YETI
Gradually Extensible Trace Interpreter

Mathew Zaleski

supervised by Professor Angela Demke Brown
co-supervised by Professor Michael Stumm

committee members:
Professor David Wortman
Professor Tarek Abdelrahman
Mr Kevin Stoodley
Many important languages do not deploy a JIT
Many important languages do not deploy a JIT
Many important languages do not deploy a JIT
Virtual Machine Design Space

- Many important languages do not deploy a JIT
Impact on users

Many important languages do not perform as well as they could if they deployed a JIT

- Emulation: interpreters
- Direct
- Threaded
- Selective Inlining
- Performance
- Hotspot, J9
- Mixed-mode: JIT compilers
Impact on developers

- “Big Bang” of method-based JIT build is risky

- Direct
- Threaded
- Selective
- Inlining

- Emulation: interpreters

- Mixed-mode: JIT compilers

- Hotspot, J9
Outline

• Motivation and Problem
  ▶ Interpretation
• Method-based JIT compilation
• Our Approach
• Contribution
• Measuring Yeti
• Future Work
Java Source

```java
int f() {
    c = a + b + 1
}
```

Virtual Program

```java
int f(boolean);
Code:
    iload a
    iload b
    iconst 1
    iadd
    iadd
    istore c
```

Run portably by High Level Language Virtual Machine
**Challenges of Interpretation**

<table>
<thead>
<tr>
<th>vPC</th>
<th>Virtual Instruction bodies</th>
</tr>
</thead>
</table>
| iload a  
  iload b  
  iconst 1  
  iadd  
  iadd  
  istore c | iload:  
  iconst:  
  iadd:  
  istore:  |

- **Virtual instruction body** emulates instruction at vPC.
- Often cases in C switch statement.
- **Dispatch** transfers control from body to body.
- Historically, issue was path length of dispatch code. Today, challenge is branch prediction.
Regular JIT compiles entire methods

Hot Method

```java
int f(boolean);

Code:
  iload a
  iload b
  iconst 1
  iadd
  iadd
  istore c
```

- Compile every virtual instruction - whole language
Regular JIT compiles entire methods

Hot Method

```java
int f(boolean);
Code:
    iload a
    iload b
    iconst 1
    iadd
    iadd
    istore c
```

Native code

```
01010101110101
11010101110100
10101010111011
00010101110100
111010101110111
01010101110101
11010101110100
10101010111011
00010101110100
111010101110111
```

- Compile every virtual instruction - whole language
Methods may contain cold code

Hot method

```java
fhot(){
    if(c){
        new Hot();
        h.hot();
    }else{
        new Cold();
        c.cold();
    }
}
```

JIT compiled code

```
01010101110101
11010101110100
10101010111011
00010101110100
111010101110111
01010101110101
11010101110100
10101010111011
00010101110100
111010101110111
```

- Cold portions of hot methods complicate runtime
Methods may contain cold code

Hot method

```java
fhot()
{
  if(c){
    new Hot();
    h.hot();
  }else{
    new Cold();
    c.cold();
  }
}
```

JIT compiled code

```
01010101110101
11010101110100
10101010111011
00010101110100
```

- Cold portions of hot methods complicate runtime
Methods may contain cold code

Hot method

```java
fhot(){
    if(c){
        new Hot();
        h.hot();
    }else{
        new Cold();
        c.cold();
    }
}
```

JIT compiled code

```
01010101110101
11010101110100
10101010111011
00010101110100
```

 resolve Cold

- Cold portions of hot methods complicate runtime
Methods may contain cold code

Hot method

```java
fhot()
if(c){
    new Hot();
    h.hot();
}else{
    new Cold();
    c.cold();
}
```

JIT compiled code

- 01010101110101
- 11010101110100
- 10101010111011
- 00010101110100

- resolve Cold
- invoke c.cold

- Cold portions of hot methods complicate runtime
Methods may contain cold code

**Hot method**

```java
fhot(){
    if(c){
        new Hot();
        h.hot();
    }else{
        new Cold();
        c.cold();
    }
}
```

**JIT compiled code**

```
01010101110101
11010101110100
10101010111011
00010101110100
```

`resolve Cold`, `invoke c.cold`, `BINARY_ADD b,c`

- Cold portions of hot methods complicate runtime
Outline

✓ Motivation and Problem
  • Our Approach
    ▶ Callable bodies
  • Subroutine threaded interpretation
  • Trace-based JIT compilation

• Contributions
• Measuring Yeti
• Future Work
Callable bodies

• Suppose virtual instruction bodies are callable.
  • Then JIT compiler would have the option of compiling some virtual instructions and fall back on calling the bodies for others.
  • This could smooth part of the “big bang”
• Only viable if there is an efficient way to build a simple interpreter also.
Subroutine Threading

