

An Overview of the Indus Framework for Analysis and Slicing of Concurrent Java Software

SAnToS Laboratory, Kansas State University, USA

<http://indus.projects.cis.ksu.edu>

Indus & Kaveri

Venkatesh Ranganath
Ganeshan Jayaraman
John Hatcliff

Slicing for Model Checking

Matt Dwyer
John Hatcliff
Matt Hoosier

Venkatesh Ranganath
Robby
Todd Walletine

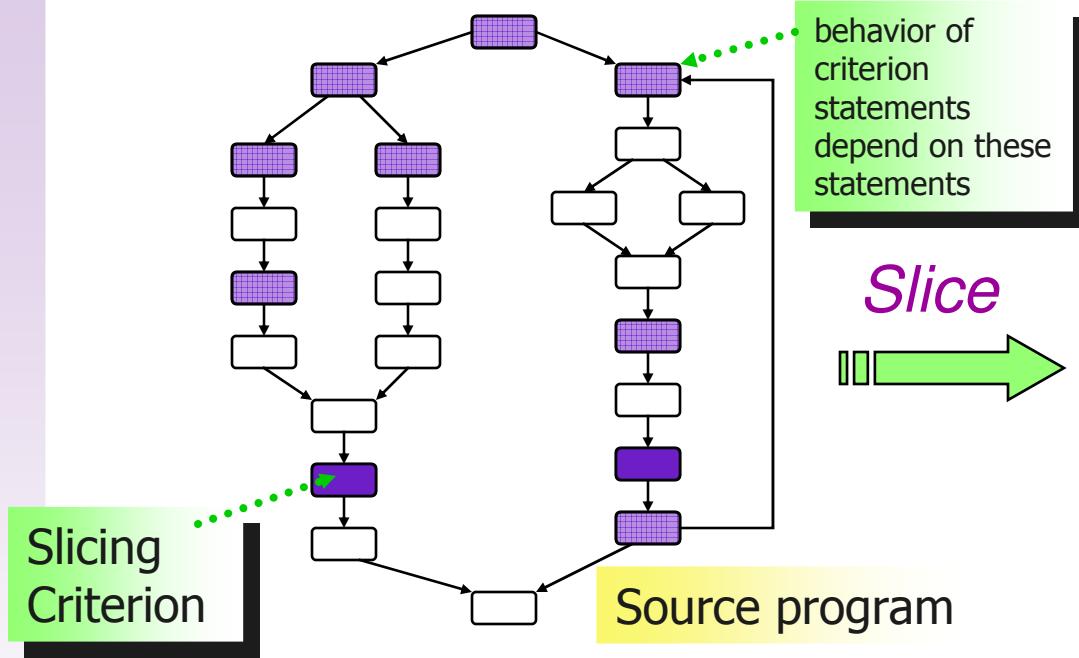
Support

US Army Research Office
US Defense Advanced Research Projects Agency (DARPA)
Lockheed Martin
National Science Foundation

*...thanks to Hongjun Zheng
for building the initial
prototype of our slicer!*

Program Slicing

Static Backwards Slicing



The screenshot shows the Eclipse IDE interface with a Java project named 'Example1'. The code editor displays the following Java code:

```
public void process1() {
    /* Show interspersed data flow through the slicer.
     */
    int _param1 = 10;
    process1(val, _param1 + 2);
}

private void process2(int val2, int _param1) {
    int _result = 10;
    if (val2 > 10)
        _result = result + val2;
    else {
        _result = _result - val2;
    }
    _result = _result * 10;
}
```

A statement `$1 = r0.<test: int val>` is highlighted in the code editor. A callout box at the bottom right provides the results of the slicing analysis:

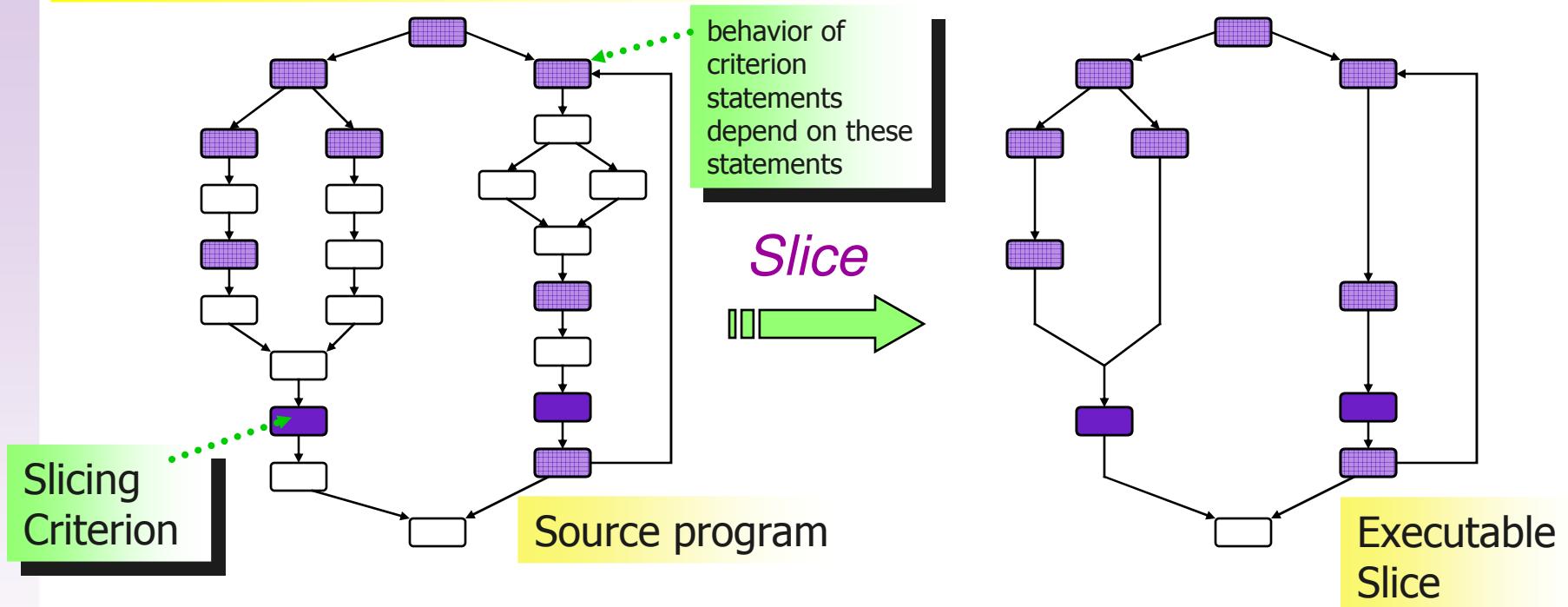
Simple Statement	Part of Slice
<code>\$1 = r0.<test: int val></code>	true
<code>\$2 = i0 + 2</code>	false
<code>specialInvoke r0.<test: void process1(int,int)>(\$1,\$2)</code>	true

Visualize & Query

- **slicing criterion** statements are provided by the user
- backwards slicing automatically finds all statements that...
 - ...the criterion statements **depend on**
 - ...might **influence** the behavior of the slicing criterion

Program Slicing

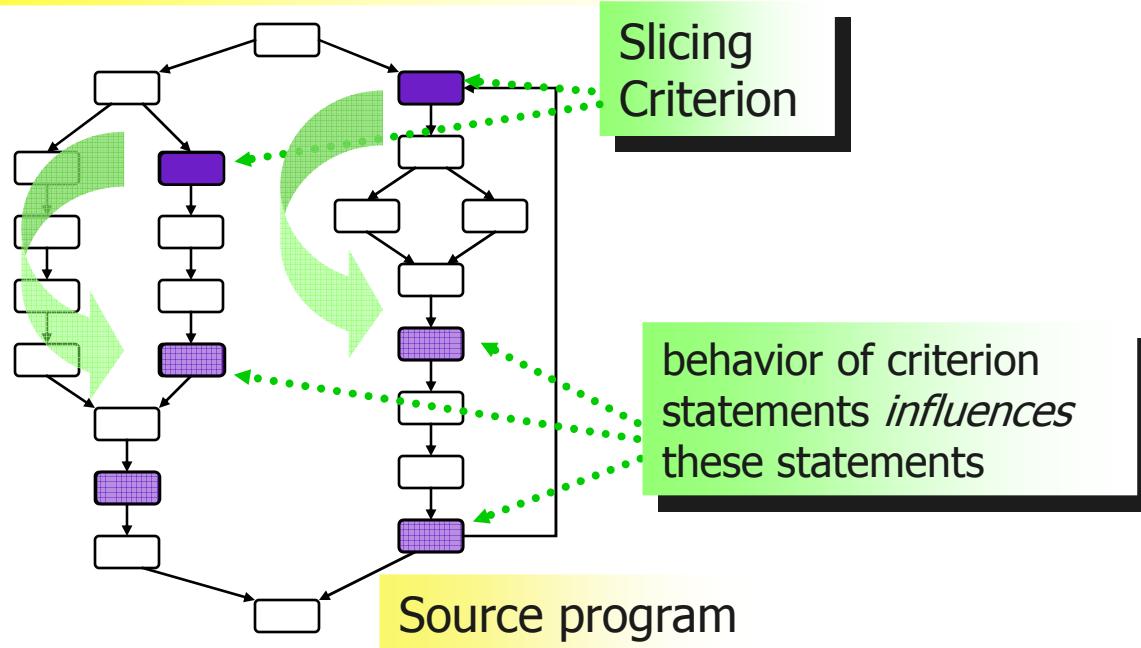
Static Backwards Slicing



- **slicing criterion** statements are provided by the user
- backwards slicing automatically finds all statements that...
 - ...the criterion statements **depend on**
 - ...might **influence** the behavior of the slicing criterion

