Optimized Return Value Prediction for Java

Christopher J.F. Pickett          Clark Verbrugge
{cpicke, clump}@sable.mcgill.ca

School of Computer Science, McGill University
Montréal, Québec, Canada H3A 2A7
Overview

- Introduction and Related Work
- Contributions
- Predictor Design
- Accuracy
- Memory Usage
- Compiler Analysis Framework
- Return Value Use Analysis
- Parameter Dependence Analysis
- Combined Analyses
- Conclusions and Future Work
Introduction and Related Work

Speculative method-level parallelism (SMLP) allows for dynamic parallelisation of single-threaded programs
- speculative threads are forked at callsites
- suitable for Java virtual machines

Perfect return value prediction can double performance of SMLP (Hu et al., 2003)

Goals
- Implement Hu’s predictors in SableVM
- Achieve higher accuracy
- Use Soot to optimize performance
Speculative Method-Level Parallelism

// execute foo non-speculatively
r = foo (a, b, c);

// execute past return point
// speculatively in parallel with foo()
if (r > 10)
{
    s = o1.f; // buffer heap reads
    o2.f = r; // buffer heap writes
}
...

Impact of Return Value Prediction

<table>
<thead>
<tr>
<th>RVP strategy</th>
<th>return value</th>
<th>SMLP speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>arbitrary</td>
<td>1.52</td>
</tr>
<tr>
<td>best</td>
<td>predicted</td>
<td>1.92</td>
</tr>
<tr>
<td>perfect</td>
<td>correct</td>
<td>2.76</td>
</tr>
</tbody>
</table>

- 26% speedup over no RVP with Hu’s best predictor
- 82% speedup over no RVP with perfect prediction
  - Improved hybrid accuracy is highly desirable

Contributions (1)

- Implement existing predictors in JVM
  - Various simple constant space predictors
  - Table-based finite context method (FCM) predictor
  - Hybrid predictor
- New memoization predictor
  - Table-based, like context predictor
  - Hashes together method arguments
  - Performs well in a hybrid
Contributions (2)

- Explore predictor performance limits
  - Allocate storage until accuracy no longer improves
  - Reduce memory requirements
    - Dynamically expand hashtables
    - Exploit VM info about value widths
- Perform compiler analyses to help prediction
  - Return value use analysis
    - Allows for use of incorrect predictions
  - Parameter dependence analysis
    - Eliminates unnecessary memoization inputs
Design

- Implement all predictors in software JVM
  - not trace-based
  - not simulated
- Variable space table-based predictors:
  - Context – inputs are return value history
  - Memoization – inputs are method parameters
  - Hybrid predictor – best sub-predictor over last 32 values
- Attach context and memoization tables per callsite
- Expand tables dynamically, up to a fixed maximum
# Memoization

<table>
<thead>
<tr>
<th>hash(a,b,c)</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo(7,5,3)</td>
<td>11</td>
</tr>
<tr>
<td>foo(4,6,8)</td>
<td>9</td>
</tr>
<tr>
<td>foo(9,1,2)</td>
<td>10</td>
</tr>
</tbody>
</table>
Context and Memoization Predictors

(a) Context Predictor

(b) Memoization Predictor
Size Variation

![Graph showing the relationship between maximum per-callsite table size (bits) and accuracy (%). The graph indicates an upward trend, with accuracy increasing as the maximum per-callsite table size increases.]
Size Variation

Accuracy (%) vs. Maximum Per-Callsite Table Size (bits)

- Context
- Memoization
- Hybrid
mtrt Context Size Variation

The graph illustrates the accuracy (%) of mtrt-up and mtrt-mp as a function of the maximum per-callsite table size (bits). The accuracy increases with increasing table size, reaching a plateau at higher values. The accuracy for mtrt-up is consistently higher than mtrt-mp across the range of table sizes shown.
mtrt Memoization Size Variation

accuracy (%)

maximum per-callsite table size (bits)

mtrt_up
mtrt_mp
Hybrid Size Variation

Accuracy (%) vs. Maximum per-callsite table size (bits)