Straight-line Virtual code

Sequence of direct call instructions generated when method is loaded

- iload a
- iload b
- iconst 1
- iadd
- iadd
- istore c

- Straight-line code efficiently dispatched due to return branch predictor stack in modern processor
Subroutine Threading

Straight-line Virtual code

- iload a
- iload b
- iconst 1
- iadd
- iadd
- istore c

Sequence of direct call instructions generated when method is loaded

- call iload
- call iload
- call iload
- call iload
- call iload
- call iload

- call iadd
- call iadd
- call iadd
- call iadd
- call istore
- call istore

- Straight-line code efficiently dispatched due to return branch predictor stack in modern processor
Subroutine Threading

Straight-line Virtual code

Sequence of direct call instructions generated when method is loaded

- iload a
- iload b
- iconst 1
- iadd
- iadd
- call iload
- call iload
- call iload
- call iconst
- call iadd
- call iadd

- Straight-line code efficiently dispatched due to return branch predictor stack in modern processor
Subroutine Threading

Straight-line Virtual code

- iload a
- iload b
- icont 1
- iadd
- iadd
- ifeq xx

Sequence of direct call instructions generated when method is loaded

- call iload
- call iload
- call icont
- call iadd
- call iadd
- ??

- Straight-line code efficiently dispatched due to return branch predictor stack in modern processor
Synopsis of our approach

- Region bodies called from same dispatch loop as virtual instruction bodies

```c
VM
interp()
{
  while(1)
  {
    (*vPC)();
  }
};
```

Direct Call Threaded Dispatch Loop

- Region bodies called from same dispatch loop as virtual instruction bodies
Synopsis of our approach

Region bodies called from same dispatch loop as virtual instruction bodies

```c
interp()
{
    while(1){
        profile(vPC);
        (*vPC)();
    }
}
```
Synopsis of our approach

- Region bodies called from same dispatch loop as virtual instruction bodies

```javascript
interp()
    while(1)
        profile(vPC);
        (*vPC)();
    }
```

```javascript
fhot()
    flg = x || y
    if(flg)
        new Hot();
        h.hot();
    else
        new Cold();
        c.cold();
    }
```
Synopsis of our approach

- Region bodies called from same dispatch loop as virtual instruction bodies
Synopsis of our approach

- **Region bodies** called from same dispatch loop as virtual instruction bodies.
Traces easy to compile

```java
fhot()
{
    if (flg) {
        new Hot();
        h.hot();
    } else {
        new Cold();
        c.cold();
    }
}
```

- Traces contain no merge points or cold code
Traces easy to compile

Suppose flg is usually true

```java
fhot()
{
    if (flg)
    {
        new Hot();
        h.hot();
    }
    else
    {
        new Cold();
        c.cold();
    }
}
```

- Traces contain no merge points or cold code
Traces easy to compile

```java
fhot(){
    if(fl) {
        new Hot();
        h.hot();
    } else {
        new Cold();
        c.cold();
    }
}
```

- Traces contain no merge points or cold code
Traces easy to compile

```java
fhot()
{
    if(flg)
    {
        new Hot();
        h.hot();
    }
    else
    {
        new Cold();
        c.cold();
    }
}
```

Traces contain no merge points or cold code
Traces easy to compile

```java
fhot(){
    if(flg){
        new Hot();
        h.hot();
    }else{
        new Cold();
        c.cold();
    }
}
```

Traces contain no merge points or cold code
Interpreted traces

Interpreted traces run virtual branches efficiently because trace exits predict each destination

<table>
<thead>
<tr>
<th>generated code</th>
</tr>
</thead>
<tbody>
<tr>
<td>call</td>
</tr>
<tr>
<td>call</td>
</tr>
<tr>
<td>call ifne</td>
</tr>
<tr>
<td>cmpl vPC,d</td>
</tr>
<tr>
<td>jne TEH_c</td>
</tr>
<tr>
<td>call</td>
</tr>
<tr>
<td>call</td>
</tr>
<tr>
<td>call</td>
</tr>
<tr>
<td>call</td>
</tr>
</tbody>
</table>
### Interpreted traces

Interpreted traces run virtual branches efficiently because trace exits predict each destination.

#### Generated Code

```
@call
@call
@call
@call_ifne
@cmpl vPC, d
@jne TEH_c
@call
@call
@call
```

> “Interpreted” - all real work done in bodies
Interpreted traces run virtual branches efficiently because trace exits predict each destination.

Call virtual branch body which sets the vPC to destination.

"Interpreted" - all real work done in bodies.