Program Slicing

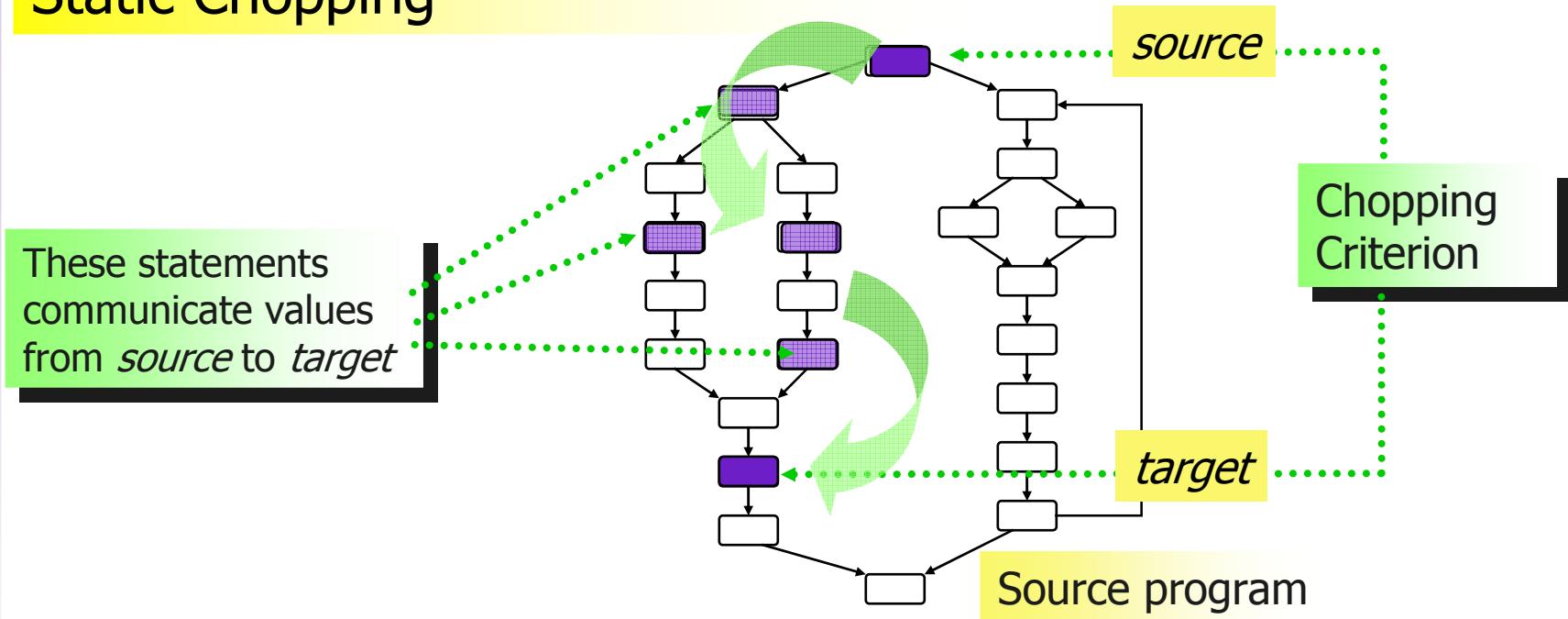
Static Forwards Slicing



- **slicing criterion** statements are provided by the user
- *forwards* slicing automatically finds all statements that...
 - ...the criterion statements **influence**
 - ...might **depend on** the behavior of the slicing criterion

Program Chopping

Static Chopping



- *source* and *target* **chopping criterion** statements are provided by the user
- *chopping* finds all statements that...
 - ...contribute to control and data flows from the source to the target
- roughly equivalent to intersection of source forward slice and target backward slice

Applications

Program Understanding



If I change this statement, what pieces of the program are going to be affected?

```
Prints a list of ODBC data sources and associated descriptions
DBCVER 0x0250
DS2
"win2os2.h"
sql.h>
sqltext.h>
in(int argc, char *argv[])
{
    HENV henv;
    RETCODE rc;
    UCHAR dsName[SQL_MAX_DSN_LENGTH+1];
    SWORD dsNameLen;
    UCHAR dsDesc[257];
    SWORD dsDescLen;
    rc = SQLAllocEnv(&henv);
    if (rc == SQL_ERROR)
    {
        printf("Error allocating environment\n");
        return -1;
    }

    // fetch all the data sources defined on this system
    while (
        SQLDataSources(henv, SQL_FETCH_NEXT, dsName,
        (SWORD) (SQL_MAX_DSN_LENGTH), &dsNameLen, dsDesc, 256, &dsDes
        != SQL_NO_DATA_FOUND
    )
    {
        printf("DSN: %s Desc: %s\n", dsName, dsDesc);
    }
    return 0;
}
```



Where are the values that flow into this statement coming from?

Forward slice with given statement as criteria

Backward slice with given statement as criteria

"Studies of software maintainers have shown that approximately 50% of their time is spent in the process of understanding the code that they are to maintain". Jussi Koskinen "Software Maintenance Costs"

Applications

Program Sub-setting

```

00001//-
00002// Prints a list of ODBC data sources and associated descriptions
00003//
00004//-----
00005 function parse ( $handle )
00006 {
00007     // Get the file
00008     $contents = $this->FILES[$handle];
00009     // If there's no template variables in the file, don't
00010     // bother processing them
00011     if ( strpos($contents, OPEN_VAR) === false )
00012     {
00013         echo $contents;
00014         return;
00015     }
00016     // Substitute global vars. This is the easy part
00017     foreach ( $this->VARS as $var_name =>
00018     {
00019         $contents = str_replace( OPEN_VAR . $var_name . CLOSE_VAR,
00020                                $this->vars[$var_name], $contents );
00021     }
00022     // If there's no block vars, don't bother processing them
00023     if ( strpos($contents, '<!-- BEGIN ') === false )
00024     {
00025         echo $contents;
00026         return;
00027     }
00028     // Now the tricky part: Substituting an HTML code block for multiple
00029     // Get all the blocks matching $block_name
00030     foreach ( $this->BLOCK_VARS as $block_name => $block_array )
00031     {
00032         // Get all the blocks matching $block_name
00033         $count = preg_match_all("#<!-- BEGIN $block_name -->(.+?)<!-- END $block_name --&gt;#", $contents, $block_matches);
00034         for ($i=0; $i &lt; $count; $i++)
00035         {
00036             $currentPower = $block_matches[1][$i];
00037             $powerOfTwo = 2;
00038         }
00039     }
00040     // Define 0000 API version 2.5
00041     // define _0000_API_0x0250
00042     return;
00043 }
</pre>

```



This code is bloated.
How can we limit the functionality to only what we need?

Backward slice on desired modules to obtain desired subset of functionality

```

function parse ( $handle )
{
    // Get the file
    $contents = $this->FILES[$handle];
    // If there's no template variables in the file, don't
    // bother processing them
    if ( strpos($contents, OPEN_VAR) === false )
    {
        echo $contents;
        return;
    }
    // Substitute global vars. This is the easy part
    foreach ( $this->VARS as $var_name => $var_value )
    {
        $contents = str_replace( OPEN_VAR . $var_name . CLOSE_VAR,
                                $var_value, $contents );
    }
    // If there's no block vars, don't bother processing them
    if ( strpos($contents, '<!-- BEGIN ') === false )
    {
        echo $contents;
        return;
    }
    // Now the tricky part: Substituting an HTML code block for multiple
    // Get all the blocks matching $block_name
    foreach ( $this->BLOCK_VARS as $block_name => $block_array )
    {
        $count = preg_match_all("#<!-- BEGIN $block_name -->(.+?)<!-- END $block_name --&gt;#", $contents, $block_matches);
        for ($i=0; $i &lt; $count; $i++)
        {
            $currentPower = $block_matches[1][$i];
        }
    }
}
</pre>

