An Efficient Stack Machine

Martin Schöberl

Overview

- JVM stack machine
- Parameter passing
- Stack access patterns
- Common stack caches
- Two-level stack cache
- Results

The Java Virtual Machine

- JVM is a stack machine
- All instructions access the stack
- 40% access local variables
- Stack and local variables need caching

An Efficient Stack Machine

- JVM stack is a logical stack
  - Frame for return information
  - Local variable area
  - Operand stack
- We *could* use independent stacks
- Argument-passing regulates the layout

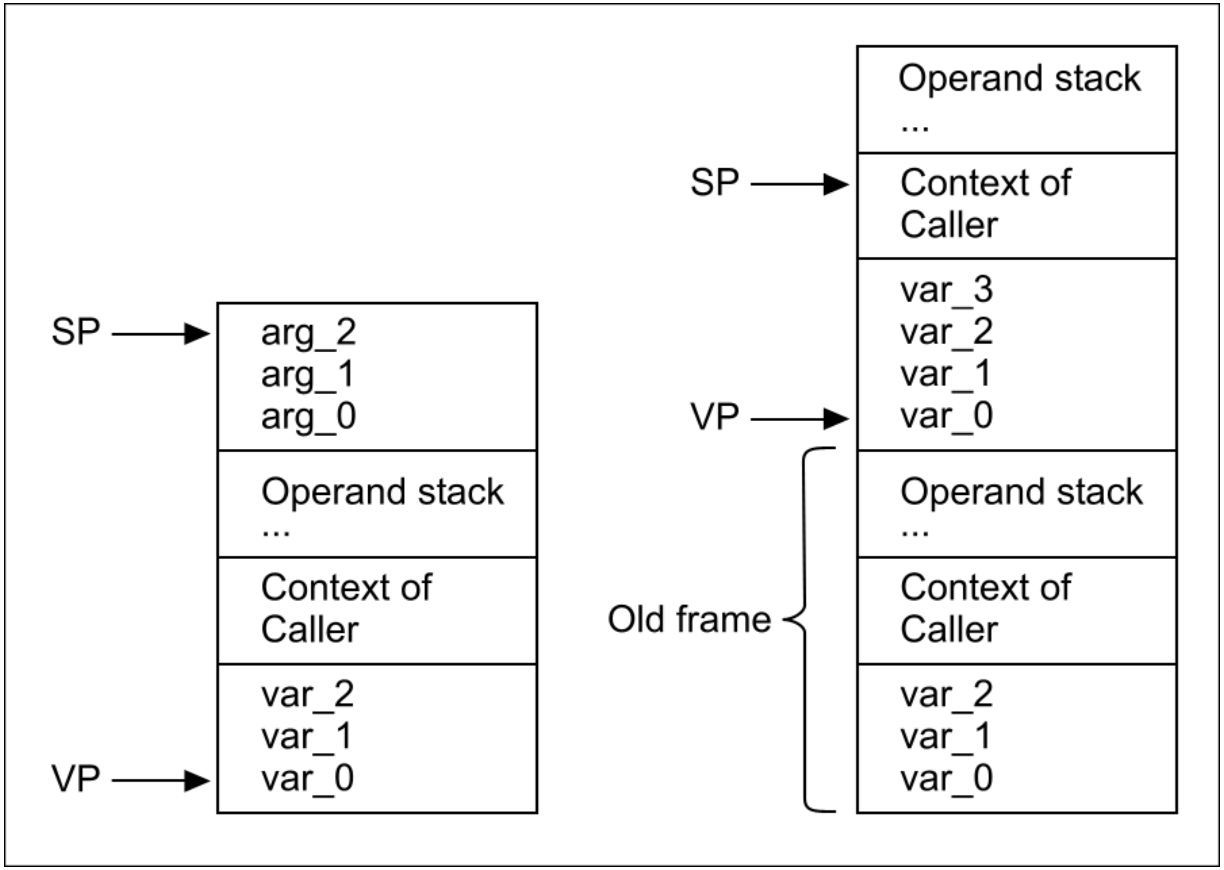
Parameter passing

```
    int val = foo(1, 2);
    ...
    public int foo(int a, int b) {
        int c = 1;
        return a+b+c;
    }

The invocation sequence:
    aload_0             // Push the object reference
    iconst_1            // and the parameter onto the
    iconst_2            // operand stack.
    invokevirtual   #2  // Invoke method foo:(II)I.
    istore_1            // Store the result in val.

public int foo(int,int):
    iconst_1            // The constant is stored in a method
    istore_3            // local variable (at position 3).
    iload_1             // Arguments are accessed as locals
    iload_2             // and pushed onto the operand stack.
    iadd                // Operation on the operand stack.
    iload_3             // Push c onto the operand stack.
    iadd
    ireturn             // Return value is on top of stack.
```