- mtrt_up
- mtrt_mp
\{jack, javac, jess\} Context

The diagram shows the relationship between the accuracy (%) and the maximum per-callsite table size (bits) for three different Java compilers: jack, javac, and jess. The accuracy percentage increases as the maximum per-callsite table size increases for all three compilers. However, the accuracy for jess is the highest followed by javac and then jack. The relationship is approximately linear, indicating that increasing the table size leads to a proportional increase in accuracy.
\{jack, javac, jess\} Memoization
Context Size Variation (all)

accuracy (%)

graph showing the relationship between maximum per-callsite table size (bits) and accuracy (%) for various programs and libraries, including comp, db, jack, javac, jess, mpeg, mtrt_up, and mtrt_mp.
Memoization Size Variation (all)

The graph shows the accuracy (%) on the y-axis as a function of the maximum per-callsite table size (bits) on the x-axis. Different tools and applications are represented with distinct markers and lines, including:

- comp
- db
- jack
- javac
- jess
- mpeg
- mtrt_up
- mtrt_mp

The accuracy increases as the maximum per-callsite table size increases, varying across different tools and applications.
Hybrid Size Variation (all)
Memory Usage

- Most hashtables are small.
- Most variability is confined to a few callsites.
- We can exploit VM level value width info to conserve memory; a naïve approach would require 64 bits per table value.
  - Exploiting type information yields a 35% space reduction.
- Memoization consistently requires less space than context-based prediction. This indicates suitability for hardware designs.
### Table Memory

<table>
<thead>
<tr>
<th>benchmark</th>
<th>context</th>
<th>memoization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>size</td>
<td>original</td>
</tr>
<tr>
<td>comp</td>
<td>24</td>
<td>313M</td>
</tr>
<tr>
<td>db</td>
<td>24</td>
<td>541M</td>
</tr>
<tr>
<td>jack</td>
<td>14</td>
<td>15.8M</td>
</tr>
<tr>
<td>javac</td>
<td>20</td>
<td>291M</td>
</tr>
<tr>
<td>jess</td>
<td>14</td>
<td>13.5M</td>
</tr>
<tr>
<td>mpeg</td>
<td>12</td>
<td>3.72M</td>
</tr>
<tr>
<td>mtrt</td>
<td>14</td>
<td>69.4M</td>
</tr>
<tr>
<td><strong>average</strong></td>
<td><strong>17</strong></td>
<td><strong>178M</strong></td>
</tr>
</tbody>
</table>

Accurate return value prediction is expensive! We now look to static analysis to address this.
Analysis Framework

- Soot: Java bytecode compiler framework
  - Spark: points-to analysis and callgraph
  - Jimple: typed, stackless, 3-address IR
  - Attribute generation framework

- SableVM: portable Java virtual machine
  - Attribute parsing
  - RVP implementation

- SPEC Client JVM98 Benchmark Suite
  - S100 (size 100), no harness
  - All benchmarks except raytrace
Return Value Use Analysis

- An incorrect return value $r$ may be OK
  - If $r$ is unused
  - If $r$ appears only inside a boolean expression

```c
r = foo(a, b, c);
if (r > 10)
{
    ... // r == 11, 12, 13, ...
}
else
{
    ... // r == 10, 9, 8, ...
}
```
Return Value Use Analysis

- Collect *use expressions* for each return value
- Evaluate use expressions at runtime in SableVM
  - If predicted and actual return values satisfy use expressions identically, we can substitute an inaccurate prediction
  - ++accuracy
Return Value Use

S1: foo()

S2: q = bar()

S3: if (q < 5)

S4: b++

S5: c++

S6: r = baz()

S7: o1.f = r
Return Value Use

```
S1  foo()
S2  q = bar()
S3  if (q < 5)
S4  b++  c++
S5
S6  r = baz()
S7  o1.f = r
```

<table>
<thead>
<tr>
<th></th>
<th>consumed</th>
<th>accurate</th>
<th>uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Return Value Use

```
S1    foo()
S2    q = bar()
S3    if (q < 5)
S4    b++
S5    c++
S6    r = baz()
S7    o1.f = r
```