- `call`
- `call`
- `call ifne`
- `cmpl vPC,d`
- `jne TEH_c`
- `call`
- `call`
- `call`
- `call`
Interpreted traces run virtual branches efficiently because trace exits predict each destination.
Interpreted traces run virtual branches efficiently because trace exits predict each destination.
Build out more virtual instructions

Hot virtual code

```java
int f(boolean);
Code:
iload a
iload b
iconst 1
iadd
iadd
istore c
```

Translated code

- By tightly integrating JIT and interpreter we can gradually build out our interpreter to be a JIT
Build out more virtual instructions

Hot virtual code

int f(boolean);
Code:
  iload a
  iload b
  iconst 1
  iadd
  iadd
  istore c

Translated code

iconst 1 compiled to native code
mov $1,(%vsp)
inc %vsp

- By tightly integrating JIT and interpreter we can gradually build out our interpreter to be a JIT
By tightly integrating JIT and interpreter we can gradually build out our interpreter to be a JIT
Yeti design trajectory

- Our infrastructure supports broad range of performance
Outline

✓ Motivation and Problem
✓ Our Approach
  ▶ Contribution
• Measuring Yeti
• Future Work
Overview of Contribution

We show:

- DCT dispatch loop
- callable bodies
- interpreted trace
- trace as unit of compilation
Overview of Contribution

We show:
1. Callable bodies dispatch straight-line code efficiently.

<table>
<thead>
<tr>
<th>YETI</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DCT dispatch loop</td>
<td>callable bodies</td>
</tr>
<tr>
<td>interpreted trace</td>
<td>trace as unit of compilation</td>
</tr>
</tbody>
</table>
Overview of Contribution

We show:

1. Callable bodies dispatch straight-line code efficiently.
2. Traces capture much of execution, eliminate trips around dispatch loop

<table>
<thead>
<tr>
<th>YETI</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCT dispatch loop</td>
</tr>
<tr>
<td>callable bodies</td>
</tr>
<tr>
<td>interpreted trace</td>
</tr>
<tr>
<td>trace as unit of compilation</td>
</tr>
</tbody>
</table>
Overview of Contribution

<table>
<thead>
<tr>
<th>YETI</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCT dispatch loop</td>
</tr>
<tr>
<td>callable bodies</td>
</tr>
<tr>
<td>interpreted trace</td>
</tr>
<tr>
<td>trace as unit of compilation</td>
</tr>
</tbody>
</table>

We show:

1. Callable bodies dispatch straight-line code efficiently.

2. Traces capture much of execution, eliminate trips around dispatch loop.

3. Interpreted traces improve virtual branch performance.
Overview of Contribution

We show:

1. Callable bodies dispatch straight-line code efficiently.

2. Traces capture much of execution, eliminate trips around dispatch loop

3. Interpreted traces improve virtual branch performance.

4. Trace based JIT simple - smooths “big bang”.

YETI

<table>
<thead>
<tr>
<th>DCT dispatch loop</th>
<th>callable bodies</th>
</tr>
</thead>
<tbody>
<tr>
<td>interpreted trace</td>
<td>trace as unit of compilation</td>
</tr>
</tbody>
</table>
Experiments

- Built subroutine threaded versions of SableVM and OCaml for PPC and Pentium4. (CGO 2005)
Experiments

• Built subroutine threaded versions of SableVM and OCaml for PPC and Pentium4. (CGO 2005)
• Built version of JamVM for PPC970 with interpreted traces and trace JIT. (VEE 07)
Experiments

• Built subroutine threaded versions of SableVM and OCaml for PPC and Pentium4. (CGO 2005)
• Built version of JamVM for PPC970 with interpreted traces and trace JIT. (VEE 07)
• Performance Evaluation:
  • Compare elapsed time to direct threading.
Experiments

- Built subroutine threaded versions of SableVM and OCaml for PPC and Pentium4. (CGO 2005)
- Built version of JamVM for PPC970 with interpreted traces and trace JIT. (VEE 07)
- Performance Evaluation:
  - Compare elapsed time to direct threading.
  - Benchmarks suite is the SPECjvm98 + scimark.
1. Efficient dispatch of callable bodies

Elapsed time performance of subroutine threading relative to direct threading (Pentium4)

SPECjvm98 + scitest sorted by LB length
1. Efficient dispatch of callable bodies

Elapsed time performance of subroutine threading relative to direct threading (Pentium4)

SPECjvm98 + scitest sorted by LB length
1. Efficient dispatch of callable bodies

Elapsed time performance of subroutine threading relative to direct threading (Pentium4)

SPECjvm98 + scitest sorted by LB length
1. Efficient dispatch of callable bodies

- Straight-line code dispatched with few mispredictions
- Virtual branches not improved
- Similar on PPC970

Elapsed time performance of subroutine threading relative to direct threading (Pentium4)

SPECjvm98 + scittest sorted by LB length
2. Traces account for almost all execution