```

Applications

Impact Analysis

```
1 /**
2 * A simple program that prints all of the powers of two from 0 up to
3 * 31. After this is done, it prints 32 as a power of two.
4 */
5
6 public class PowerOfTwo {
7
8     /* A field that represent a power of two.
9      */
10    protected static int powerOfTwo = 1;
11
12    /* The main method.
13     */
14    public static void main(String[] args) {
15
16        // counter that will give us the exponent of the power of two.
17        int i = 0;
18
19        // loop to display the power of two.
20        while (powerOfTwo <= 32) {
21
22            // prints the next power of two we are currently at.
23            System.out.println("The current power of 2 is " + powerOfTwo);
24
25            // Prints the next power of two we are currently at.
26            powerOfTwo *= 2;
27
28        }
29    }
30
31    /**
32     * Prints a list of ODBC data sources and associated descriptions
33     */
34    public void printODBC() {
35
36        // define ODBC API version
37        #define _ODBCVER_ 0x0250
38        #define _ODBC_ _ODBC_
39        #include "win32.h"
40
41        /* Header files */
42        #include <sqle.h>
43        #include <sqlext.h>
44
45        prot
46
47        int main(int argc, char *argv[])
48        {
49            HENV henv; // environment handle
50            RETCODE rc; // error return code
51            UCHN dsname[SQL_MAX_DSN_LENGTH+1]; // name of the data source
52            DSNAME dsName; // description of data source name returned by
53            UCHN dsdesc[257]; // description of data source
54            SDESC dsDescLen; // holds length
55
56            rc = SQLAllocEnv(&henv); // allocate the environment
57            if (rc != SQL_ERROR)
58            {
59
60                printf("Program allocation successful\n");
61
62                ret
63
64                /* A simple program that prints all of the powers of two from
65                 * 0 up to 31. After this is done, it prints 32 as a power
66                 */
67                public class PowerOfTwo {
68
69                    /* A field that represent a power of two.
70                     */
71                    protected static int powerOfTwo = 1;
72
73                    /* The main method.
74                     */
75                    public static void main(String[] args) {
76
77                        // counter that will give us the exponent of the power of two.
78                        int i = 0;
79
80                        // loop to display the power of two.
81                        while (powerOfTwo <= 32) {
82
83                            // prints the next power of two we are currently at.
84                            System.out.println("The current power of 2 is " + powerOfTwo);
85
86                            // update the exponent
87                            i++;
88
89                        }
89
90                        System.out.println("2^" + i + " = " + powerOfTwo);
91
92                    }
93
94                }
95
96                /**
97                 * Given that code->currentPower is a power of two,
98                 * code->getNextPower(int) will return the next power of two after
99                 * code->currentPower.
100                */
101                DSNAME code;
102                code->currentPower = 1;
103                code->getNextPower(code->currentPower);
104                code->currentPower = code->getNextPower(code->currentPower);
105
106                protected static int getNextPower(int currentPower) {
107                    return currentPower + 2;
108                }
109
110            }
111
112        }
113
114    }
115
116}
```



My regression tests take 20 hours to run. How can I limit the tests to only those that exercise the changed code?

relevant
tests

change

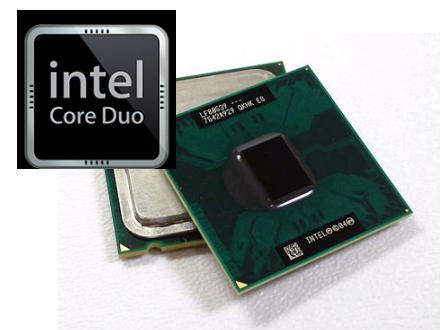
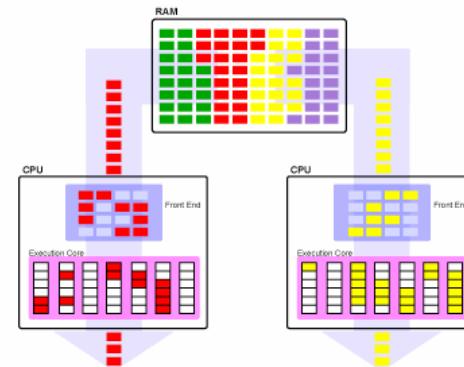
Backward slice
with changed
statements as
criteria

Test suite

Current Trends

Slicing in a multi-threaded context?

- Significant increase in use of multi-threading due to languages with built-in threading primitives such as Java and C#
- Dramatic increase in use of concurrency in near future due to proliferation of multi-core processors



Slicing Multi-Threaded Programs

Questions...

- What are the basic notions of dependence required for multi-threaded Java programs?
- What sort of tool infrastructure is needed to calculate slices and present the results to developers?
- What optimizations and engineering decisions are necessary for scaling slicing?
- What are some non-conventional applications of slicing that arise in a multi-threaded setting?



Let's move beyond queries like:
“what other statements does my current statement depend on?”

Beyond Traditional Slicing Questions

Mining the results of wide-ranging static analysis for concurrency

What other synchronized statements may acquire the lock used in the current monitor?



```
R 0x0250
42os2.h"
list of ODBC data sources and associated descriptions
on 2.5

00014 int main(int argc, char *argv[])
00015 {
00016     HENV hEnv;
00017     RETCODE rc;
00018     UCHAR szName[ODBC_MAX_NAME];
00019     SWORD nNameLen;
00020     UCHAR szDesc[ODBC_MAX_DESCR];
00021     SWORD nDescLen;
00022
00023     rc = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hEnv);
00024     if (rc == SQL_ERROR)
00025     {
00026         printf("Error %d\n", rc);
00027         return -1;
00028     }
00029
00030     // Prints a list of ODBC data sources and associated descriptions
00031
00032     // define ODBC API version 2.5
00033
00034     /* Fetch all the data sources defined */
00035     while (SQLGetAllDataSources(hEnv, SQL_FETCH_NEXT,
00036                                 &nNameLen, szName, &nDescLen, szDesc) ==
00037             SQL_NO_DATA_FOUND)
00038     {
00039         printf("DSN: %s Desc: %s\n", szName, szDesc);
00040     }
00041
00042     return 0;
00043 }
```

What other statements may generate concurrent conflicting writes with the current statement?

What locks are held when this statement is executed?



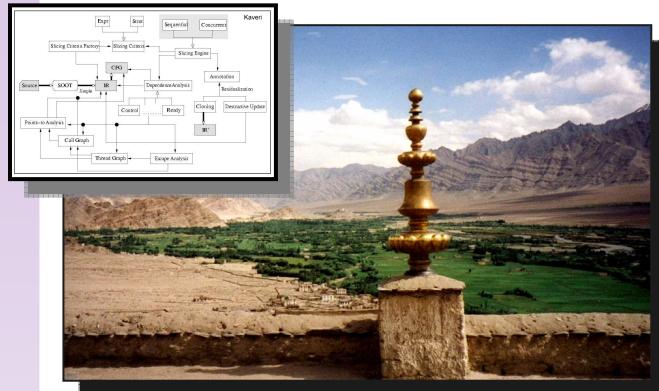
Which notify statements may “wake up” this particular wait statement?

```
00008 // Fetch all the data sources defined
00009 while (
00010     SQLGetAllDataSources(hEnv, SQL_FETCH_NEXT,
00011                           &nNameLen, szName, &nDescLen, szDesc) ==
00012                           SQL_NO_DATA_FOUND)
00013     {
00014         printf("DSN: %s Desc: %s\n", szName, szDesc);
00015     }
00016
00017     return 0;
00018 }
```

Does method *m* write to any data cell reachable from argument *a*?

...we are interested in going well beyond than the usual slicing questions

Indus



Indus provides a rich collection of program analyses and transformations for Java programs (full Java)

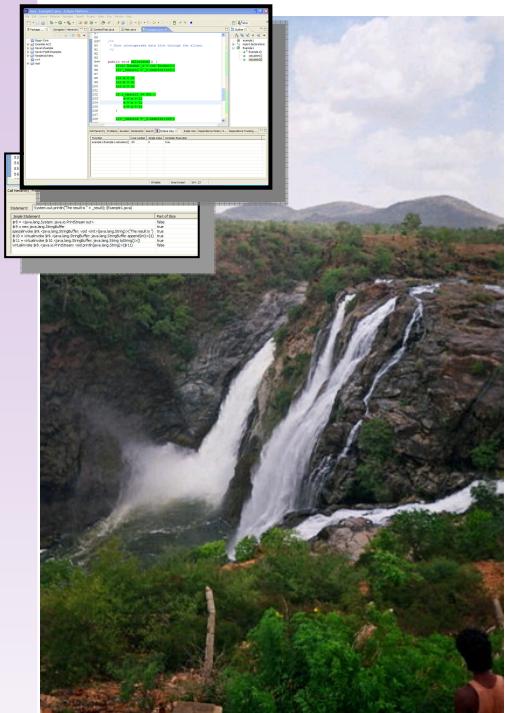
I. Platform for static analyses such as points-to analysis, escape analysis, and dependence analyses,

- ... Object-flow Analysis (OFA) ... Side-Effect Analysis ... Safe Lock Analysis
- ... Escape Analysis ... Monitor Analysis ... Dependence Analyses

II. Transformations such as program slicing and program specialization via partial evaluation

- ... Backwards and Forwards Static Slicing ... Slice restriction using scope specifications and call paths
- ... Generation of executable slices ... Property-aware slicing

Kaveri



Kaveri provides a featureful Eclipse-based UI front-end for Indus integrated with the Eclipse Java development environment

-
- I. Configuring, invoking and visualizing results of Indus slicing
 - II. Scripting framework for flexible querying of underlying analysis APIs

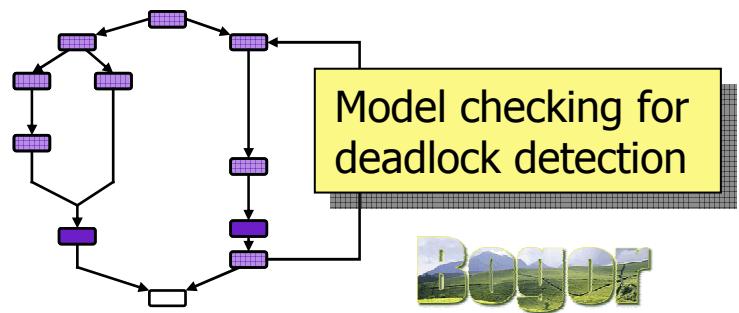
“According to Hindu tradition, the Kaveri river cleanses all sins.”

