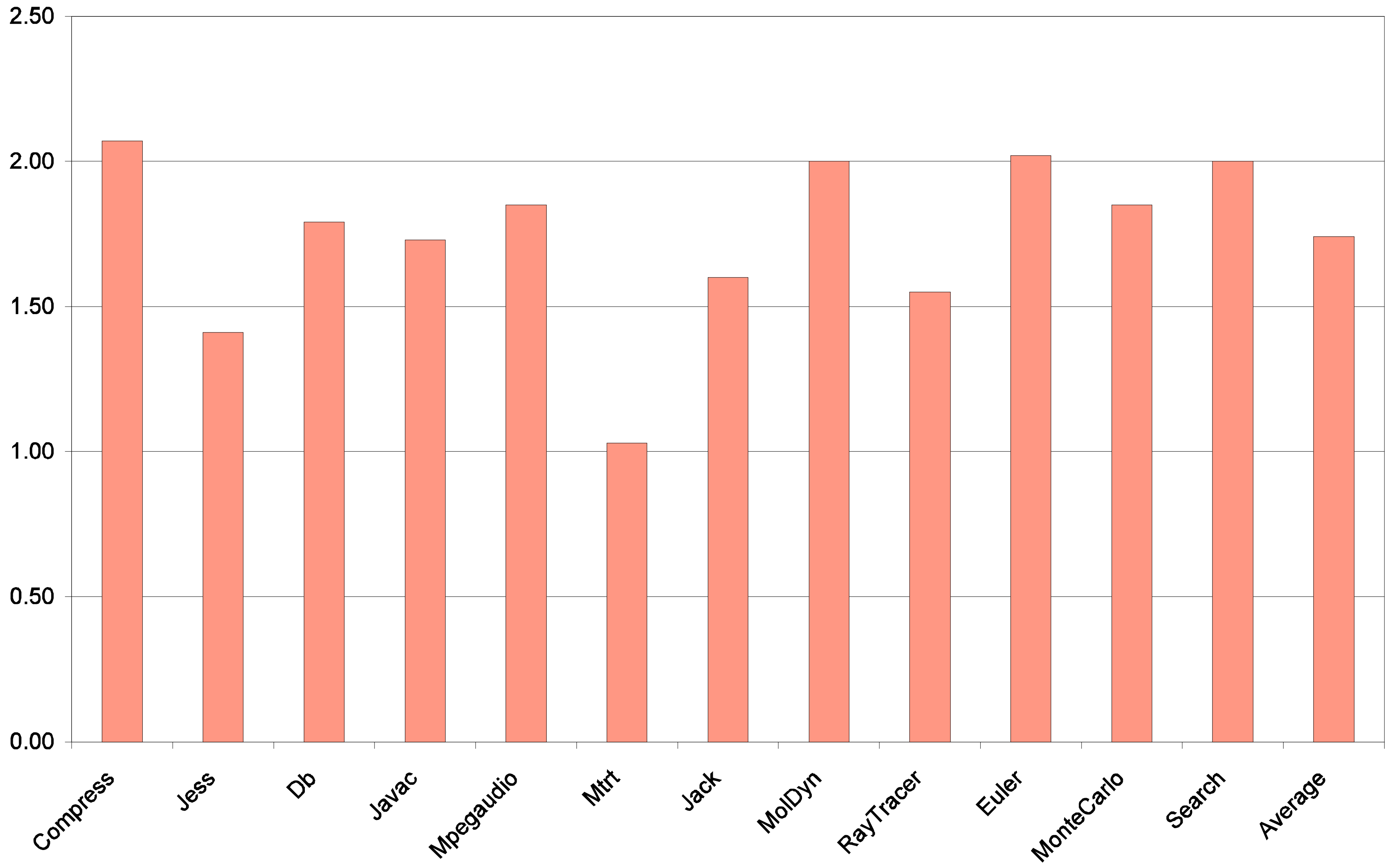
```
/* iload c */
*(++sp) = locals[c];

/* 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



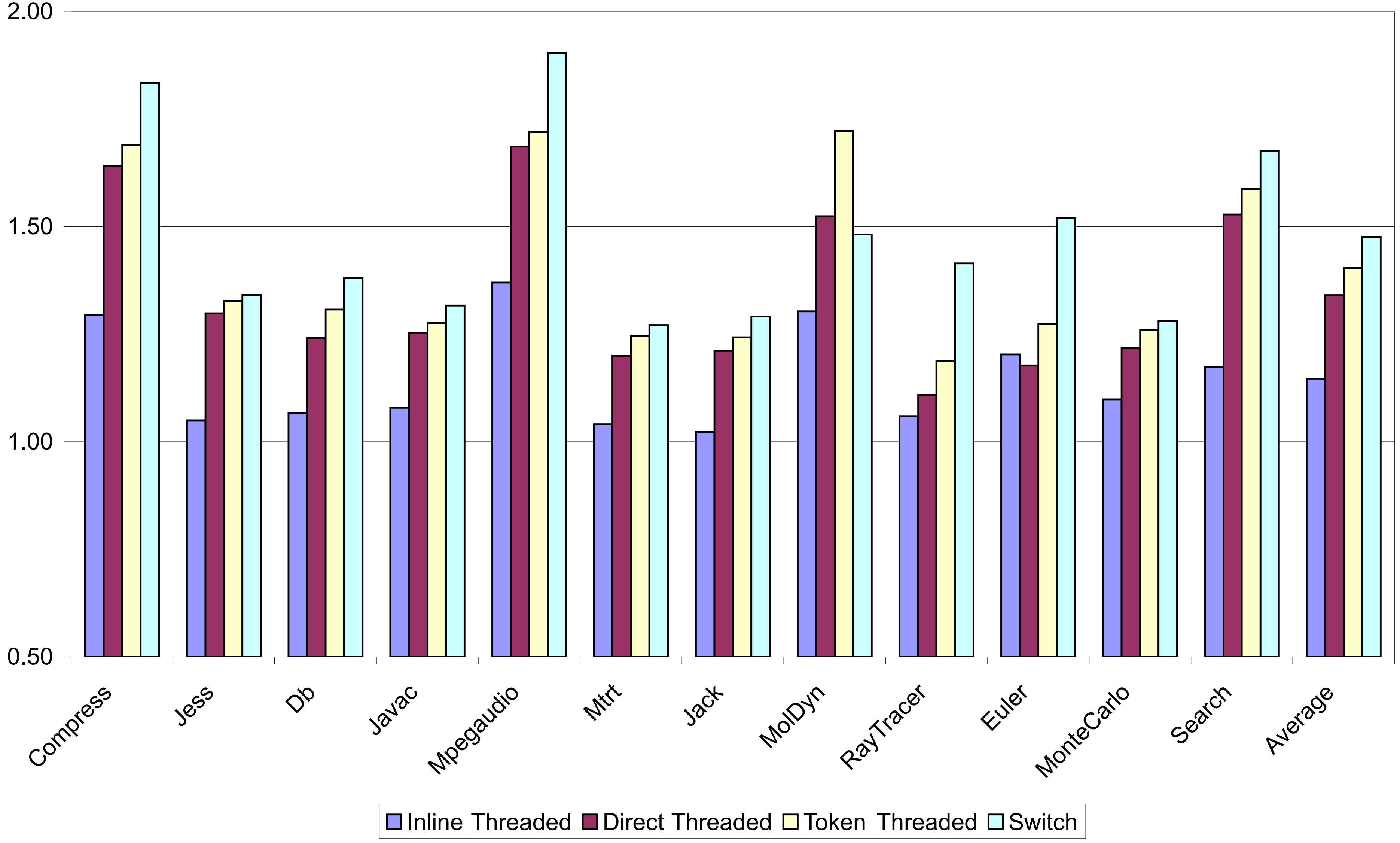
Real Running Times

- Hardware platforms
  - AMD 64
  - Intel P4
  - Intel Core 2 Duo
  - Digital Alpha
  - IBM PowerPC