- **LINEAR BLOCK**
- **i-TRACE NO LINK**
- **INTERPRETED TRACE**

Virtual instructions executed per region dispatch:

- $10^5$
- $10^4$
- $10^3$
- $10^2$
- $10^1$

**SPECjvm98 + scitest**
2. Traces account for almost all execution

<table>
<thead>
<tr>
<th>LINEAR BLOCK</th>
<th>i-TRACE NO LINK</th>
<th>INTERPRETED TRACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>10^5</td>
<td>10^4</td>
<td>10^3</td>
</tr>
<tr>
<td>10^2</td>
<td>10^1</td>
<td></td>
</tr>
</tbody>
</table>

virtual instructions executed per region dispatch

geomean

SPECjvm98 + scitest
2. Traces account for almost all execution

- The dynamic average LB executes 6.3 virtual instructions between branches.

<table>
<thead>
<tr>
<th>virtual instructions executed per region dispatch</th>
<th>(10^5)</th>
<th>(10^4)</th>
<th>(10^3)</th>
<th>(10^2)</th>
<th>(10^1)</th>
</tr>
</thead>
</table>

geomean

SPECjvm98 + scitest
2. Traces account for almost all execution

- The dynamic average LB executes 6.3 virtual instructions between branches.
- 99.9% of virtual instruction executed from traces.
- Traces with linking disabled remain on trace for about 5 trace exits on average.
2. Traces account for almost all execution

- The dynamic average LB executes 6.3 virtual instructions between branches.
- 99.9% of virtual instruction executed from traces.
- Traces with linking disabled remain on trace for about 5 trace exits on average.
- Trace linking closes loop nests explaining strong effect.
3. Interpreted traces improve virtual branch performance.

- LINEAR BLOCKS
- INTERPRETED TRACE

**SPECjvm98 + scitest**
3. Interpreted traces improve virtual branch performance.

Elapsed Time Relative to Direct Threading

- LINEAR BLOCKS
- INTERPRETED TRACE

SPECjvm98 + scitest
3. Interpreted traces improve virtual branch performance.

- Linear blocks are runtime generated subroutine threaded code.
- Slower than SUB due to overhead
3. Interpreted traces improve virtual branch performance.

- Linear blocks are runtime generated subroutine threaded code.
- Slower than SUB due to overhead.
- Interpreted, linked traces outperform direct threading, selective inlining.
- 4% greater speedup than selective inlining.
- Because (previous slide) traces predict destination of about 5 branches, on average.
4. Trace-based JIT is easy to build

Reduces need for a “big bang” project compared to a method-based JIT:

- Support for 50 integer bytecodes requires only 1800 statements of C (;)
- 1100 LOC common, 700 LOC for PPC
- Development experience:
  - Easy to debug because can add support for one virtual instruction at a time
  - Easy to isolate bugs and sidestep corner cases
Outline

✓ Motivation and Problem
✓ Our Approach
✓ Contribution
  ▶ Measuring Yeti
• Future Work
Measuring Yeti
Measuring Yeti

• Suppose Direct Call Threading, Linear Blocks, Traces, etc were incremental releases of a VM.

• How would performance improve from release to release?
Measuring Yeti

- Suppose Direct Call Threading, Linear Blocks, Traces, etc were incremental releases of a VM.
- How would performance improve from release to release?
- We’ve already seen that interpreted traces perform well compared selective inlining, a high performance dispatch technique.
Elapsed Time Relative to Direct Threading

- DIRECT CALL THREADING
- LINEAR BLOCKS
- i-TRACE NO LINK
- INTERPRETED TRACE
- TR-JIT

SPECjvm98 + scistest
Elapsed Time Relative to Direct Threading

- DIRECT CALL THREADING
- LINEAR BLOCKS
- i-TRACE NO LINK
- INTERPRETED TRACE
- TR-JIT

geomean

SPECjvm98 + scitest
Elapsed Time Relative to Direct Threading

- Direct Call Threading about as fast as switch (on PPC).
• Direct Call Threading about as fast as switch (on PPC).
• Simple trace JIT 32% faster
- Direct Call Threading about as fast as switch (on PPC).
- Simple trace JIT 32% faster
- Almost 2x direct threading.
Elapsed Time Relative to Direct Threading

- Direct Call Threading about as fast as switch (on PPC).
- Simple trace JIT 32% faster
- Almost 2x direct threading.
- NB: Hotspot still 4x faster.
Stall Cycles (JamVM scitest PPC970FX)
Stall Cycles (JamVM scitest PPC970FX)

• negligible stalls due to i-cache misses for scitest (no blue hatch on top)
Stall Cycles (JamVM scitest PPC970FX)