Outline

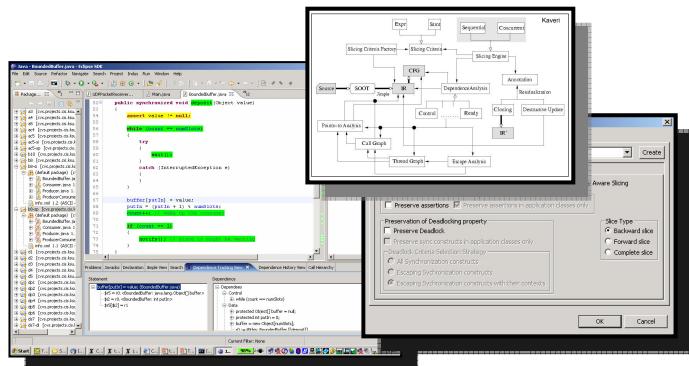
I. Program Dependences for Java

```
public synchronized void deposit(Object value) {  
    while (count == numSlots) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
    buffer[putIn] = value;  
    putIn = (putIn + 1) % numSlots;  
    count++;  
  
    if (count == 1)  
        notify();  
    }  
}  
  
public synchronized Object fetch() {  
    Object value;  
  
    while (count == 0) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
  
    value = buffer[takeOut];  
    takeOut = (takeOut + 1) % numSlots;  
    count--;  
  
    if (count == (numSlots - 1))  
        notify();  
    }  
    return value;  
}
```

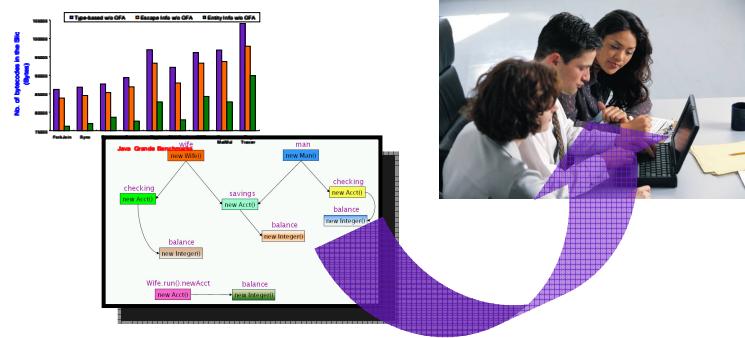
III. Application to Verifying Concurrent Java Programs



II. Indus & Kaveri



IV. Analysis optimization and mining analysis results



Dependence

Various notions of Dependences [Hatcliff et. al.:SAS99]

- Intra-thread data dependence | - The usual stuff...
- Control dependence | - Dependences related to threading & locking...
- Interference dependence | - Dependences for preserving liveness properties...
- Synchronization dependence
- Divergence dependence
- Ready dependence

...dealing with all of these in the presence of, e.g., pointers, exceptions, etc.

Bounded Buffer Example

```
class BoundedBuffer {  
    protected int numSlots = 0; /* size of buffer */  
    protected Object[] buffer = null; /* buffer array */  
    protected int putIn = 0; /* next empty slot */  
    protected int takeOut = 0; /* next item */  
    protected int count = 0; /* number items */  
  
    public BoundedBuffer(int numSlots)  
    {...}  
  
    public synchronized void deposit(Object value)  
    {...}  
  
    public synchronized Object fetch ()  
    {...}  
}
```

Data Dependence

```
public class BoundedBuffer {  
    protected int numSlots = 0;  
    protected Object[] buffer = null;  
    protected int putIn = 0;  
    protected int takeOut = 0;  
    protected int count = 0;  
  
    public BoundedBuffer(int numSlots) {  
        if (numSlots <= 0) {  
            throw new IllegalArgumentException();  
        }  
  
        this.numSlots = numSlots;  
        buffer = new Object[numSlots];  
    }  
  
    ....  
}
```

Data Dependence

Definition of variable V
at statement S_1
reaches a use of V at
statement S_2

...these statements
depend on the data value
assigned to parameter
numSlots

Data Dependence

```
public class BoundedBuffer {  
    protected int numSlots = 0;  
    protected Object[] buffer = null;  
    protected int putIn = 0;  
    protected int takeOut = 0;  
    protected int count = 0;  
  
    public BoundedBuffer(int numSlots) {  
        if (numSlots <= 0) {  
            throw new IllegalArgumentException();  
        }  
  
        this.numSlots = numSlots;  
        buffer = new Object[numSlots];  
    }  
  
    ....  
}
```

Data Dependence

Definition of variable V
in statement S_1
reaches a use of V at
statement S_2

...these statements
depend on the data value
assigned to parameter
numSlots

Data Dependence

```
public synchronized void deposit(Object value) {  
    while (count == numSlots) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
  
    buffer[putIn] = value;  
    putIn = (putIn + 1) % numSlots;  
    count++;  
  
    if (count == 1) {  
        notify();  
    }  
}
```

Data Dependence
w/ Interprocedural
Control Flow

```
public synchronized Object fetch() {  
    Object value;  
  
    while (count == 0) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
  
    value = buffer[takeOut];  
    takeOut = (takeOut + 1) % numSlots;  
    count--;  
  
    if (count == (numSlots - 1)) {  
        notify();  
    }  
    return value;  
}
```

Data Dependence

```
public synchronized void deposit(Object value) {  
    while (count == numSlots) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
  
    buffer[putIn] = value;  
    putIn = (putIn + 1) % numSlots;  
    count++;  
  
    if (count == 1) {  
        notify();  
    }  
}
```

Data Dependence
w/ Interprocedural
Control Flow

```
public synchronized Object fetch() {  
    Object value;  
  
    while (count == 0) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
  
    value = buffer[takeOut];  
    takeOut = (takeOut + 1) % numSlots;  
    count--;  
  
    if (count == (numSlots - 1)) {  
        notify();  
    }  
    return value;  
}
```

Data Dependence

```
public synchronized void deposit(Object value) {  
    while (count == numSlots) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
  
    buffer[putIn] = value;  
    putIn = (putIn + 1) % numSlots;  
    count++;  
  
    if (count == 1) {  
        notify();  
    }  
}
```

Data Dependence
w/ Interprocedural
Control Flow

```
public synchronized Object fetch() {  
    Object value;  
  
    while (count == 0) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
  
    value = buffer[takeOut];  
    takeOut = (takeOut + 1) % numSlots;  
    count--;  
  
    if (count == (numSlots - 1)) {  
        notify();  
    }  
    return value;  
}
```

Interference Dependence

```
public synchronized void deposit(Object value) {  
    while (count == numSlots) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
  
    buffer[putIn] = value;  
    putIn = (putIn + 1) % numSlots;  
    count++;  
  
    if (count == 1) {  
        notify();  
    }  
}
```

Data Dependence
across threads

```
public synchronized Object fetch() {  
    Object value;  
  
    while (count == 0) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
  
    value = buffer[takeOut];  
    takeOut = (takeOut + 1) % numSlots;  
    count--;  
  
    if (count == (numSlots - 1)) {  
        notify();  
    }  
    return value;  
}
```

Control Dependence

```
public class BoundedBuffer {  
    protected int numSlots = 0;  
    protected Object[] buffer = null;  
    protected int putIn = 0;  
    protected int takeOut = 0;  
    protected int count = 0;  
  
    public BoundedBuffer(int numSlots) {  
        if (numSlots <= 0) {  
            throw new IllegalArgumentException();  
        }  
  
        this.numSlots = numSlots;  
        buffer = new Object[numSlots];  
    }  
  
    ....  
}
```

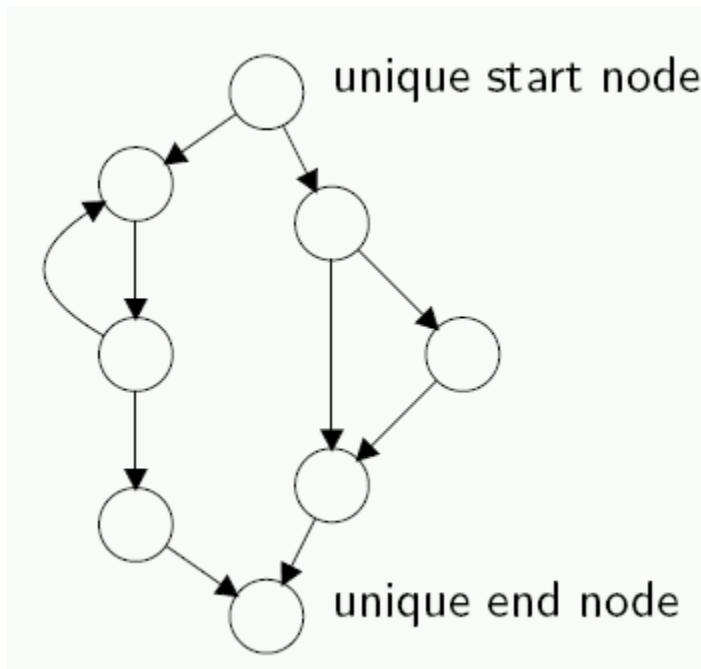
Control Dependence

A conditional statement controls whether or not the current statement is executed

...this statement depends on the conditional which controls whether it is reached or bypassed by throwing an exception

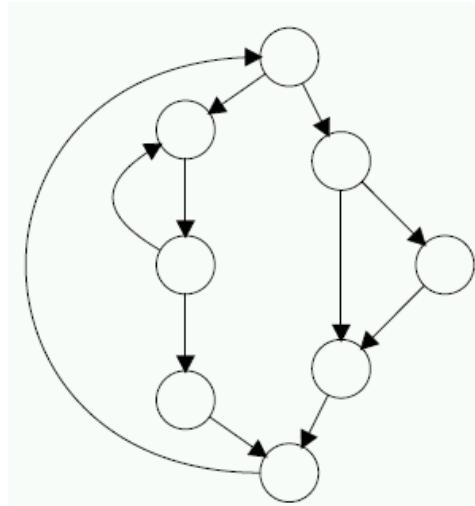
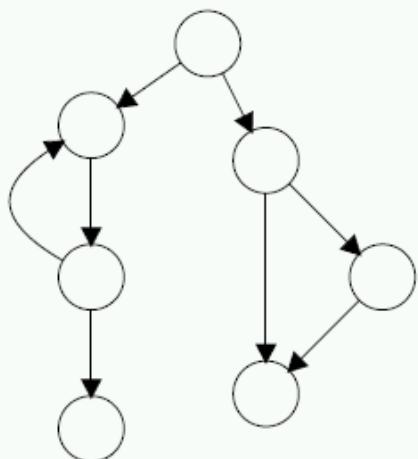
Enhanced Control Dependence

Conventional definitions of control dependence are presented in terms of CFGs with unique start and end nodes



Enhanced Control Dependence

...but CFGs of Java often violate this constraint



Multiple end nodes

- multiple return nodes in a method
- exceptional exits from a method

No end node

- reactive programs (designed to run indefinitely)
- event loops in GUI frameworks, etc.