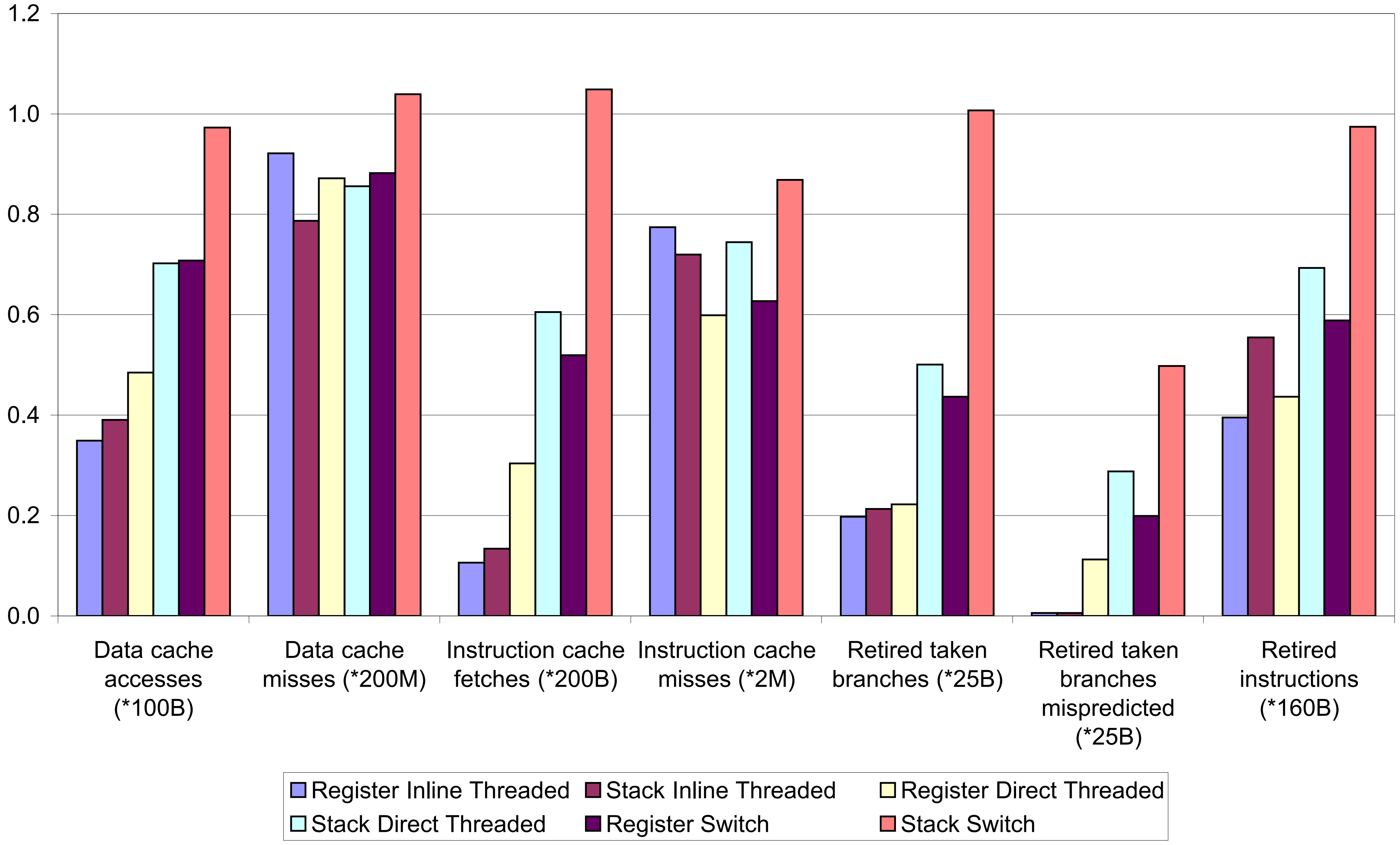
Interpreter Dispatch

- Switch dispatch
  - Large case statement in a loop
- Token Threaded dispatch
  - Interpreter is set of routines that jump from one to another
  - Requires table of machine code addresses
- Direct threaded dispatch
  - Replace bytecode with sequence of machine code addresses