- negligible stalls due to i-cache misses for scitest (no blue hatch on top)
- mispredicted conditional branches
Stall Cycles (JamVM scitest PPC970FX)

- negligible stalls due to i-cache misses for scitest (no blue hatch on top)
- mispredicted conditional branches
- "other stalls" include lr/ctr stalls caused by indirect branches
Stall Cycles (JamVM scitest PPC970FX)

- negligible stalls due to i-cache misses for scitest (no blue hatch on top)
- mispredicted conditional branches
- “other stalls” include lr/ctr stalls caused by indirect branches
- completed instructions
scitest (float long blocks) vs SUB

Subroutine Threading

- DISTRO
- SUB

Cycles relative to jam-distro:
- DISTRO
- SUB

- i-cache
- br_misp
- fxu
- fpu
- d-cache
- basic_lsu
- compl

Zaleski Departmental Oct 16/07
scitest (float long blocks) vs SUB

Subroutine Threading

Better prediction
scitest (float long blocks) vs SUB

Subroutine Threading

Better prediction

some i-cache

<table>
<thead>
<tr>
<th>Cycles Relative to Jam-Distro</th>
<th>DISTRO</th>
<th>SUB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- i-cache
- br_misp
- fxu
- fpu
- d-cache
- basic_lsu
- compl
scitest (float long blocks) vs SUB

Subroutine Threading

Better prediction

some i-cache

fewer lr/ctr stalls
scitest (float long blocks) vs SUB

Subroutine Threading

- Better prediction
- Some i-cache
- Fewer lr/ctr stalls
- More stalls on real work (FPU)
scitest (float long blocks) vs SUB

Subroutine Threading
- Better prediction
- Some i-cache
- Fewer lr/ctr stalls
- More stalls on real work (FPU)
- Longer path length (calls & returns)

Zaleski Departmental Oct 16/07
scitest (float long blocks) vs TR-JIT
scitest (float long blocks) vs TR-JIT

- No i-cache stalls for TR-JIT
scitest (float long blocks) vs TR-JIT

- No i-cache stalls for TR-JIT
- Even fewer lr/ctr stalls (traces improve virtual branches)
scitest (float long blocks) vs TR-JIT

- No i-cache stalls for TR-JIT
- Even fewer lr/ctr stalls (traces improve virtual branches)
- JIT improves path length

Graph showing cycles relative to jam-distro for DISTRO, SUB, and TR-JIT.
Javac (int, trace cache bloat)

- i-cache stall
- traces contain 4x virtual instructions of loaded methods
• Our approach offers breadth of deployable milestones.
  ▸ A more gradual approach to building a mixed-mode system.
- Our approach offers breadth of deployable milestones.
  - A more gradual approach to building a mixed-mode system.
Weak aspects of our prototype

- `inline asm("ret")` obscures flowgraph of interpreter from `gcc` optimizer.
- No classical optimizations performed on traces
  - Probably would require industry project to learn ultimate performance potential relative to existing method-based JITs.
✓ Motivation and Problem
✓ Our Approach
✓ Contribution
✓ Measuring Yeti
  ▶ Future Work
Future Work

1. Apply dynamic compilation to runtime typed languages (e.g. Python).
   - Use trace exits to guard speculatively optimized regions of code.
   - Speculatively specialize runtime typed virtual instructions (e.g. Python’s `BINARY_ADD`).

2. Investigate a new shape of compilation unit
   - Built from network of linked traces.

3. Investigate trace-cache bloat observed in javac.

4. Work to improve performance of bodies implemented as `gcc` nested functions.
End
BACK
Interp background

• too detailed for departmental?
Direct Call Threaded Interpreter

- Body also can be called from code generated by JIT

```c
interp(t_vpc *vPC){
    vPC = rep;
    while(1)
        (*vPC)();

    iload:
        //push local *vPC++
    vPC++;  
    asm ("ret");  //x86
    iconst:
    iadd:
    istore:
```

### Register Stack

<table>
<thead>
<tr>
<th>rep</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;iload</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>&amp;iload</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>&amp;iconst</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>&amp;iadd</td>
</tr>
<tr>
<td>&amp;iadd</td>
</tr>
<tr>
<td>&amp;istore</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>..</td>
</tr>
</tbody>
</table>
Direct Call Threaded Interpreter

```c
interp(t_vpc *vPC){
    vPC = rep;
    while(1)
        (*vPC)();

    iload:
        //push local *vPC++
        vPC++;
    iconst:
    iadd:
    istore:
```

- Body also can be called from code generated by JIT

Zaleski Departmental Oct 16/07

39
Direct Call Threaded Interpreter

```c
interp(t_vpc *vPC){
    vPC = rep;
    while(1)
        (*vPC)();
    iload:
        //push local *vPC++
        vPC++;
    iadd:
    iconst:
    istore:
}
```

