Context Threading: A flexible and efficient dispatch technique for virtual machine interpreters

Marc Berndl
Benjamin Vitale
Mathew Zaleski
Angela Demke Brown
Interpreter performance

• Why not just JIT?
  • High performance JITs still interpret
  • People use interpreted languages that don’t yet have JITs
  • They still want performance!

• 30-40% of execution time is due to branch misprediction
• Our technique eliminates 95% of branch mispredictions
Overview

✓ Motivation
  • Background: The Context Problem
  • Existing Solutions
  • Our Approach
  • Inlining
  • Results
A Tale of Two Machines

Virtual Machine Interpreter

Virtual Program

Loaded Program

Execution Cycle

Bytecode Bodies

Real Machine

CPU

Execution Cycle

Pipeline

Predictors

Target Address (Indirect)

Return Address

Wayness (Conditional)
Interpreter

Execution Cycle

- **fetch**
- **dispatch**
- **execute**
- **Load Parms**

**Loaded Program**

**Internal Representation**

**Bytecode bodies**
Running Java Example

Java Source

```java
void foo(){
    int i=1;
    do{
        i+=i;
    } while(i<64);
}
```

Java Bytecode

```
0:  iconst_0
1:  istore_1
2:  iload_1
3:  iload_1
4:  iadd
5:  istore_1
6:  iload_1
7:  bipush 64
9:  if_icmplt 2
12: return
```
Switched Interpreter

...  
 iload_1  
 iload_1  
iadd  
istore_1  
iload_1  
bipush 64  
if_icmplt 2  
...

\[
\begin{array}{c}
\text{vPC} \\
<i\text{load}_1> \\
<i\text{load}_1> \\
<i\text{add}> \\
<i\text{store}_1> \\
<i\text{load}_1> \\
<i\text{load}_1> \\
<i\text{bipush}> \\
64 \\
<i\text{if\_icmplt}> \\
-7 \\
\end{array}
\]

while(1) {
switch(*vPC++) {
    case iload_1:
        ..
        break;
    case iadd:
        ..
        break;
}
};

Virtual Program  
Internal Representation  
Switched Body Implementation

Simple, portable and extremely slow
Direct Threaded Interpreter

...  
**vPC**

<table>
<thead>
<tr>
<th>&amp;&amp;iload_1</th>
<th>&amp;&amp;iadd</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;istore_1</td>
<td>&amp;&amp;iadd</td>
</tr>
<tr>
<td>&amp;&amp;iadd</td>
<td>&amp;&amp;bipush</td>
</tr>
<tr>
<td>&amp;&amp;bipush</td>
<td>64</td>
</tr>
<tr>
<td>&amp;&amp;if_icmplt</td>
<td>-7</td>
</tr>
</tbody>
</table>

Virtual Program

DTT - Direct Threading Table

☞ Target of computed goto is data-driven
### Context Problem

#### DTT - Direct Threading Table

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&amp;iload_1</code></td>
</tr>
<tr>
<td><code>&amp;iload_1</code></td>
</tr>
<tr>
<td><code>&amp;iadd</code></td>
</tr>
<tr>
<td><code>&amp;istore_1</code></td>
</tr>
<tr>
<td><code>&amp;iload_1</code></td>
</tr>
<tr>
<td><code>&amp;bipush</code></td>
</tr>
<tr>
<td>64</td>
</tr>
<tr>
<td><code>&amp;if_icmplt</code></td>
</tr>
<tr>
<td>-7</td>
</tr>
</tbody>
</table>

#### Indirect Branch Predictors

- `&bipush`
- `iadd`
- `bipush`
- `vPC`
- `iload_1`

```
&iload_1:
  ..
goto *vPC++;
```
Existing Solutions

Replicate

1. `iload 1
   GOTO *PC
   1
   2`

2. `iload 1
   GOTO *PC
   1
   2`

Ertl & Gregg:
Bodies and Dispatch
Replicated

Super Instruction

| Body |
| Body |
| Body |
| Body |

| Body |

| GOTO *PC |

| ???? |

Piumarta & Ricardi:
Bodies Replicated

⚠️ Limited to relocatable virtual instructions
Overview

✓ Motivation
✓ Background: The Context Problem
✓ Existing Solutions
  • Our Approach
  • Inlining
  • Results
Key Observation

• Virtual and native control flow have same branch types
  • Linear (not really a branch)
  • Conditional
  • Calls and Returns
  • Indirect

• Hardware has predictors for each type

_solution: Leverage hardware predictors_


Essence of our Solution

... iload_1 iload_1 iadd istore_1 iload_1 bipush 64 if_icmplt 2 ...

CTT - Context Threading Table (generated code)

- call iload_1
- call iload_1
- call iadd
- call istore_1
- call iload_1

Bytecode bodies (ret terminated)

- iload_1:
  - ...
  - ret;

- iadd:
  - ...
  - ret;

Return Branch Predictor Stack

☞ Package bodies as subroutines and call them
Context Threading

... iload_1 iload_1 iadd istore_1 iload_1 bipush 64 if_icmplt 2 ...

DTT contains addresses in CTT

CTT load time generated code

Generate calls into the CTT at load time

-7

64

vPC

Bytecode bodies (ret terminated)

-iload_1:
  ...
  ret;

-iadd:
  ...
  ret;

-if_cmplt:
  ...
  goto *vPC++;

Call bipush

Call iload_1

Call iload_1

Call iload_1

Call istore_1

Call iadd

Call iload_1

Call if_icmplt

Call iload_1
The Context Threading Table

- A sequence of calls
- A sequence of generated instructions
- An internal representation of the program’s control flow
  - Virtual branches are also control flow

Can virtual branches go into the CTT?
Virtual Branches

Context problem is worse for virtual branches
Specialized Branch Inlining

Inlining conditional branches provides context
Tiny Inlining

- Context Threading is a dispatch technique
  - But, we inline branches
- Some non-branching bodies are very small
  - Why not inline those?

Inline all tiny linear bodies into the CTT
What can go in the CTT?

- Calls to bodies
- Inlined bodies
- Mixed-Mode virtual machine?
- Partially Inlined bodies
- Calls to subroutines that aren’t bytecodes
- Generated code

Performance?
Overview

✓ Motivation
✓ Background: The Context Problem
✓ Existing Solutions
✓ Our Approach
✓ Inlining
  • Results
Experimental Setup

- Two Virtual Machines on two hardware architectures.
  - VM: Java/SableVM, OCaml interpreter
  - Arch: P4, PPC
- Branch Misprediction
- Execution Time

Is our technique effective and general?
Mispredicted Taken Branches

95% mispredictions eliminated on average
**Execution time**

27% average reduction in execution time
Our technique is effective and general.
Conclusions

• Context Problem: branch mispredictions due to mismatch between native and virtual control flow

• Solution: Generate control flow code into the Context Threading Table

• Results
  • Eliminate 95% of branch mispredictions
  • Reduce execution time by 30-40%