<table>
<thead>
<tr>
<th></th>
<th>consumed</th>
<th>accurate</th>
<th>uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>no</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

QCPLS-11-2004 – p.28/59
Return Value Use

- **S1**: `foo()`
  - **S2**: `q = bar()`
    - **S3**: `if (q < 5)`
      - **S4**: `b++`
      - **S5**: `c++`
      - **S6**: `r = baz()`
  - **S7**: `o1.f = r`

- **Table**:

<table>
<thead>
<tr>
<th></th>
<th>consumed</th>
<th>accurate</th>
<th>uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>no</td>
<td>no</td>
<td>______</td>
</tr>
<tr>
<td>S2</td>
<td>yes</td>
<td>no</td>
<td><code>q &lt; 5</code></td>
</tr>
<tr>
<td>S6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Return Value Use

<table>
<thead>
<tr>
<th></th>
<th>consumed</th>
<th>accurate</th>
<th>uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>no</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>yes</td>
<td>no</td>
<td>q &lt; 5</td>
</tr>
<tr>
<td>S6</td>
<td>yes</td>
<td>yes</td>
<td></td>
</tr>
</tbody>
</table>
## Static Return Value Use Results

<table>
<thead>
<tr>
<th>benchmark</th>
<th>non-void callsite return values</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total</td>
<td>unconsumed</td>
<td>inaccurate</td>
<td>accurate</td>
</tr>
<tr>
<td>comp</td>
<td>7156</td>
<td>23.1%</td>
<td>9.4%</td>
<td>67.5%</td>
</tr>
<tr>
<td>db</td>
<td>7322</td>
<td>22.7%</td>
<td>9.6%</td>
<td>67.7%</td>
</tr>
<tr>
<td>jack</td>
<td>8090</td>
<td>21.2%</td>
<td>10.0%</td>
<td>68.8%</td>
</tr>
<tr>
<td>javac</td>
<td>10503</td>
<td>17.3%</td>
<td>12.8%</td>
<td>69.9%</td>
</tr>
<tr>
<td>jess</td>
<td>9531</td>
<td>18.8%</td>
<td>9.4%</td>
<td>71.8%</td>
</tr>
<tr>
<td>mpeg</td>
<td>7586</td>
<td>22.4%</td>
<td>9.9%</td>
<td>67.7%</td>
</tr>
<tr>
<td>mtrt</td>
<td>8029</td>
<td>21.3%</td>
<td>8.7%</td>
<td>70.0%</td>
</tr>
<tr>
<td>average</td>
<td>8317</td>
<td>20.7%</td>
<td>10.1%</td>
<td>69.2%</td>
</tr>
</tbody>
</table>
## Dynamic Return Value Use Results

<table>
<thead>
<tr>
<th>benchmark</th>
<th>non-void callsite return values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total</td>
</tr>
<tr>
<td>comp</td>
<td>133M</td>
</tr>
<tr>
<td>db</td>
<td>115M</td>
</tr>
<tr>
<td>jack</td>
<td>34M</td>
</tr>
<tr>
<td>javac</td>
<td>82M</td>
</tr>
<tr>
<td>jess</td>
<td>102M</td>
</tr>
<tr>
<td>mpeg</td>
<td>77M</td>
</tr>
<tr>
<td>mtrt</td>
<td>267M</td>
</tr>
<tr>
<td>average</td>
<td>116M</td>
</tr>
</tbody>
</table>
Return Value Use Results

- Accuracy of db prediction increases by 5–7%.
- Currently, use expressions only involve $r$ and a constant.
  - In the future, we’ll allow for locals as well as constants.
- We use canonical values of $r$ to save on predictor memory.
- Overall memory savings are discussed later.
Parameter Dependence Analysis

Problem: redundant entries in memoization hashtables

<table>
<thead>
<tr>
<th>hash(a,b,c)</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo(7,5,3)</td>
<td>11</td>
</tr>
<tr>
<td>foo(7,2,3)</td>
<td>11</td>
</tr>
<tr>
<td>foo(7,8,3)</td>
<td>11</td>
</tr>
</tbody>
</table>
Parameter Dependence Analysis

Insight: not all parameters affect return value. If we can eliminate predictor inputs, we should be able to increase hashtable sharing.