Body also can be called from code generated by JIT
Body also can be called from code generated by JIT
Direct Call Threaded Interpreter

 interp(t_vpc *vPC){
  vPC = rep;
  while(1)
    (*vPC)();

  iload:
    //push local *vPC++
    vPC++; 
    asm ("ret"); //x86

  iconst:
  iadd:
  istore:

Body also can be called from code generated by JIT
Direct Call Threaded Interpreter

```
interp(t_vpc *vPC) {
  vPC = rep;
  while(1) {
    (*vPC)();
  }
  iload:
    //push local *vPC++
    vPC++;
  iadd:
  iconst:
  istore:
}
```

- Body also can be called from code generated by JIT

Zaleski Departmental Oct 16/07
Traces are interprocedural paths through program.
Dynamo Traces

Traces are interprocedural paths through program
Traces are interprocedural paths through program

```c
for(;;){
  if (c){
    b1;
  } else {
    b2;
  }
  b3;
}
```
Traces are interprocedural paths through program
Traces are interprocedural paths through program.

\[ \text{for(;;){} if (c){ b1; } else { b2; } b3; } \]
Dynamo Traces

```
for(;;){
  if (c){
    b1;
  } else {
    b2;
  }
  b3;
}
```

Traces are interprocedural paths through program
Essence of Subroutine Threading

Context Threading Table

<table>
<thead>
<tr>
<th>DTT</th>
<th>CTT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>call iload_1</td>
</tr>
<tr>
<td>4</td>
<td>call ifeq</td>
</tr>
<tr>
<td>42</td>
<td>call bipush</td>
</tr>
<tr>
<td></td>
<td>call ireturn</td>
</tr>
<tr>
<td></td>
<td>call iconst_0</td>
</tr>
<tr>
<td></td>
<td>call ireturn</td>
</tr>
</tbody>
</table>

- iload_1: ...
  asm(ret);

- ifeq: vPC=...
  goto *vPC;

- Package bodies as subroutines and call them
Context Threading Table

```
<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 ifeq</td>
</tr>
<tr>
<td></td>
<td>call bipush</td>
</tr>
<tr>
<td>4</td>
<td>call ireturn</td>
</tr>
<tr>
<td>42</td>
<td>call icontst_0</td>
</tr>
<tr>
<td></td>
<td>call ireturn</td>
</tr>
</tbody>
</table>
```

```
iload_1:
  ..
  asm(ret);
ifeq:
  vPC=..
  goto *vPC;
```

- Package bodies as subroutines and call them
Essence of Subroutine Threading

Package bodies as subroutines and call them

Context Threading Table

DTT
- 
- 
4
- 
42
- 

CTT
- call iload_1
- call ifeq
- call bipush
- call ireturn
- call iconst_0
- call ireturn

iloading:
  ...
  asm(ret);

ifeq:
  vPC=..
  goto *vPC;

ret terminated

generated code
Package bodies as subroutines and call them
Essence of Subroutine Threading

Package bodies as subroutines and call them.

**Context Threading Table**

<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 ifeq</td>
</tr>
<tr>
<td></td>
<td>call bipush</td>
</tr>
<tr>
<td></td>
<td>call ireturn</td>
</tr>
<tr>
<td></td>
<td>call iconst_0</td>
</tr>
<tr>
<td></td>
<td>call ireturn</td>
</tr>
</tbody>
</table>

- **iload_1**: ..
  asm(ret);
- **ifeq**: vPC=..
  goto *vPC;
- ret terminated
- virtual branches as direct
- points to generated code
- generated code

Zaleski Departmental Oct 16/07
Context Threading (CT) -- Generating specialized code in CTT

Specialized bodies can also be generated in CTT!

call iload_1
subl
movl ;pop stack
cmpl ;compare 0
jne
movl ; vPC =
jmp ; else
addl ; vPC +=
call bipush
call ireturn
call iconst_0
call ireturn
Context Threading (CT) -- Generating specialized code in CTT

Specialized bodies can also be generated in CTT!

```
call iload_1
subl
movl ;pop stack
cmpl ;compare 0
jne
movl ; vPC =
jmp ; else
addl ; vPC +=
call bipush
call ireturn
call iconst_0
call ireturn
```

inlined code for if_eq
Context Threading (CT) -- Generating specialized code in CTT

Specialized bodies can also be generated in CTT!

```
call iload_1
subl
movl ;pop stack
cmpl ;compare 0
jne
movl ; vPC =
jmp ; else
addl ; vPC +=
call bipush
call ireturn
call iconst_0
call ireturn
```
OUTLINE

• Introduction
• Implementation
  • Efficient Interpretation
  • YETI
    1. Linear Blocks
    2. Traces
    3. Simple Trace JIT
• Experimental Results.
Trace Compilation - 3 stage process

1. Dispatch instructions, identify *linear blocks* (LB)
   - LB is a sequence of virtual instructions, ending with branch.

