Software Method Level Speculation for Java

Christopher John Francis Pickett
chris.pickett@mail.mcgill.ca

Sable Research Group
School of Computer Science
McGill University

February 17th, 2012

Ph.D. Oral Defense
Parallelization
thesis
Parallelization

automatic

thesis
Parallelization
Parallelization

- Software
- Automatic
- Thesis
- Speculation
Parallelization

speculation

automatic

thesis

software

method level
Method Level Speculation

Before:
- **P**
  - pre-invocation instructions
  - method body
  - method continuation
  - invoke
  - return

After:
- **P**
  - invoke
  - return
  - fork
  - join
  - C1
Initial System

- initial system
Initial System

dependence buffering

initial system
Initial System

- initial system
  - dependence buffering
  - return value prediction
Initial System

- initial system
  - dependence buffering
  - method level speculation
  - return value prediction
Initial System

- fork heuristics
- method level speculation
- dependence buffering
- return value prediction
Initial System

VM design
fork heuristics
Java language considerations
method level speculation
dependence buffering
return value prediction
// execute foo() non-speculatively
r = foo(x, y, z);
Example

// execute foo() non-speculatively
r = foo(x, y, z);

// execute foo()'s continuation speculatively?
call stack growth
A()
{
    a;
    B()
    {
        b;
    }
    a';
}
Single Child Speculation

```c
A()
{
    a;
    B()
    {
        b;
    }
    a';
}
```

![Diagram of call stack growth showing stack frames for A, B, and C1 with variables a, b, and a']
Out-of-Order Nesting

A()
 a;
 B()
 b;
 C()
 c;
}  
b';
}  
a';
}
Out-of-Order Nesting

```c
A() {
    a;
    B() {
        b;
        C() {
            c;
        }
        b';
    }
    a';
}
```
Out-of-Order Nesting
Out-of-Order Nesting

A()
{
    a;
    B()
    {
        b;
        C()
        {
            c;
        }
    }
    b';
}

a';
Example

// execute foo() non-speculatively
r = foo(x, y, z);

// execute foo()'s continuation speculatively
if (r > 10)  // predict return value
{
    ...
}

<table>
<thead>
<tr>
<th>predictor</th>
<th>history</th>
<th>prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>last value</td>
<td>1, 1, 1</td>
<td>1</td>
</tr>
<tr>
<td>predictor</td>
<td>history</td>
<td>prediction</td>
</tr>
<tr>
<td>---------------</td>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td>last value</td>
<td>1, 1, 1</td>
<td>1</td>
</tr>
<tr>
<td>stride</td>
<td>1, 2, 3</td>
<td>4</td>
</tr>
</tbody>
</table>
## Return Value Prediction

<table>
<thead>
<tr>
<th>predictor</th>
<th>history</th>
<th>prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>last value</td>
<td>1, 1, 1</td>
<td>1</td>
</tr>
<tr>
<td>stride</td>
<td>1, 2, 3</td>
<td>4</td>
</tr>
<tr>
<td>context</td>
<td>1, 5, 6, 8, ... , 1, 5, 6</td>
<td>8</td>
</tr>
</tbody>
</table>
## Return Value Prediction

<table>
<thead>
<tr>
<th>predictor</th>
<th>history</th>
<th>prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>last value</td>
<td>1, 1, 1</td>
<td>1</td>
</tr>
<tr>
<td>stride</td>
<td>1, 2, 3</td>
<td>4</td>
</tr>
<tr>
<td>context</td>
<td>1, 5, 6, 8, \ldots, 1, 5, 6</td>
<td>8</td>
</tr>
<tr>
<td>memoization</td>
<td>( f(2, 4) : 7, \ldots, f(2, 4) )</td>
<td>7</td>
</tr>
</tbody>
</table>
Software Hybrid RVP

callsites  naïve hybrid predictors

f(); →
  .
  .
  . 10%

f(); →
  . 70%

f(); →
  . 40% 80%

g(); →
  . 40% 80%
// execute foo() non-speculatively
r = foo (x, y, z);

// execute foo()’s continuation speculatively
if (r > 10) // predict return value
    {
        s = o1.f; // buffer heap & static reads
        o2.f = r; // buffer heap & static writes
    }
// execute foo() non-speculatively
r = foo(x, y, z);

// execute foo()'s continuation speculatively
if (r > 10) // predict return value
{
    s = o1.f; // buffer heap & static reads
    o2.f = r; // buffer heap & static writes
}