Enhanced Control Dependence

Indus uses an enhanced notion of control dependence that...

- handles CFGs with multiple ends and no end nodes
- works with both reducible (structured programs) and irreducible (unstructured programs) CFGs
- has formal correctness properties established using a notion of weak bisimulation for both finite and infinite program traces

Ranganath et. al., ESOP 2005

Divergence Dependence

```
public synchronized void deposit(Object value) {  
    while (count == numSlots) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
  
    buffer[putIn] = value;  
    putIn = (putIn + 1) % numSlots;  
    count++;  
  
    if (count == 1) {  
        notify();  
    }  
}
```

Divergence Dependence

Statement S_1 influences S_2 if S_1 may infinitely delay S_2 (e.g., by going into an infinite loop)

*...capturing
delaying
influences within
a single thread*

```
public synchronized Object fetch() {  
    Object value;  
  
    while (count == 0) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
  
    value = buffer[takeOut];  
    takeOut = (takeOut + 1) % numSlots;  
    count--;  
  
    if (count == (numSlots - 1)) {  
        notify();  
    }  
    return value;  
}
```

Ready Dependence

```
public synchronized void deposit(Object value) {  
    while (count == numSlots) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
  
    buffer[putIn] = value;  
    putIn = (putIn + 1) % numSlots;  
    count++;  
  
    if (count == 1)  
        notify();  
}
```

Ready Dependence

The failure of a lock release to complete may cause a lock acquire to be delayed (blocked) indefinitely

*...capturing
delaying influences
across threads*

```
public synchronized Object fetch() {  
    Object value;  
  
    while (count == 0) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
  
    value = buffer[takeOut];  
    takeOut = (takeOut + 1) % numSlots;  
    count--;  
  
    if (count == (numSlots - 1))  
        notify();  
    }  
    return value;  
}
```

Ready Dependence

```
public synchronized void deposit(Object value) {  
    while (count == numSlots) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
  
    buffer[putIn] = value;  
    putIn = (putIn + 1) % numSlots;  
    count++;  
  
    if (count == 1)  
        notify();  
}
```

```
public synchronized Object fetch() {  
    Object value;  
  
    while (count == 0) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
  
    value = buffer[takeOut];  
    takeOut = (takeOut + 1) % numSlots;  
    count--;  
  
    if (count == (numSlots - 1))  
        notify();  
    }  
    return value;  
}
```

Ready Dependence

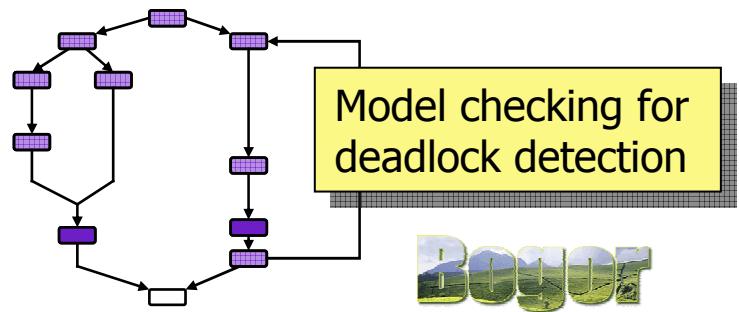
The failure of a `notify` to complete may cause a `wait` to be delayed indefinitely

Outline

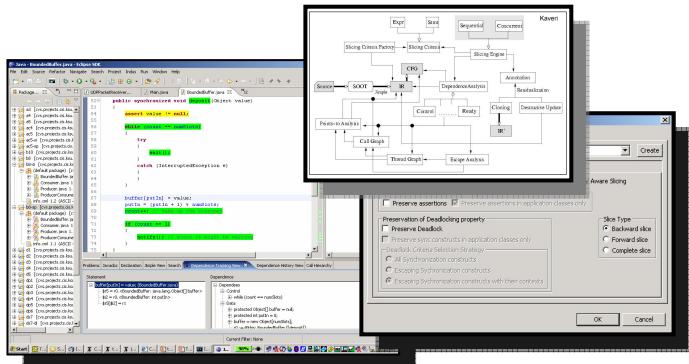
I. Program Dependences for Java

```
public synchronized void deposit(Object value) {  
    while (count == numSlots) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
    buffer[putIn] = value;  
    putIn = (putIn + 1) % numSlots;  
    count++;  
  
    if (count == 1)  
        notify();  
    }  
}  
  
public synchronized Object fetch() {  
    Object value;  
  
    while (count == 0) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
  
    value = buffer[takeOut];  
    takeOut = (takeOut + 1) % numSlots;  
    count--;  
  
    if (count == (numSlots - 1))  
        notify();  
    }  
    return value;  
}
```

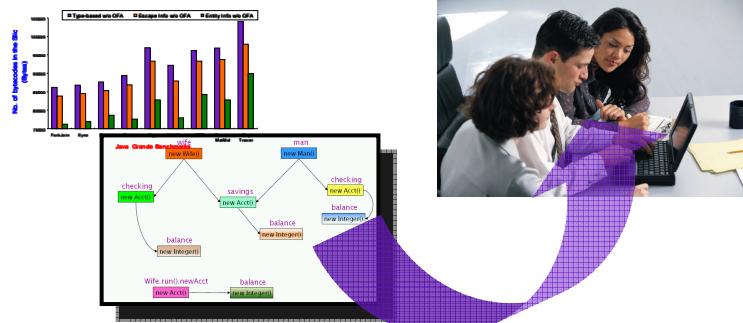
III. Application to Verifying Concurrent Java Programs