Stack Layout



Stack Content

- Operand stack
  - TOS and TOS-1
- Local variable area
  - Former op stack
  - At a deeper position
- Saved context
  - Between locals and operand stack

A = B + C * D	
Stack	JVM
push B	iload_1
push C	iload_2
push D	iload_3
*	imul
+	iadd
pop A	istore_0

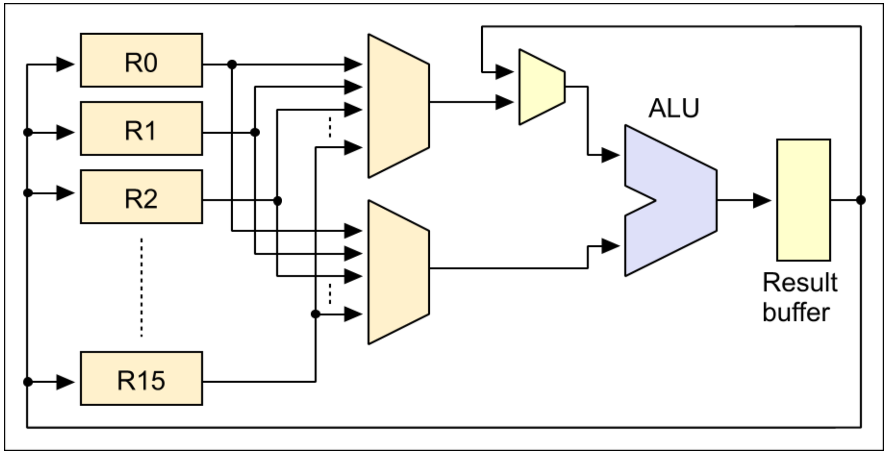
Stack access

- Stack operation
  - Read TOS and TOS-1
  - Execute
  - Write back TOS
- Variable load
  - Read from deeper stack location
  - Write into TOS
- Variable store
  - Read TOS
  - Write into deeper stack location

Three Port Stack Memory

- Single cycle execution
- Two read ports for
  - TOS and TOS-1 or
  - Local variable
- One write port for
  - TOS or
  - Local variable