// invoke bar() speculatively
o3.bar();
Speculative Method Invocation

```c
A()
{
    a;
    B()
    {
        b;
        C()
        {
            c;
        }
        D()
        {
            d;
        }
    }
    a';
}

b';
```

Diagram:
- Call stack growth
- Stack frames:
  - c
  - b
  - a
  - P
  - C1
Speculative Method Invocation

A()
  a;
  B()
    b;
    C()
      c;
  }
  b';
  D()
    d;
  }
  a';
}

C1
  d
  b'
  b
  c
P
Speculative Method Invocation

```
A()
{
    a;
    B()
    {
        b;
        C()
        {
            c;
        }
    }
    b';
    D()
    {
        d;
    }
    a';
}
```
Speculative Method Invocation

```
A()
{
    a;
    B()
    {
        b;
        C()
        {
            c;
        }
    }
    b';
    D()
    {
        d;
    }
}
}
```

Call stack growth:

- a
- b
- c
- P
- a'
- C1
// execute foo() non-speculatively
r = foo(x, y, z);

// execute foo()'s continuation speculatively
if (r > 10) // predict return value
{
    s = o1.f; // buffer heap & static reads
    o2.f = r; // buffer heap & static writes
}

// invoke bar() speculatively
o3.bar();

// stop speculation due to unsafe operation
synchronized (o4) { ... }

Profiling

VM design
fork heuristics
Java language considerations
method level speculation
dependence buffering
return value prediction

initial system

profiling
Profiling

- VM design
- fork heuristics
- Java language considerations
- dependence buffering
- method level speculation
- return value prediction

profile

high RVP overhead
Profiling

- VM design
- fork heuristics
- Java language considerations
- dependence buffering
- method level speculation
- return value prediction

profiling

- high RVP overhead
- idle processors
Profiling

VM design
fork heuristics
Java language considerations
method level speculation
dependence buffering
return value prediction

profiling

high RVP overhead
idle processors
short threads
Optimization

- high RVP overhead
- idle processors
- short threads

adaptive RVP
Adaptive RVP

callsites naïve hybrid predictors

f();
  .
  10%

100%

80%

g();

70%

40%
Adaptive RVP

callsites  naïve hybrid predictors

\[ f(); \quad 10\% \quad 80\% \]
\[ g(); \quad 70\% \quad 40\% \]

specialization

callsites  specialized hybrid predictors

\[ f(); \quad 80\% \]
\[ g(); \quad 80\% \quad 70\% \]

despecialization
Optimization

- high RVP overhead
- idle processors
- short threads

- adaptive RVP
- in-order nesting
A() {
    a;
    B() {
        b;
    }
    a';
    C() {
        c;
    }
    a'';
}

Call stack growth
In-Order Nesting

A()
{
    a;
    B()
    {
        b;
    }
    a';
    C()
    {
        c;
    }
    a'';
}

call stack growth

a

p
In-Order Nesting

```c
A()
{
  a;
  B()
  {
    b;
  }
  a';
  C()
  {
    c;
  }
  a'';
}
```

![Call stack growth diagram](image)
In-Order Nesting

```c
A()
{
    a;
    B()
    {
        b;
    }
    a';
    C()
    {
        c;
    }
    a'';
}
```

Call stack growth:

- `A()`: `a`, `b`, `a'`, `C()`, `a''`
- `B()`: `b`
- `C()`: `c`

Call stack:

- `P`: `a`, `b`
- `C1`: `a'`, `c`
- `C2`: `a''`
A()
    a;
B()
    b;
    C()
        c;
    }
    b';
    D()
        d;
    }
    b'';
}
    a';
}
A()
{
    a;
    B()
    {
        b;
        C()
        {
            c;
        }
        b';
        D()
        {
            d;
        }
        b'';
    }
    a';
}

 call stack growth

a
P
A()
{
    a;
    B()
    {
        b;
        C()
        {
            c;
        }
        b';
        D()
        {
            d;
        }
        b'';
    }
    a';
}
Mixed Nesting

```c
A()
{
    a;
    B()
    {
        b;
        C()
        {
            c;
        }
        b';
        D()
        {
            d;
        }
        b'';
    }
    a';
}
```
Mixed Nesting

A()
{
    a;
    B()
    {
        b;
        C()
        {
            c;
        }
    }
    b';
    D()
    {
        d;
    }
    b'';
}

\[ a'; \]

\[ \text{call stack growth} \]

\[ P \quad \rightarrow \quad C2 \quad \rightarrow \quad C3 \quad \rightarrow \quad C1 \]
Optimization