- Inline threaded dispatch
  - Simple macro-expanding JIT compiler

Speedup of Register VM - AMD64



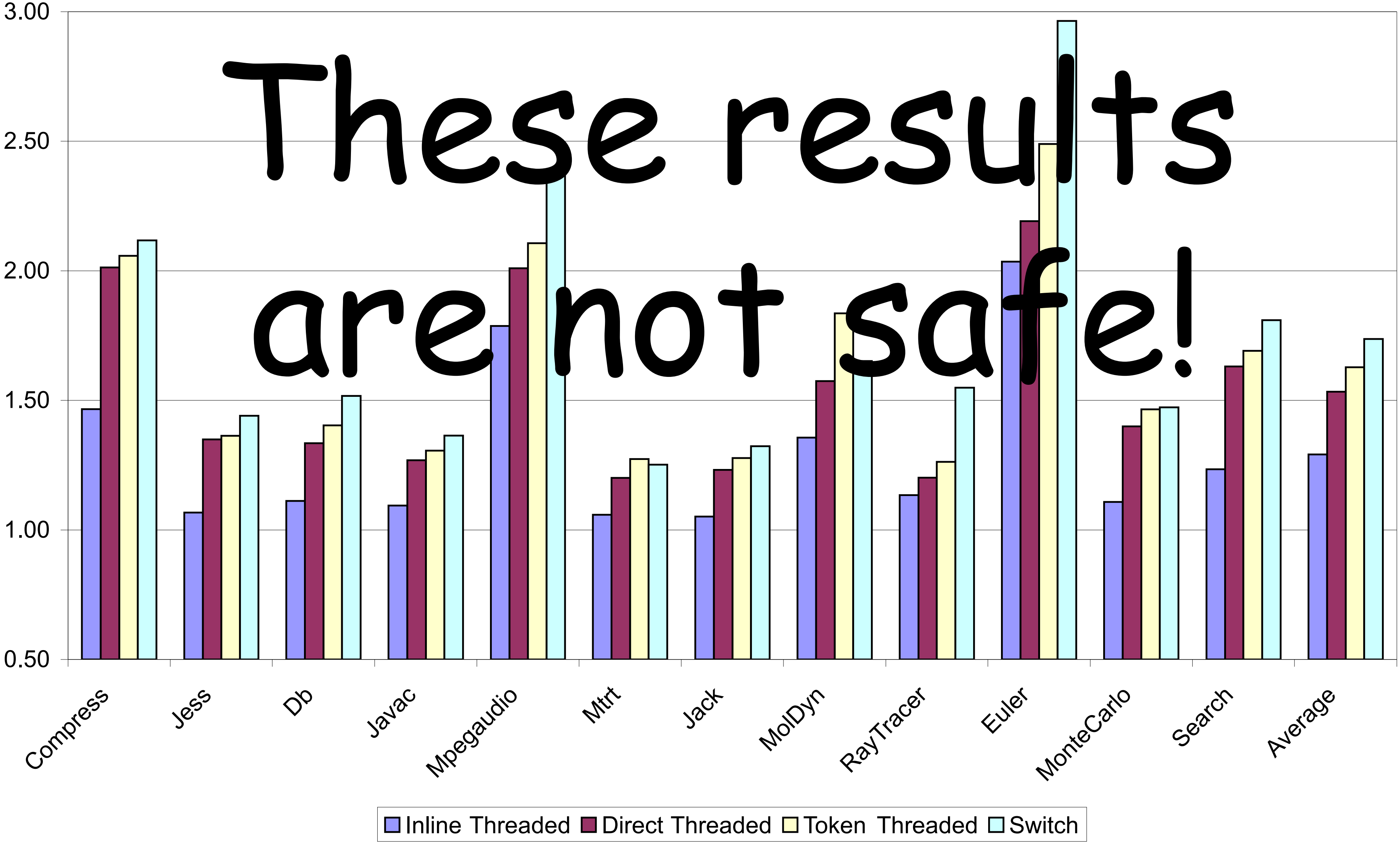
AMD64 Event Counters - Compress



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



These results are not safe!

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

Real world VMs

- Register vs. stack debate ongoing for many years
  - Perl 6 VM started in 2000
    - Version 1.0 March 2009
    - Partially motivated our work
- Some newish register VMs
  - Lua 5 VM
  - SquirrelFish
  - Google Dalvik VM in Android
  - Unladen Swallow (??)