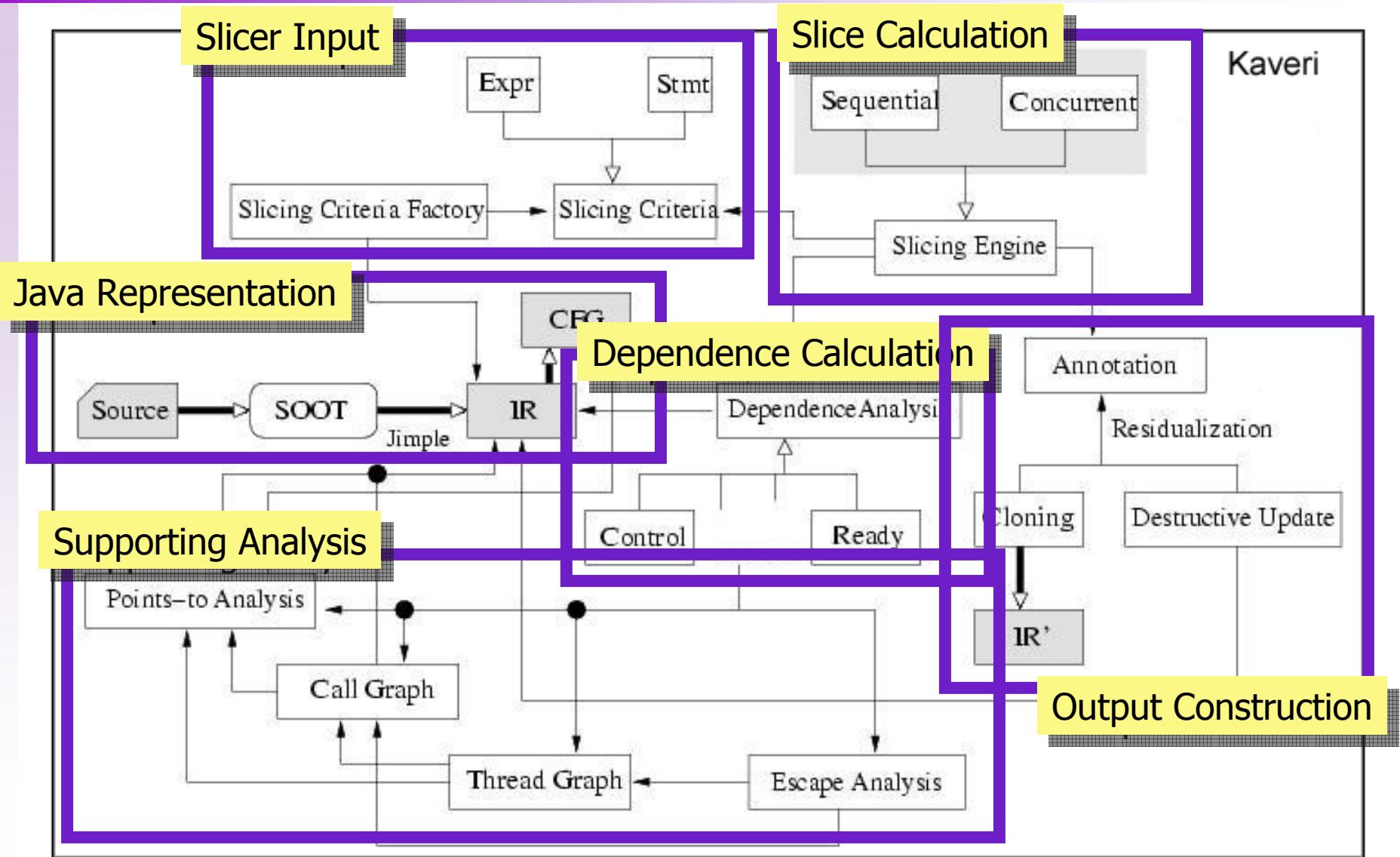
II. Indus & Kaveri



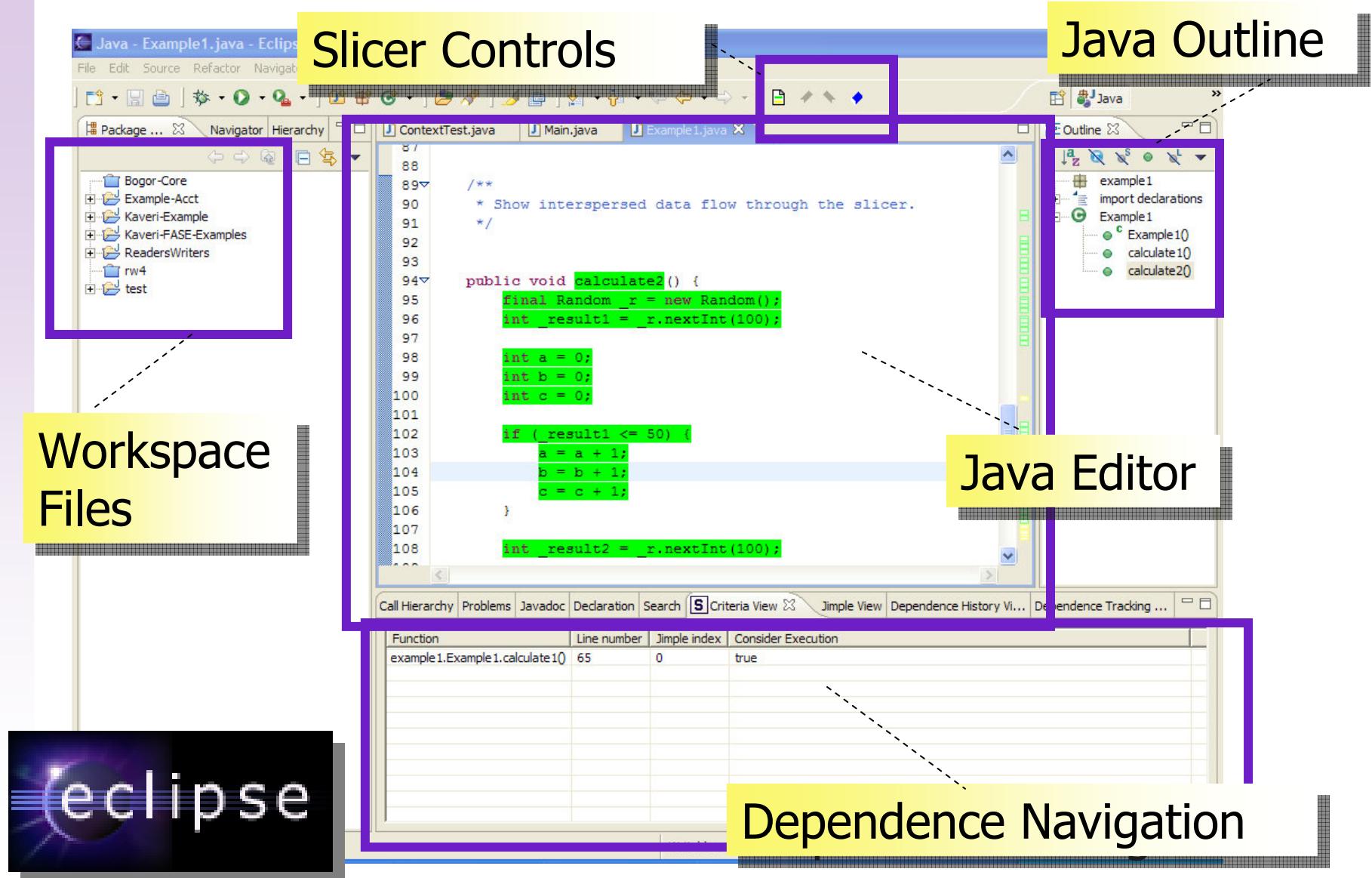
IV. Analysis optimization and mining analysis results