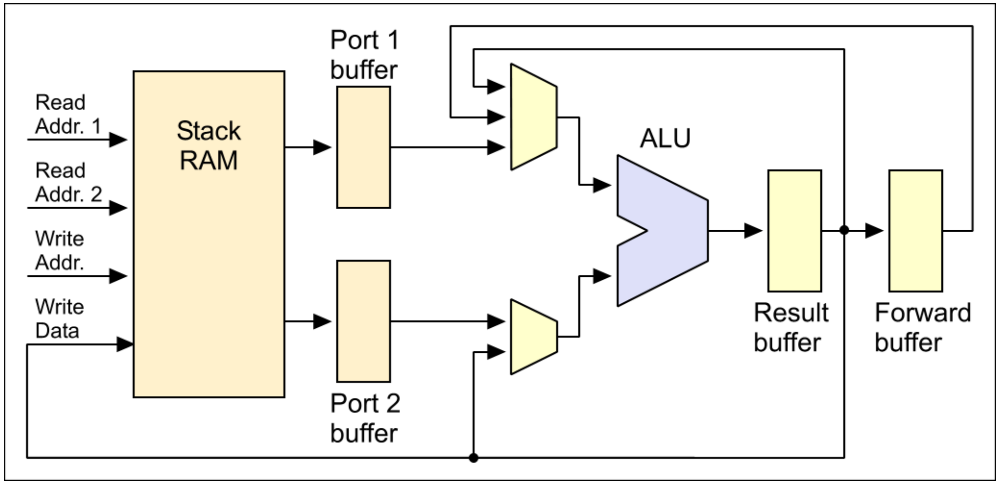
Register File Stack Cache



- Register file as circular buffer - small
- Automatic spill/fill
- Five access ports
- picoJava, aJile

- Instruction fetch
- Instruction decode
- RF read and execute
- RF write back

On-chip Memory Stack Cache



- *Large* cache
- Three-port memory
- Additional pipeline stage
- Komodo, FemtoJava

- Instruction fetch
- Instruction decode
- Memory read
- Execute
- Memory write back

JVM Stack Access Revised

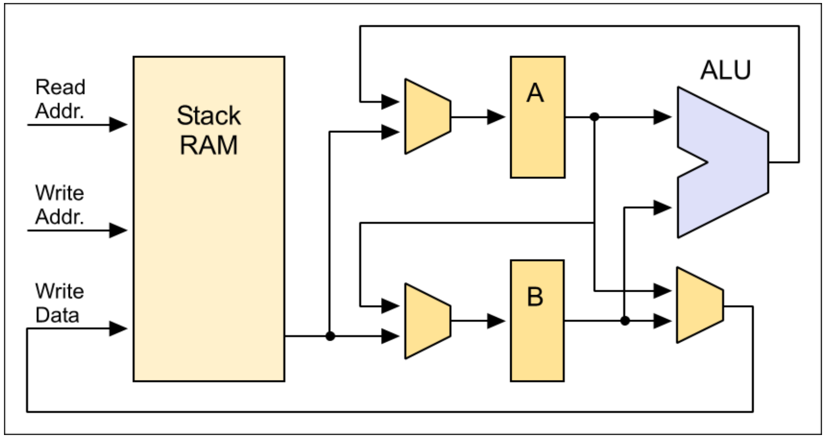
- ALU operation
  A <- A op B
  B <- sm[p]
  p <- p -1
- Variable load (*Push*)
  A <- sm[v+n]
  B <- A
  sm[p+1] <- B
  p <- p +1
- Variable store (*Pop*)
  sm[v+n] <- A
  A <- B
  B <- sm[p]
  p <- p -1

- *A* is TOS
- *B* is TOS-1
- *sm* is stack array
- *p* points to TOS-2
- *v* points to local area
- *n* is the local offset
- *op* is a two operand stack operation

Do we need a 3-port memory?

- Stack operation:
  - Dual read from TOS and TOS-1
  - Write to TOS
- Variable load/store:
  - One read port
  - One write port
- TOS and TOS-1 as register
- Deeper locations as on-chip memory

Two-Level Stack Cache



- Dual read only from TOS and TOS-1
- Two register (A/B)
- Dual-port memory
- Simpler Pipeline
- No forwarding logic

- Instruction fetch
- Instruction decode
- Execute, load or store

Stack Caches Compared

Design	Cache		f_{max}	Size
	(LC)	(bit)	(MHz)	(word)
ALU	-	-	237	-
16 register	707	0	110	16
RAM	111	8192	153	128
Two-level	112	4096	213	130

Summary

- The JVM is a stack machine
- Stack and local variables need caching
- Two-level cache
  - Two top levels as register
  - Rest as on-chip memory (two ports)
  - Small design
  - Short pipeline

Further Information

- JOP Thesis: p 78-93
- Martin Schoeberl, Design and Implementation of an Efficient Stack Machine, In *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop, RAW 2005*, Denver, Colorado, USA, April 2005.