Mixed Mode Execution with Context Threading

Mathew Zaleski, Marc Berndl, Angela Demke Brown
University of Toronto

{matz,berndl,demke}@cs.toronto.edu

(CASCON 2005, Oct 19/2005.)
Overview

- Introduction
  - Background:
    - Interpretation
    - Traces
  - Our Approach
  - Selecting Regions
  - Results and Conclusion
VM performance

- Native code performs better than an interpreter.
- Ahead-of-time compilation not always possible.
- High-performance VMs interpret and compile.
  - Hence term *mixed-mode* execution.
  - Typically method-based.

- Perl, python, php, Tcl, JavaScript and many others do *not* run mixed-mode. Why?
VM complexity

- Much up-front effort needed before method-based JIT works
- JIT must be able to compile complete inlined method nests before performance benefit accrues.
  - We aim to create a more incremental approach to building a mixed-mode system.
Vision of virtual machine lifecycle

- Begin with a high performance interpreter.
- Deploy modest system that compiles slow, hot, simple regions.
- Incrementally increase the size and generality of the regions.
  - Requires flexible region shape.
  - Should be able to fall back on interpretation
    - To reduce complexity of compiler.
Incremental VM lifecycle

- Context
- Threaded interpreter
- Basic Blocks
- Traces
- Partial methods?
- Optimized inlined method nests

Complexity of Compiled Code Regions

- Step up to more ambitious regions as required
Overview

✓ Introduction
  ▸ Background:
    • Interpretation
    • Traces
    • Our Approach
    • Selecting Regions
    • Results and Conclusion
Where does bytecode come from?

Java Source

```java
int f(boolean parm){
  if (parm){
    return 42;
  } else{
    return 0;
  }
}
```

Java Bytecode

```java
int f(boolean);  
Code:  
0: iload_1  
1: ifeq  7  
4: bipush 42  
6: ireturn  
7: iconst_0  
8: ireturn
```

Javac compiler
Interpreter

Execution Cycle

- **Load Program**
- **Internal Representation**
- **Bytecode bodies**
- **Execution Cycle**
  - **fetch**
  - **dispatch**
  - **execute**
  - **LoadParms**
Switched Interpreter

```c
while(1){
    switch(*vPC++){
    case iload_1:
        ..
        break;
    case ifeq:
        ..
        break;
    //and many more..
    }
}
```

» slow. burdened by switch and loop overhead
int f(boolean);

Code:
0: iload_1
1: ifeq 7
4: bipush 42
6: ireturn
7: iconst_0
8: ireturn

evaluation of virtual program “threads” through bodies
(as in needle & thread)

No switch overhead. Still nasty indirect branch.
Direct Threaded Interpreter

... vPC

**Virtual Program**

**DTT - Direct Threading Table**

- `iload_1`
- `ifeq 7`
- `bipush 42`
- `ireturn`
- `iconst_0`
- `ireturn`

- `& & iload_1`
- `& & ifeq`
- `& & 4`
- `& & bipush`
- `& & 42`
- `& & ireturn`
- `& & iconst_0`
- `& & ireturn`

**Target of computed goto is data-driven**

`iload_1:`

```
... goto *vPC++;
```

`ifeq:`

```
if () vPC= goto *vPC++;
```

`bipush:`

```
... goto *vPC++;
```

- `C implementation of each body`
We recently reported (CGO 2005) that on modern hardware (Pentium 4 and Power PC) dispatching virtual instruction bodies by calling them reduces branch mispredictions significantly.

Package bodies as subroutines and call them
Generating specialized code in CTT

Specialized bodies can also be generated in CTT!

Wayness Prediction also mobilized

Branch Inlined Into the CTT

- if(eq)
- goto target:
- call bipush
- call ...
- ...
- ...
- target:
HP Dynamo

- Trace-oriented dynamic optimization system.
  - HP PA-8000 computers.
- Counter-Intuitive approach:
  - Don’t execute optimized binary -- interpret it.
  - Count transits of each reverse branch.
  - Trace-generate straightened code.
  - Dispatch traces when encountered.
- Soon, essentially all execution from cache.
  - faster than binary!
Trace with if-then-else

```c
// c => b2
if (c)
  b1;
else
  b2;
b3;
```

- Trace contains path followed by conditional branches taken by program.
- Conditional branches turn into assertions, or trace exits.
- Expect trace exits to be not taken.
Other Related work

- Ertl & Gregg
- Piumarta & Riccardi
- Vitale & Abdelrahman
- Bala, Duesterwald and Banerjia
- Whaley
- Many JIT authors
Overview

✓ Introduction
✓ Background: Interpretation & traces
  ▸ Our Approach
    • Why Context Threading?
    • Case study: Forward Branch.
    • Selecting Regions
    • Results and Conclusion
Technical Focus of this work

- Much research exists on dynamically generating high performance code from Java bytecode.
- Relatively little investigation of how to JIT variously shaped regions of a running program.
  - Region selection;
  - Dispatch and execution;
  - Code generation.
  