Indus / Kaveri Architecture

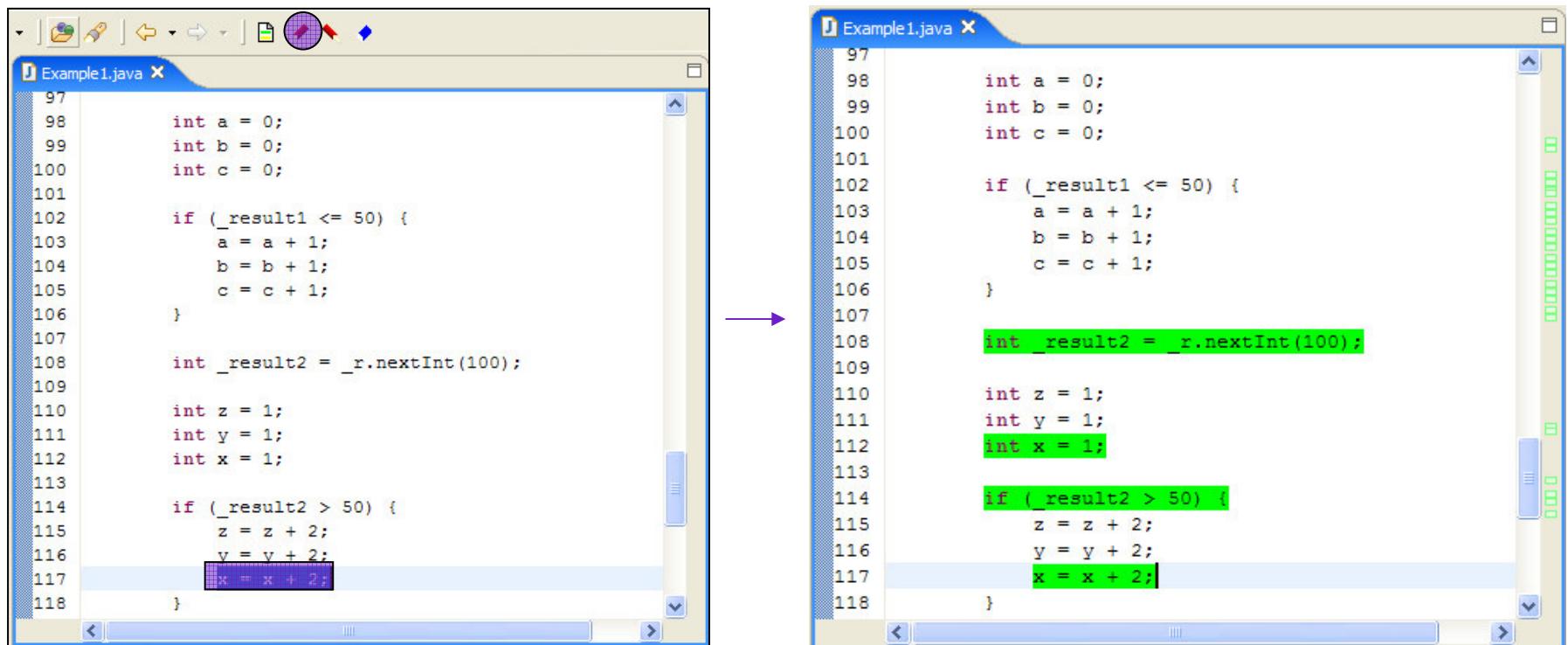


Theme: Java Development Integration



Basic Use Mode

Launch Backward Slice for single statement criteria



The figure displays two screenshots of a Java IDE interface. On the left, the code for `Example1.java` is shown with several statements highlighted in purple, indicating they are part of the current slice. On the right, the same code is shown after applying a backward slice, where only the statements that affect the selected criteria are highlighted in green. A purple arrow points from the left editor to the right editor, indicating the transformation process.

```
Example1.java (Left Editor):
97     int a = 0;
98     int b = 0;
99     int c = 0;
100
101    if (_result1 <= 50) {
102        a = a + 1;
103        b = b + 1;
104        c = c + 1;
105    }
106
107    int _result2 = _r.nextInt(100);
108
109    int z = 1;
110    int y = 1;
111    int x = 1;
112
113    if (_result2 > 50) {
114        z = z + 2;
115        y = y + 2;
116        x = x + 2;
117    }
118

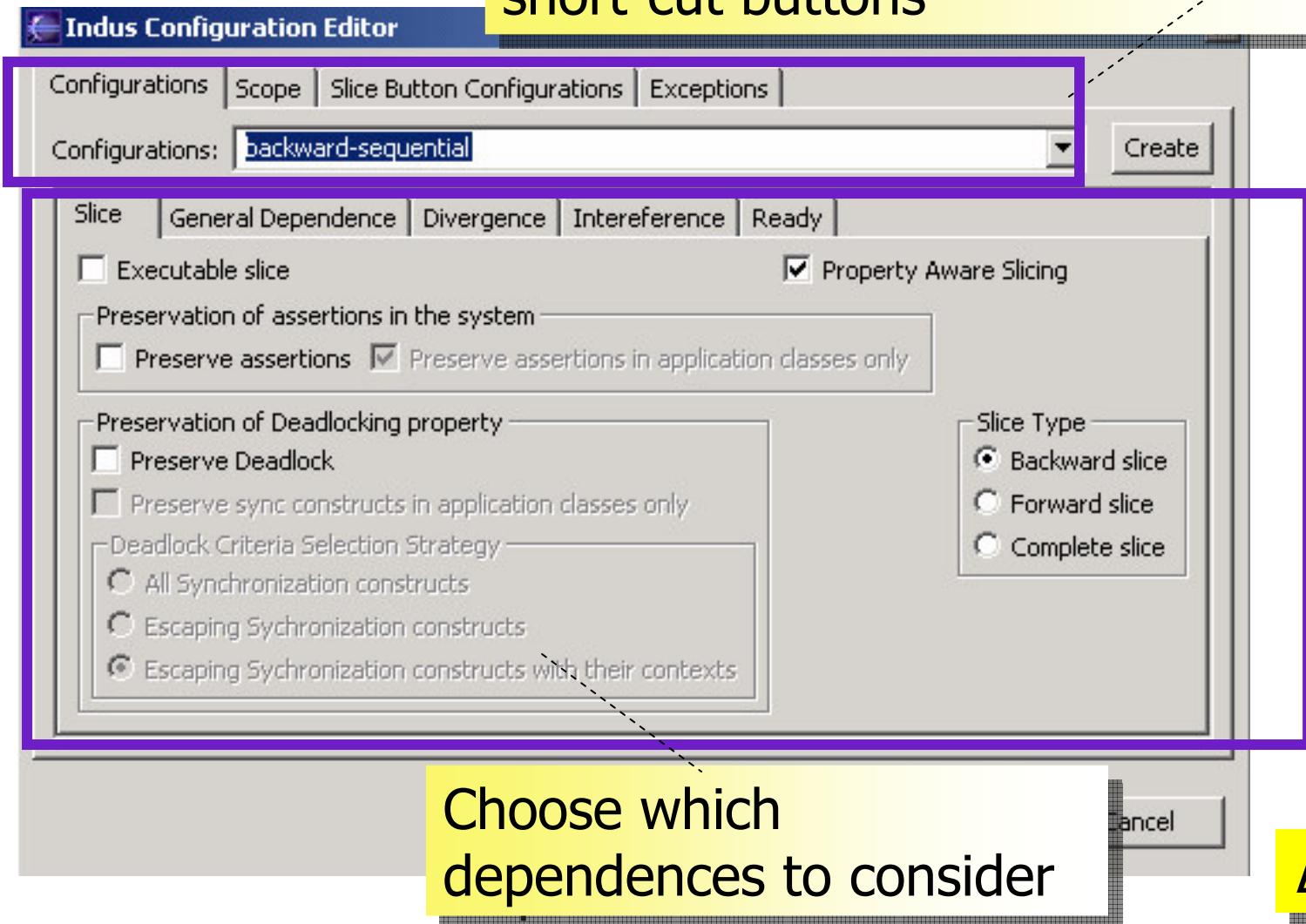
Example1.java (Right Editor):
97
98
99
100
101
102    if (_result1 <= 50) {
103        a = a + 1;
104        b = b + 1;
105        c = c + 1;
106    }
107
108    int _result2 = _r.nextInt(100);
109
110    int z = 1;
111    int y = 1;
112    int x = 1;
113
114    if (_result2 > 50) {
115        z = z + 2;
116        y = y + 2;
117        x = x + 2;
118    }
```

- A single statement is used as the criteria

Demo

Demo: Slicer Configuration

Name/Save configurations and bind to short-cut buttons



Source Code / Byte Code Issues

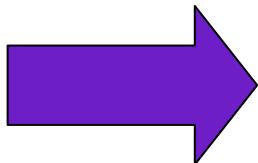
Understanding the correspondence between Java source and byte code representation (Jimple)

- We would like to enable developers to work at the Java source level...
- ...yet in many cases the semantics of Java and dependences of Java can only be explained precisely by working at the byte code level

Jimple Representation of Java

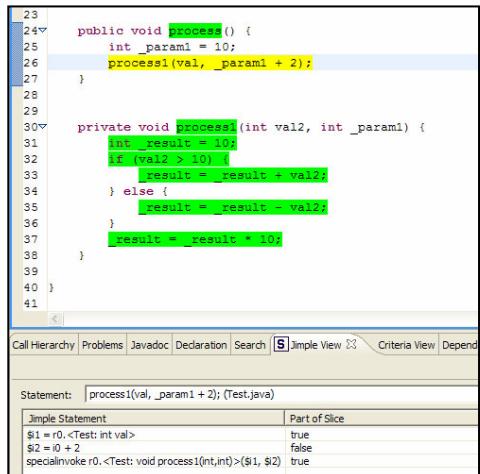
Jimple is an intermediate representation of Java

if (!empty(Clist))



- \$r4 = r0.<temp.DiskScheduler:java.util.LinkedList Clist>
- \$z0 = virtualinvoke r0.<temp.DiskScheduler: boolean empty(java.util.LinkedList)>(\$r4)
- if \$z0 != 0 goto \$r7 = r0.<temp.DiskScheduler:java.util.LinkedList NList>

Kaveri provides a special view to see the Jimple statements that correspond to a particular Java statement



The screenshot shows a Java code editor with the following code:

```
23  public void process() {
24    int _param1 = 10;
25    process1(val, _param1 + 2);
26  }
27
28
29
30  private void process1(int val2, int _param1) {
31    int result = 10;
32    if (val2 > 10) {
33      result = result - val2;
34    } else {
35      result = result - val2;
36    }
37    result = result * 10;
38  }
39
40 }
41
```

Below the code editor, a "Jimple View" tab is selected. The "Statement:" dropdown shows "process1(val, _param1 + 2); (Test.java)". The table below lists the Jimple statements for this statement:

Jimple Statement	Part of Slice
\$1 = r0.<Test: int val>	true
\$2 = 10 + 2	false
specialinvoke r0.<Test: void process1(int,int)>(\$1, \$2)	true

Java

Jimple

Jimple Representation of Java

Example

The screenshot shows a Java code editor with the following code:

```
98     int a = 0;
99     int b = 0;
100    int c = 0;
101
102    if (_result1 <= 50) {
103        a = a + 1;
104        b = b + 1;
105        c = c + 1; // Statement highlighted with a blue selection bar and an orange arrow pointing to it.
106    }
107
108
```

Below the code, the IDE interface includes tabs for Call Hierarchy, Problems, Javadoc, Declaration, Search, Jimple View (which is selected), and Criteria View.

The Jimple View panel displays the statement `c = c + 1;` from `(Example1.java)`. It has two tabs: **Jimple Statement** and **Part of Slice**. The **Jimple Statement** tab shows the Jimple representation `i3 = i3 + 1`, which is circled in orange. The **Part of Slice** tab shows the value `true`, also circled in orange.

Java Statement

- Some Java statements correspond to a single Jimple statement

Jimple Representation of Java

Example

The screenshot shows a software interface for analyzing Java code. At the top, there's a code editor window displaying Java code:

```
83     System.out.println("Random value was " + 4);
84 }
85 System.out.println("The result is " + _result);
86 }
```

A red arrow points from the word "Statement" in the title "Java Statement" to the highlighted line of code "System.out.println("The result is " + _result);".

Below the code editor is a navigation bar with tabs: Call Hierarchy, Problems, Javadoc, Declaration, Search, S Jimple View, Criteria View, and Dependence History Vi... The "S Jimple View" tab is selected.

The main area displays the "Statement:" field containing the selected Java statement: "System.out.println("The result is " + _result); (Example1.java)".

Below this, a table titled "Jimple Statement" shows the underlying Jimple representation:

Jimple Statement	Part of Slice
\$r8 = <java.lang.System: java.io.PrintStream out>	false
\$r9 = new java.lang.StringBuffer	true
specialinvoke \$r9.<java.lang.StringBuffer: void <init>(java.lang.String)>"(The result is ")	true
\$r10 = virtualinvoke \$r9.<java.lang.StringBuffer: java.lang.StringBuffer append(int)>(i1)	true
\$r11 = virtualinvoke \$r10.<java.lang.StringBuffer: java.lang.String toString()>()	true
virtualinvoke \$r8.<java.io.PrintStream: void println(java.lang.String)>(\$r11)	false

An orange oval encloses the entire "Jimple Statement" table.