- ++accuracy
- --size

<table>
<thead>
<tr>
<th>hash(a,c)</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo(7,5,3)</td>
<td>11</td>
</tr>
<tr>
<td>foo(7,2,3)</td>
<td></td>
</tr>
<tr>
<td>foo(7,8,3)</td>
<td></td>
</tr>
</tbody>
</table>

QCPLS-11-2004 – p.35/59
Intraprocedural Parameter Dependence

public int foo (int a, int b, int c)

int k, r

START

k = 0

if (a > 5)

print(b)    k = 7

k = k + 1

r = c + k

return r

END
Intraprocedural Parameter Dependence

public int foo (int a, int b, int c)

    int k, r

    START

    k = 0

    if (a > 5)

        print(b)    k = 7

        k = k + 1

        r = c + k

        return r

    END

{}
Intraprocedural Parameter Dependence

public int foo (int a, int b, int c)

int k, r

START

k = 0

if (a > 5)

print(b)  k = 7

k = k + 1

r = c + k

{return r}

{r}

END
Intraprocedural Parameter Dependence

public int foo (int a, int b, int c)

int k, r

START

k = 0

if (a > 5)

print(b)

k = 7

k = k + 1

{k, k}

r = c + k

{r}

return r

{ }

END
Intraprocedural Parameter Dependence

public int foo (int a, int b, int c)

int k, r

START

k = 0

if (a > 5)

print(b)       k = 7

{k, c}

k = k + 1

{k, c}

r = c + k

{r}

return r

{}
Intraprocedural Parameter Dependence

```java
public int foo (int a, int b, int c)

    int k, r

    START

    k = 0

    if (a > 5)

        {c, k}

        print(b)

        {c, k}

        k = 7

        k = k + 1

        {c, k}

        r = c + k

        {r}

        return r

        {}

    END
```
Intraprocedural Parameter Dependence

public int foo (int a, int b, int c)

int k, r

START

k = 0

if (a > 5)

{c, k}

print(b) k = 7

{c, k}

k = k + 1

{c, k}

r = c + k

{r}

return r

{}
public int foo (int a, int b, int c)

int k, r

START

k = 0

if (a > 5)

print(b)  k = 7

{k, k}  {c, k}  {c}

k = k + 1

r = c + k

{c, k}  {r}

return r

{r}

{}
Intraprocedural Parameter Dependence

public int foo (int a, int b, int c)

int k, r

START

k = 0

if (a > 5)

{c, k} print(b) {c}

{c, k} k = 7 {c, k}

k = k + 1 {c, k}

r = c + k {r}

return r {}
Intraprocedural Parameter Dependence

```java
public int foo (int a, int b, int c)

    int k, r

    START

    k = 0

    if (a > 5)

        {c, k}

        print(b)

        {c}

        k = 7

        {c, k}

        k = k + 1

        {c, k}

        r = c + k

        {r}

        return r

        {}

    END
```

branch dependent

branch independent
Intraprocedural Parameter Dependence

public int foo (int a, int b, int c)

    int k, r

    START

    k = 0

    if (a > 5)

    {c, k}  

    print(b)  

    {c}

    k = 7  

    {c, k}  

    writes to out(s)

    k = k + 1

    {c, k}  

    r = c + k

    {r}  

    return r

    {c, k}  

    END

branch dependent

branch independent
Intraprocedural Parameter Dependence

public int foo (int a, int b, int c)

int k, r

START

k = 0

if (a > 5)

print(b)  \[\{a, c, k}\]

k = 7  \[\{c\}\]

