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

```java
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 operand stack.
- `iconst_2` // operand stack.
- `invokevirtual #2` // Invoke method foo:(II)I.
- `istore_1` // Store the result in val.

```java
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

<table>
<thead>
<tr>
<th>SP</th>
<th>Operand stack</th>
<th>Context of Caller</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...</td>
<td>var_3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>var_2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>var_1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>var_0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VP</th>
<th>Operand stack</th>
<th>Context of Caller</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...</td>
<td>var_2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>var_1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>var_0</td>
</tr>
</tbody>
</table>

Current frame

Old frame

- arg_2
- arg_1
- arg_0
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

---

<table>
<thead>
<tr>
<th>A = B + C * D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
</tr>
<tr>
<td>push B</td>
</tr>
<tr>
<td>push C</td>
</tr>
<tr>
<td>push D</td>
</tr>
<tr>
<td>*</td>
</tr>
<tr>
<td>+</td>
</tr>
<tr>
<td>pop A</td>
</tr>
</tbody>
</table>
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
J VM Stack Access Revised

- **ALU operation**
  
  \[
  A \leftarrow A \text{ op } B \\
  B \leftarrow \text{sm}[p] \\
  p \leftarrow p - 1
  \]

- **Variable load (Push)**
  
  \[
  A \leftarrow \text{sm}[v+n] \\
  B \leftarrow A \\
  \text{sm}[p+1] \leftarrow B \\
  p \leftarrow p + 1
  \]

- **Variable store (Pop)**
  
  \[
  \text{sm}[v+n] \leftarrow A \\
  A \leftarrow B \\
  B \leftarrow \text{sm}[p] \\
  p \leftarrow p - 1
  \]

- \(A\) is TOS
- \(B\) is TOS-1
- \(\text{sm}\) is stack array
- \(p\) points to TOS-2
- \(v\) points to local area
- \(n\) is the local offset
- \(\text{op}\) 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

<table>
<thead>
<tr>
<th>Design</th>
<th>Cache</th>
<th>$f_{\text{max}}$</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(LC)</td>
<td>(bit)</td>
<td>(MHz)</td>
</tr>
<tr>
<td>ALU</td>
<td>-</td>
<td>-</td>
<td>237</td>
</tr>
<tr>
<td>16 register</td>
<td>707</td>
<td>0</td>
<td>110</td>
</tr>
<tr>
<td>RAM</td>
<td>111</td>
<td>8192</td>
<td>153</td>
</tr>
<tr>
<td>Two-level</td>
<td>112</td>
<td>4096</td>
<td>213</td>
</tr>
</tbody>
</table>
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