  ▶ We concentrate on how to extend a CT interpreter to detect, translate and execute basic blocks and traces.
Context Threading was easy to program

Three main reasons:

1. Bodies organized as callable routines.
2. The DTT always points to implementation.
3. CTT callsites provide a convenient interposition opportunity.
1. Bodies are callable

Packaging bytecode bodies as lightweight subroutines

- Easy to intersperse generated code and dispatch.

```
call iload_1
call iload_1
specialized code
for iadd
call istore_1

iload_1:
  ...
  ret;

istore:
  ...
  ret;
```
2. DTT always points to implementation

..of corresponding region of virtual program

```
//branches to iadd
pc goto *pc

DTT
```

<table>
<thead>
<tr>
<th>DTT</th>
<th>CTT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>call iload_1</td>
</tr>
<tr>
<td></td>
<td>call iload_1</td>
</tr>
<tr>
<td></td>
<td>specialised code</td>
</tr>
<tr>
<td></td>
<td>for iadd</td>
</tr>
<tr>
<td></td>
<td>call istore_1</td>
</tr>
</tbody>
</table>

```
\begin{align*}
iload_1: & \\
& \ldots \\
& \text{ret;}
\end{align*}
```

```
\begin{align*}
\text{istore:} & \\
& \ldots \\
& \text{ret;}
\end{align*}
```

DTT/CTT correspondence enables *soft link* to dispatch code or body for a virtual instruction.
3. CTT provides for efficient interposition

An *Interposer* is a generated trampoline

```
DTT

iload_1

preworker()
  //instrument
  //or debug

postworker()
  //instrument
  //or debug
```

- Regular C functions called between every dispatch
CT as basis for light-weight JIT

- Code can be generated:
  - Inline in CTT.
  - As new dynamically generated callable region.

- Interposers support profiling:
  - Discover interesting regions at runtime.

- Rewrite DTT to “soft link” new code into program.
Overview

✓ Introduction
✓ Background: Interpretation & traces
  • Our Approach
    ✓ Why Context Threading?
    ‣ Case study: Forward Branch.
  • Selecting Regions
  • Results and Conclusion
Case Study: Forward Branch

Java Source

```java
int f(boolean parm){
    if (parm){
        return 42;
    } else{
        return 0;
    }
}
```

Java Bytecode

```java
int f(boolean);
Code:
0: iload_1
1: ifeq 7
4: bipush 42
6: ireturn
7: iconst_0
8: ireturn
```

- Address of destination needed to load branch
Loading Forward Branches

CTT

call iload_1
jmp 10
call bipush
42
call ireturn

10:
call iconst_0
call ireturn

iload_1:

ifeq: ..pc=.. ret;

call ifeq
call lazy

lazy()
   //rewrite ctt
   //to relative
}

Runtime -- lazily rewrite code as relative branch
Overview

✓ Introduction
✓ Background: Interpretation & traces
✓ Our Approach
  ▸ Selecting Regions
    • Basic Blocks
    • Traces
  • Results and Conclusion
Detecting basic blocks

- Detect basic blocks lazily.
- A virtual branch always ends a basic block.
  - Detect in post-worker of virtual branch.
- A basic block is always the destination of a virtual branch.
  - Detect in pre-worker called before every non-branching virtual instruction.
Detecting basic blocks

```
ifeq:
  call ifeq
  call post
  jmp *pc

post()
{
  curr_bb = 0;
}

iconst_0:

pre()
{
  if (!curr_bb)
  {
    curr_bb = new_bb()
  }
  append_bb();
}

ret
```
Generated code for a basic block

- Basic block is a run-time superinSTRUCTION
Overview

✓ Introduction
✓ Background: Interpretation & traces
✓ Our Approach
  • Selecting Regions
    ✓ Basic Blocks
    ‣ Traces
  • Results and Conclusion
Detecting Traces

- Use Dynamo’s SPECL trace detection heuristic.
- Instrument reverse branches until they are hot.
  - in postworker of virtual branch.
- Then trace generate
  - in preworker of each basic block region
Traces

A Trace is a run-time super-super-instruction
The code generation as we have described it today is preliminary.

We are actively working on a JIT that compiles basic blocks and traces into register allocated native code.

Meanwhile, what can we learn from the current system?
Overview

✓ Introduction
✓ Background: Interpretation & traces
✓ Our Approach
✓ Selecting Regions
  ▪ Results and Conclusion
Run Time performance

- We built our system into two VMs (Pentium 4).
  - Sablevm 1.1.8
  - Ocaml 3.08
- Region selection overhead is reasonable.

<table>
<thead>
<tr>
<th>VM</th>
<th>Benchmark Suite</th>
<th>Elapsed time to run whole suite</th>
<th>Direct Threaded (sec)</th>
<th>CT-trace (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sablevm</td>
<td>SpecJvm98</td>
<td>843</td>
<td>771</td>
<td></td>
</tr>
<tr>
<td>Ocaml</td>
<td>shootout</td>
<td>4.04</td>
<td>4.57</td>
<td></td>
</tr>
</tbody>
</table>
Incremental VM lifecycle

Complexity of Compiled Code Regions

- Ready for a better code generator..
Discussion

- Our system detects and executes basic blocks and traces.
- Paper discusses other shapes.
- Preliminary code generator shows:
  - Flexible shapes are doable.
  - Overheads are reasonable.
- Interesting to see how a better code generator affects performance.
Java histogram

Benchmark and Region Shape

% loaded code

javac/methods 48.9

0.99 0.95 0.9