2. Dispatch linear blocks, identify traces.
   - A trace is a sequence of linear blocks.

3. JIT compile hot traces.
   - Compile only selected virtual instructions.

- Prototype built on top of Lougher’s JamVM 1.3.3
1. Dispatch instructions, Identify Linear Blocks

```c
fhot()
{
    c = a + b + 1;
    if(c)
        new Hot();
        h.hot();
    else
        new Cold();
        c.cold();
}
```

When branch reached the history list contains LB

```c
interp()
{
    while(1)
    {
        pre_work(vPC);
        (*vPC)();
        post_work(vPC);
    }
}
```
1. Dispatch instructions, Identify Linear Blocks

```java
interp()
{
    while(1){
        pre_work(vPC);
        (*vPC)();
        post_work(vPC);
    }
}

fhot()
{
    c = a + b + 1;
    if(c){
        new Hot();
        h.hot();
    }else{
        new Cold();
        c.cold();
    }
}
```

- When branch reached the history list contains LB
1. Dispatch instructions, Identify Linear Blocks

When branch reached the history list contains LB
1. Dispatch instructions, Identify Linear Blocks

```
interp()
{
    while(1){
        pre_work(vPC);
        (*vPC)();
        post_work(vPC);
    }
}
```

```
fhot()
{
    c = a + b + 1;
    if(c){
        new Hot();
        h.hot();
    }else{
        new Cold();
        c.cold();
    }
}
```

- When branch reached the history list contains LB

```history_list
  iload
  iload
  iconst
  ... 
  goto
```
Use History List to generate LB

- New region body will run from now on
Use History List to generate LB

New region body will run from now on
Use History List to generate LB

New region body will run from now on
Execute LB

```plaintext
while(1){
    pre_work(vPC);
    (*vPC)();
    post_work(vPC);
}
```

- vPC set by region body

<table>
<thead>
<tr>
<th>rep</th>
<th>1b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>iload</td>
</tr>
<tr>
<td>b</td>
<td>iconst</td>
</tr>
<tr>
<td>1</td>
<td>iadd</td>
</tr>
<tr>
<td>iadd</td>
<td>istore</td>
</tr>
<tr>
<td>c</td>
<td>goto</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>generated code</th>
</tr>
</thead>
<tbody>
<tr>
<td>call iload</td>
</tr>
<tr>
<td>call iload</td>
</tr>
<tr>
<td>call iconst</td>
</tr>
<tr>
<td>call iadd</td>
</tr>
<tr>
<td>call istore</td>
</tr>
<tr>
<td>call goto</td>
</tr>
<tr>
<td>ret</td>
</tr>
</tbody>
</table>
Execute LB

```
while(1){
    pre_work(vPC);
    (*vPC)();
    post_work(vPC);
}
```

> vPC set by region body
Execute LB

<table>
<thead>
<tr>
<th>rep</th>
<th>lb</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>iload</td>
</tr>
<tr>
<td>b</td>
<td>icont</td>
</tr>
<tr>
<td>1</td>
<td>iadd</td>
</tr>
<tr>
<td>iadd</td>
<td>istore</td>
</tr>
<tr>
<td>c</td>
<td>goto</td>
</tr>
<tr>
<td>42</td>
<td></td>
</tr>
</tbody>
</table>

```
while(1) {
    pre_work(vPC);
    (*vPC)();
    post_work(vPC);
}
```

generated code:
```
call iload
call iload
call icont
call iadd
call iadd
call istore
call goto
ret
```

- vPC set by region body
Execute LB

```java
while(1){
    pre_work(vPC);
    (*vPC)();
    post_work(vPC);
}
```

- vPC set by region body
2. Run LB, identify traces

```c
//c mostly false
if(c){
    b1;
} else {
    b2;
}
b3;
```

```
LB

<table>
<thead>
<tr>
<th>c</th>
<th>call</th>
<th>call</th>
<th>ifeq</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>b2</th>
<th>b3</th>
<th>call</th>
<th>call</th>
<th>call</th>
</tr>
</thead>
</table>

history_list
```

- LB’s in trace recorded in history list
2. Run LB, identify traces

//c mostly false
if (c) {
    b1;
} else {
    b2;
}
b3;

LB

//
c mostly false
if(c){
    call
    call
    ifeq
}
}
}
}
b3;

b2|b3
    call
    call
    call

history_list

c

LB’s in trace recorded in history list
2. Run LB, identify traces

```java
// c mostly false
if (c) {
    b1;
} else {
    b2;
}
```