\{c, k\}

k = k + 1  \[\{c, k\}\]

r = c + k  \[\{r\}\]

return r  \{\}\n
END

writes to out(s)

branch dependent

branch independent
Intraprocedural Parameter Dependence

public int foo (int a, int b, int c)

int k, r

```
START
{a, c}

k = 0
{a, c, k}

if (a > 5)

{c, k}

print(b)

{c}

writes to out(s)

{k = 7}

{c, k}

k = k + 1
{c, k}

r = c + k
{r}

return r
{}

END

branch dependent

branch independent
```
Intraprocedural Parameter Dependence

```java
public int foo (int a, int b, int c)

    int k, r

    START
    {a, c} ∩ {this, a, b, c}

    k = 0
    {a, c, k}

    if (a > 5)
        {a, c, k}

        print(b)
        {c, k}

        k = 7
        {c}

        writes to out(s)

        {c, k}

        k = k + 1
        {c, k}

        {c, k}

        r = c + k
        {r}

        return r
        {r}

    END
```

- branch dependent
- branch independent
Intraprocedural Parameter Dependence

public int foo (int a, int b, int c)

int k, r

START

k = 0

{k, a, c} \cap \{this, a, b, c\} = \{a, c\}

if (a > 5)

{k, c, a, k}

print(b)

{k, c, k}

k = k + 1

{k, c, k}

r = c + k

{r}

return r

{}
Parameter Dependence

- Intraprocedural part: computes an optimistic slice, and relies on a simple branch dependence analysis.
- Interprocedural part: computes a least fixed point using a standard worklist algorithm.
  - \( \perp \) implies no parameters are used to determine the return value at a method callsite
## Static Parameter Dependence Results

<table>
<thead>
<tr>
<th>benchmark</th>
<th>consumed callsites with $n &gt; 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total</td>
</tr>
<tr>
<td>comp</td>
<td>5294</td>
</tr>
<tr>
<td>db</td>
<td>5446</td>
</tr>
<tr>
<td>jack</td>
<td>6159</td>
</tr>
<tr>
<td>javac</td>
<td>8460</td>
</tr>
<tr>
<td>jess</td>
<td>7476</td>
</tr>
<tr>
<td>mpeg</td>
<td>5671</td>
</tr>
<tr>
<td>mtrt</td>
<td>6100</td>
</tr>
<tr>
<td><strong>average</strong></td>
<td>6372</td>
</tr>
</tbody>
</table>
## Dynamic Parameter Dependence Results

<table>
<thead>
<tr>
<th>benchmark</th>
<th>consumed method calls with $n &gt; 0$</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total</td>
<td>$d = 0$</td>
<td>$d &gt; 0 \land d &lt; n$</td>
<td>$d = n$</td>
</tr>
<tr>
<td>comp</td>
<td>133M</td>
<td>0.0%</td>
<td>0.9%</td>
<td>99.1%</td>
</tr>
<tr>
<td>db</td>
<td>115M</td>
<td>3.8%</td>
<td>0.0%</td>
<td>96.2%</td>
</tr>
<tr>
<td>jack</td>
<td>33M</td>
<td>7.4%</td>
<td>1.9%</td>
<td>90.7%</td>
</tr>
<tr>
<td>javac</td>
<td>80M</td>
<td>12.8%</td>
<td>4.2%</td>
<td>83.0%</td>
</tr>
<tr>
<td>jess</td>
<td>102M</td>
<td>20.3%</td>
<td>0.0%</td>
<td>79.7%</td>
</tr>
<tr>
<td>mpeg</td>
<td>66M</td>
<td>20.0%</td>
<td>28.0%</td>
<td>52.0%</td>
</tr>
<tr>
<td>mtrt</td>
<td>259M</td>
<td>2.0%</td>
<td>0.3%</td>
<td>97.7%</td>
</tr>
<tr>
<td><strong>average</strong></td>
<td>112M</td>
<td>7.1%</td>
<td>3.1%</td>
<td>89.8%</td>
</tr>
</tbody>
</table>
Parameter Dependence Results

- Again the striking difference between static and dynamic results shows that runtime validation of static results is important.

- The *apparent* accuracy increases for most benchmarks, as memoization is not performed if there are zero dependencies. *jack*, *javac*, and *jess* benefit the most from this analysis, seeing up to 13% increases.

- However, the accuracy in a hybrid does not change.