- high RVP overhead
  - adaptive RVP
- idle processors
  - in-order nesting
- short threads
  - structural heuristics
tail (i, n) {
    work (i);
    if (i < n)
        tail (i + 1, n);
}

head (i, n) {
    if (i < n)
        head (i + 1, n);
    work (i);
}

iterate (n) {
    for (i = 1; i <= n; i++)
        work (i);
}

tree (node *n) {
    if (!leaf (n)) {
        tree (n->left);
        tree (n->right);
    }
    work (n);
}
// in-order nesting

tail (i, n) {
    spec work (i);
    if (i < n)
        tail (i + 1, n);
}

// out-of-order nesting

head (i, n) {
    if (i < n)
        spec head (i + 1, n);
    work (i);
}

// in-order nesting

iterate (n) {
    for (i = 1; i <= n; i++)
        spec work (i);
}

// mixed nesting

tree (node *n) {
    if (!leaf (n)) {
        spec tree (n->left);
        spec tree (n->right);
    }
    work (n);
}
Future Work

- high RVP overhead
- idle processors
- short threads

-adaptive RVP
-in-order nesting
-structural heuristics

future work
Future Work

- high RVP overhead
  - adaptive RVP
- idle processors
  - in-order nesting
- short threads
  - structural heuristics

future work

- manual speculation
- static analysis
Future Work

- high RVP overhead
- idle processors
- short threads

- adaptive RVP
- in-order nesting
- structural heuristics

- future work
  - manual speculation
  - static analysis
  - JIT codegen
we can build initial system software MLS
Conclusions

feedback loops cause problems

we can build software MLS

initial system
Conclusions

feedback loops cause problems

we can build software MLS

unique software solutions arise

initial system
Conclusions

feedback loops cause problems

we can build software MLS

initial system

speculation for Java works out

unique software solutions arise
Conclusions

feedback loops cause problems
we can build software MLS
unique software solutions arise

initial system

semantics are non-obvious
speculation for Java works out
Conclusions

- Feedback loops cause problems
- We can build software MLS
- Unique software solutions arise
- Initial system
  - Semantics are non-obvious
  - Speculation for Java works out
- Profiling
Conclusions

- Feedback loops cause problems
- We can build software MLS
- Unique software solutions arise
- Initial system
- Semantics are non-obvious
- Speculation for Java works out
- Profiling
- Lots of parallelism
Conclusions

- Feedback loops cause problems
- We can build software MLS
- Unique software solutions arise
- Initial system
- Profiling
- Overheads are a problem
- Semantics are non-obvious
- Speculation for Java works out

Lots of parallelism
Conclusions

- Feedback loops cause problems
- We can build software MLS
- Unique software solutions arise
- Lots of parallelism
- Initial system
- Speculation for Java works out
- Misspeculation costs are masked
- Overheads are a problem
Conclusions

- Feedback loops cause problems
- We can build software MLS
- Unique software solutions arise
- Initial system
- Speculation for Java works out
- Misspeculation costs are masked
- Overheads are a problem
- Lots of parallelism
- Profiling
- Adaptive RVP
Conclusions

- Feedback loops cause problems
- We can build software MLS
- Unique software solutions arise
- Initial system
  - Speculation for Java works out
  - Misspeculation costs are masked
  - Overheads are a problem
- Profiling
  - Lots of parallelism
- Adaptive RVP
  - Ideal per-callsite predictors exist
- Semantics are non-obvious
Conclusions

- Feedback loops cause problems
- We can build software MLS
- Unique software solutions arise
- Initial system
  - Speculation for Java works out
  - Misspeculation costs are masked
  - Overheads are a problem
- Profiling
  - Lots of parallelism
- Adaptive RVP
  - Ideal per-callsite predictors exist
- In-order nesting
Conclusions

- Feedback loops cause problems.
- We can build software MLS.
  - Unique software solutions arise.
  - Initial system.
  - Speculation for Java works out.
  - Misspeculation costs are masked.
  - Overheads are a problem.
- Profiling.
  - Lots of parallelism.
  - Adaptive RVP.
    - Ideal per-callsite predictors exist.
  - In-order nesting.
    - Mixed nesting maximizes parallelism.
- Semantics are non-obvious.
Conclusions

Feedback loops cause problems.

We can build software MLS.

Unique software solutions arise.

Speculation for Java works out.

Semantics are non-obvious.

Misspeculation costs are masked.

Overheads are a problem.

Lots of parallelism.

Profiling

Adaptive RVP

Ideal per-callsite predictors exist.

In-order nesting

Mixed nesting maximizes parallelism.

Structural heuristics