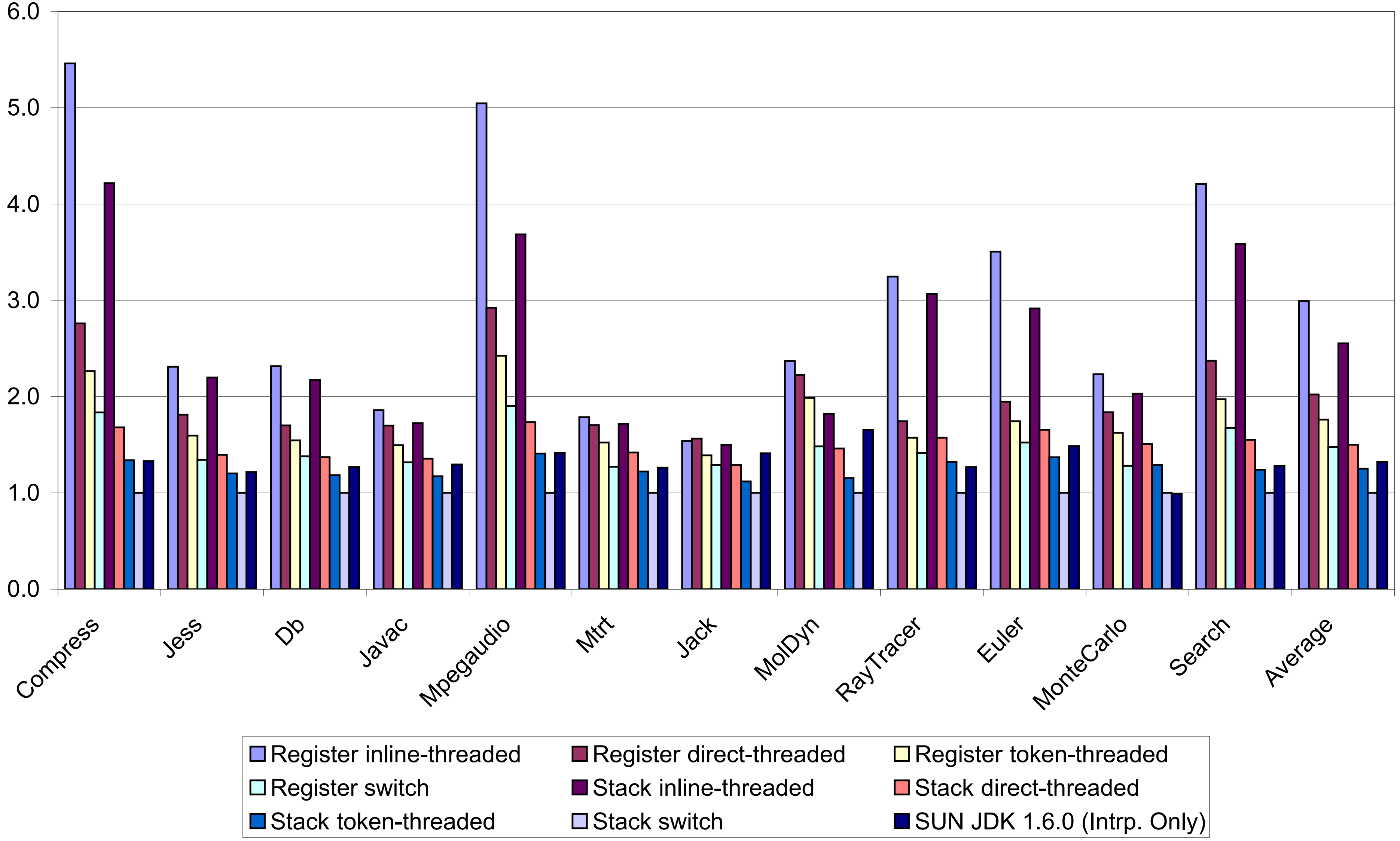
Conclusions

- Register VM offers faster execution of the VM interpreter at the cost of greater bytecode size

- Detailed results can be found in ACMTACO, January 2008

Thank you.

Dispatch Comparison – AMD 64


Compress
Jess
Db
Javac
Mpegaudio
Mtrt
Jack
MolDyn
RayTracer
Euler
MonteCarlo
Search
Average
Register inline-threaded
Register direct-threaded
Register token-threaded
Register switch
Stack inline-threaded
Stack direct-threaded
Stack token-threaded
Stack switch
SUN JDK 1.6.0 (Intrp. Only)