- We’ll now take a look at memory savings with both analyses combined.
## Combined Analyses (context)

<table>
<thead>
<tr>
<th>benchmark</th>
<th>size</th>
<th>base</th>
<th>rvu</th>
<th>savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>comp</td>
<td>24</td>
<td>208M</td>
<td>208M</td>
<td>0.0%</td>
</tr>
<tr>
<td>db</td>
<td>24</td>
<td>361M</td>
<td>297M</td>
<td>17.7%</td>
</tr>
<tr>
<td>jack</td>
<td>14</td>
<td>10.5M</td>
<td>10.3M</td>
<td>1.9%</td>
</tr>
<tr>
<td>javac</td>
<td>20</td>
<td>211M</td>
<td>203M</td>
<td>3.8%</td>
</tr>
<tr>
<td>jess</td>
<td>14</td>
<td>9.15M</td>
<td>8.76M</td>
<td>4.3%</td>
</tr>
<tr>
<td>mpeg</td>
<td>12</td>
<td>2.27M</td>
<td>2.22M</td>
<td>2.2%</td>
</tr>
<tr>
<td>mtrt</td>
<td>14</td>
<td>46.1M</td>
<td>43.8M</td>
<td>5.0%</td>
</tr>
<tr>
<td>average</td>
<td>17</td>
<td>121M</td>
<td>120M</td>
<td>5.0%</td>
</tr>
</tbody>
</table>
## Combined Analyses (memoization)

<table>
<thead>
<tr>
<th>benchmark</th>
<th>size</th>
<th>base</th>
<th>rvu</th>
<th>pd</th>
<th>pdrvru</th>
<th>savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>comp</td>
<td>18</td>
<td>6.30M</td>
<td>6.24M</td>
<td>6.28M</td>
<td>6.22M</td>
<td>1.3%</td>
</tr>
<tr>
<td>db</td>
<td>24</td>
<td>206M</td>
<td>205M</td>
<td>195M</td>
<td>195M</td>
<td>5.3%</td>
</tr>
<tr>
<td>jack</td>
<td>8</td>
<td>939K</td>
<td>874K</td>
<td>718K</td>
<td>651K</td>
<td>30.7%</td>
</tr>
<tr>
<td>javac</td>
<td>14</td>
<td>65.7M</td>
<td>61.2M</td>
<td>51.8M</td>
<td>48.5M</td>
<td>26.2%</td>
</tr>
<tr>
<td>jess</td>
<td>12</td>
<td>5.01M</td>
<td>4.52M</td>
<td>4.41M</td>
<td>3.92M</td>
<td>21.8%</td>
</tr>
<tr>
<td>mpeg</td>
<td>12</td>
<td>589K</td>
<td>523K</td>
<td>544K</td>
<td>477K</td>
<td>19.0%</td>
</tr>
<tr>
<td>mtrt</td>
<td>12</td>
<td>15.1M</td>
<td>14.3M</td>
<td>14.8M</td>
<td>13.9M</td>
<td>7.9%</td>
</tr>
<tr>
<td>average</td>
<td>14</td>
<td>42.8M</td>
<td>41.8M</td>
<td>39.1M</td>
<td>38.4M</td>
<td>16.0%</td>
</tr>
</tbody>
</table>
Conclusions

- Introduced powerful memoization predictor
- Achieved high prediction accuracy in software JVM
- Cut memory costs without sacrificing accuracy
- Two new compiler analyses for improved RVP
  - Parameter dependence analysis: *production*
    - Optimistic analysis that does not require conservative correctness.
  - Return value use analysis: *consumption*
    - Relaxes safety constraints, but must be correct.
Future Work (1)

- Extend framework with new prediction strategies, implement generalised load predictors.
- Determine extent to which memoization compensates for concurrent update problems with context predictors.
- Allow for comparisons with locals in use expressions. At runtime, these values may be:
  - Parameter locals
  - Non-parameter locals
  - Stack values
Future Work (2)

- Implement purity analysis in Soot (Sălcianu, Rinard)
  - Skip pure methods altogether via memoization!
- Finish SMLP implementation in SableVM
  - Study costs and benefits of RVP in this system