LB

```
<table>
<thead>
<tr>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>call</td>
</tr>
<tr>
<td>call</td>
</tr>
<tr>
<td>ifeq</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>b2</th>
<th>b3</th>
</tr>
</thead>
<tbody>
<tr>
<td>call</td>
<td></td>
</tr>
<tr>
<td>call</td>
<td></td>
</tr>
<tr>
<td>call</td>
<td></td>
</tr>
</tbody>
</table>
```

- LB’s in trace recorded in history list
Use history list to generate trace

Trace predicts path through virtual program
Use history list to generate trace

Trace predicts path through virtual program
Use history list to generate trace

Trace predicts path through virtual program
Use history list to generate trace

- Trace predicts path through virtual program
Use history list to generate trace

rep
ifeq
call
call
ifeq
b2|b3
call
call
call
texit..
generated code
call
call
call
trace exit
to b1
call
call
texit..region body for trace

Trace predicts path through virtual program
Details of an (Interpreted) Trace

Interpreted traces run on PPC, x86 (June).
Details of an (Interpreted) Trace

- Interpreted traces run on PPC, x86 (June).

```
  call
  call
  call ifne
  cmpl vPC,b2
  jne TEH_c
  call
  call
  call
  texit..
```

Generated code:

```
  TEH_C
  ...
  ...
  ret;
```

"Interpreted" - all real work done in bodies
Details of an (Interpreted) Trace

“Interpreted” - all real work done in bodies

Compare vPC to hardwired address of the on trace destination (b2|b3)

Interpreted traces run on PPC, x86 (June).

generated code

- call
- call
- call ifne
- cmpl vPC,b2
- jne TEH_c
- call
- call
- call
texit..

TEH_C

.. 
.. 
ret;
Details of an (Interpreted) Trace

- Interpreted traces run on PPC, x86 (June).

```
  generated code
  call
  call
  call ifne
  cmpl vPC,b2
  jne TEH_c
  call
  call
  texit..
```

- "Interpreted" - all real work done in bodies
- Compare vPC to hardwired address of the on trace destination (b2|b3)
- Trace Exit Handler (TEH) does housekeeping, then returns to dispatch loop
Details of an (Interpreted) Trace

- Interpreted traces run on PPC, x86 (June).

```
Generated code:
call
call
call ifne
cmpl vPC, b2
jne TEH_c
call
call
call
texit..

"Interpreted" - all real work done in bodies

Compare vPC to hardwired address of the on trace destination (b2|b3)

Trace Exit Handler (TEH) does housekeeping, then returns to dispatch loop

Rewrite as branch to link traces

TEH_C
..
..
ret;
```
3. JIT compiled Trace

- Traces are easy to compile (PPC only)

- generated code
  - `call`
  - `100101000`
  - `011001100`
  - `011001011`
  - `bne TEH_C`
  - `101010101`
  - `100101000`
  - `011001100`
  - `texit..`

```
TEH_C
...
...
ret;
```
3. JIT compiled Trace

Traces are easy to compile (PPC only)

Some virtual instructions still emulated

generated code

<table>
<thead>
<tr>
<th>call</th>
<th>100101000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>011001100</td>
</tr>
<tr>
<td>011001011</td>
<td></td>
</tr>
<tr>
<td>bne TEH_C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>101010101</td>
</tr>
<tr>
<td></td>
<td>100101000</td>
</tr>
<tr>
<td>011001100</td>
<td></td>
</tr>
<tr>
<td>texit..</td>
<td></td>
</tr>
</tbody>
</table>

TEH_C

.. 
.. 
ret;
3. JIT compiled Trace

- Traces are easy to compile (PPC only)

- Some virtual instructions still emulated

- No control flow merges.
- No forward branches.

```assembly
  generated code
  call 100101000
  011001100
  011001011
  bne TEH_C 101010101
  00101000
  011001100
  texit..

  TEH_C
  ..
  ..
  ret;
```
3. JIT compiled Trace

- Traces are easy to compile (PPC only)

<table>
<thead>
<tr>
<th>generated code</th>
<th>Trace Exit Handler (TEH) flushes any values in registers to stack.</th>
</tr>
</thead>
<tbody>
<tr>
<td>call</td>
<td>100101000</td>
</tr>
<tr>
<td>011001100</td>
<td></td>
</tr>
<tr>
<td>011001011</td>
<td></td>
</tr>
<tr>
<td>bne TEH_C</td>
<td>101010101</td>
</tr>
<tr>
<td>100101000</td>
<td>texit..</td>
</tr>
<tr>
<td>011001100</td>
<td></td>
</tr>
</tbody>
</table>

- Some virtual instructions still emulated

- No control flow merges.
- No forward branches.

```c
ret;
```