Java Statement

- Some Java statements correspond to several Java statements
- The example shows the underlying use of *StringBuffer* to perform String concatenation

Jimple View

Current Statement

```
23
24    public void process() {
25        int _param1 = 10;
26        process1(val, _param1 + 2);
27    }
28
29
30    private void process1(in
31        int _result = 10;
32        if (val2 > 10) {
33            _result = _result + val2;
34        } else {
35            _result = _result - val2;
36        }
37        _result = _result * 10;
38    }
39
40 }
41
```

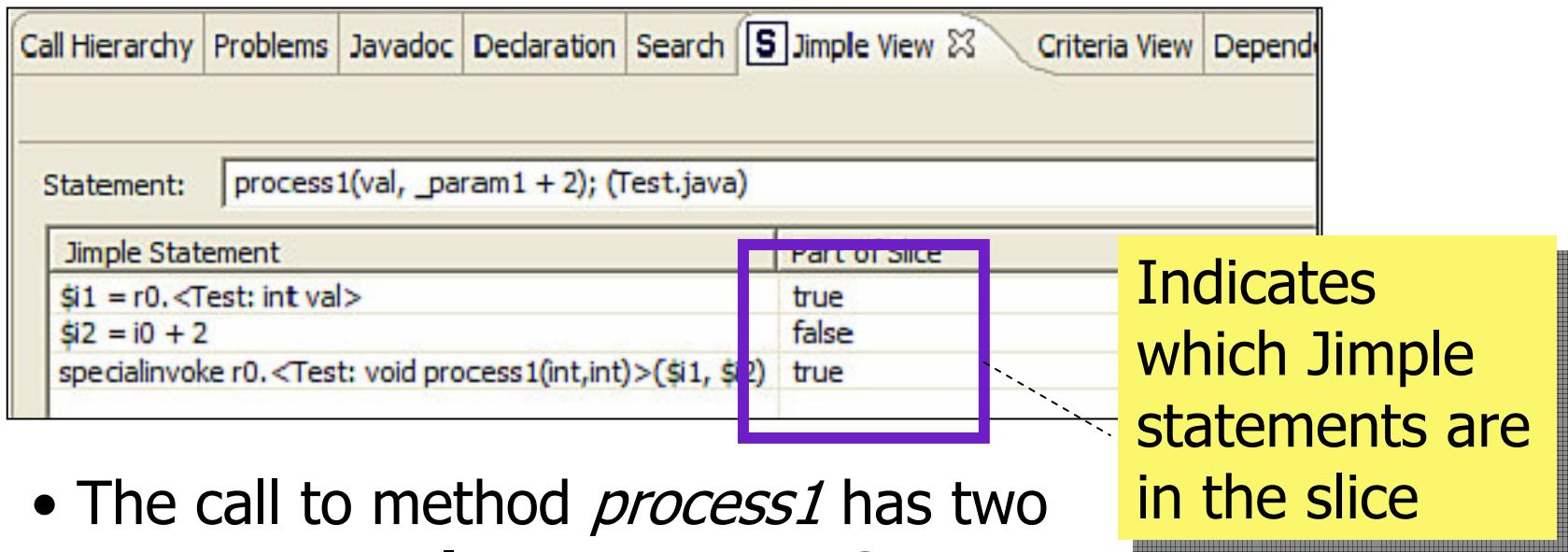
Yellow = *some corresponding Jimple in the slice*

Criteria

Green = *all corresponding Jimple in the slice*

Jimple Statement	Part of Slice
\$i1 = r0.<Test: int val>	true
\$i2 = i0 + 2	false
specialinvoke r0.<Test: void process1(int,int)>(\$i1, \$i2)	true

Jimple View



The screenshot shows a software interface titled "Jimple View". The menu bar includes "Call Hierarchy", "Problems", "Javadoc", "Declaration", "Search", "Jimple View", "Criteria View", and "Dependencies". The "Jimple View" tab is selected. Below the menu, a status bar displays "Statement: process1(val, _param1 + 2); (Test.java)". The main area contains a table titled "Jimple Statement". The table has two columns: "Jimple Statement" and "Part of Slice". The "Jimple Statement" column contains the following code:

```
$i1 = r0.<Test: int val>
$i2 = i0 + 2
specialinvoke r0.<Test: void process1(int,int)>($i1, $i2)
```

The "Part of Slice" column contains three rows with values "true", "false", and "true". A purple rectangular box highlights the "Part of Slice" column. A yellow callout box with a dashed arrow points from the highlighted column to the text "Indicates which Jimple statements are in the slice".

Jimple Statement	Part of Slice
<code>\$i1 = r0.<Test: int val></code>	true
<code>\$i2 = i0 + 2</code>	false
<code>specialinvoke r0.<Test: void process1(int,int)>(\$i1, \$i2)</code>	true

- The call to method *process1* has two parameters: **val**, **_param1 + 2**
- Only **val** is used in the method *process1*
- Hence, only the parameter **val** has the slice tag

Demo

Now Let's Slice

Setting the Slice Criterion

- A *Slicing Criterion* indicates a point of interest for slicing
 - Represented by a Jimple statement in Kaveri
 - The slicing criteria for a slice can be:
 - ...one Jimple statement
 - ...multiple Jimple statements

Criteria List

Slicing Criteria can be either a *single* Jimple statement or *multiple* Jimple statements

The screenshot shows a Java code editor with the following code:

```
97     int a = 0;
98     int b = 0;
99     int c = 0;
100
101    if (_result1 <= 50) {
102        a = a + 1;
103        b = b + 1;
104        c = c + 1;
105    }
106
107    int _result2 = _r.nextInt(100);
108
109    int z = 1;
110    int y = 1;
111    int x = 1;
112
113    if (_result2 > 50) {
114        z = z + 2;
115        y = y + 2;
116        x = x + 2;
117    }
118 }
```

The line `int _result2 = _r.nextInt(100);` is highlighted in green, indicating it is the selected criteria statement.

Single statement

The screenshot shows a Java code editor with the following code:

```
97     int a = 0;
98     int b = 0;
99     int c = 0;
100
101    if (_result1 <= 50) {
102        a = a + 1;
103        b = b + 1;
104        c = c + 1;
105    }
106
107    int _result2 = _r.nextInt(100);
108
109    int z = 1;
110    int y = 1;
111    int x = 1;
112
113    if (_result2 > 50) {
114        z = z + 2;
115        y = y + 2;
116        x = x + 2;
117    }
118 }
```

The lines `a = a + 1;`, `b = b + 1;`, and `c = c + 1;` are highlighted in green, while `x = x + 2;` is highlighted in purple, indicating they are part of the selected criteria.

Multiple statements

Kaveri Dependence Navigation

We have a bunch of “green stuff” – how did it get there?

```
switch(nRandVal) {  
    case 0:  
        // The result is 1  
        while(_ctr <= nRandVal) {  
            _result = _result + 1;  
            _ctr++;  
        }  
        break;  
    case 1:  
        // The result is 4  
        while(_ctr <= nRandVal) {  
            _result = _result + 2;  
            _ctr++;  
        }  
        break;  
}
```

Data Dependence

Control Dependence

Kaveri provides support for filtering and navigating dependence graphs

Even simple code can have a lot of dependencies

Dependence Tracking View

```
switch(nRandVal) {  
    case 0:  
        // The result is 1  
        → while(_ctr <= nRandVal) { [ ]  
            |→ result = _result + 1;  
            |→ _ctr++; [ ]  
        }  
        break;  
    case 1:  
        // The result is 4  
        → while(_ctr <= nRandVal) { [ ]  
            |→ result = _result + 2;  
            |→ _ctr++; [ ]  
        }  
        break;
```

Control Dependence

Data Dependence

S Dependence Tracking View X Jimple View Dependence History View

Statement	Dependence
[_ctr++; (Example1.java) ... i2 = i2 + 1]	[Dependees Control Data Dependents Control Data]

Demo

Dependence History View

```
switch(nRandVal) {  
    case 0:  
        // The result is 1  
        while(_ctr <= nRandVal) {  
            ② _result = _result + 1;  
            _ctr++;  
        }  
        break;  
}
```

③

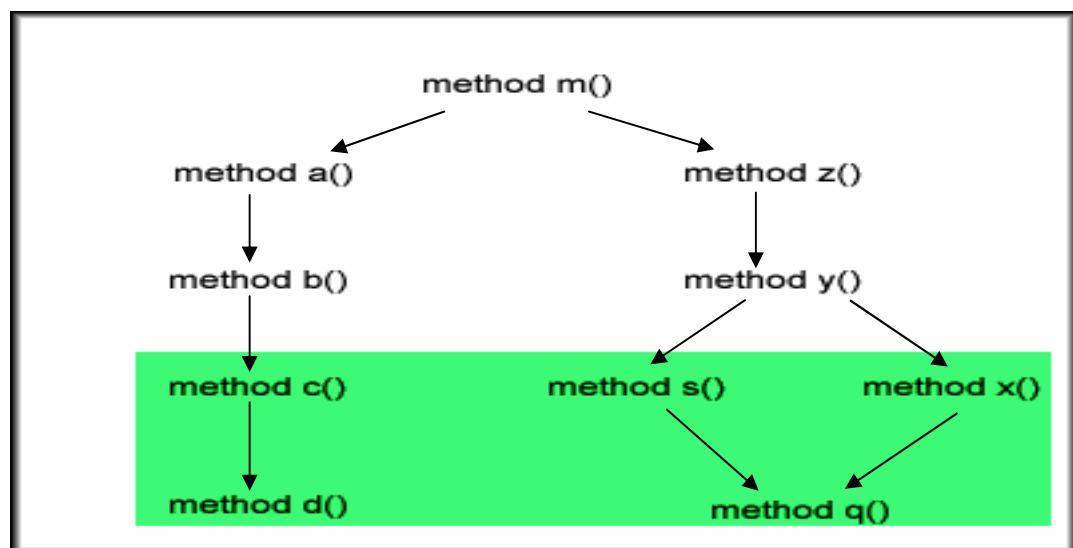
Statement	Filename	Line number	Relation with previous if
while(_ctr <= nRandVal) {	Example1.java	35	Data Dependent
_ctr++;	Example1.java	37	Control Dependent
while(_ctr <= nRandVal) {	Example1.java	35	Control Dependent
switch(nRandVal) {	Example1.java	32	Starting Program Point

- Maintains a history of program points visited through dependence tracking
- Navigate to earlier points in the path

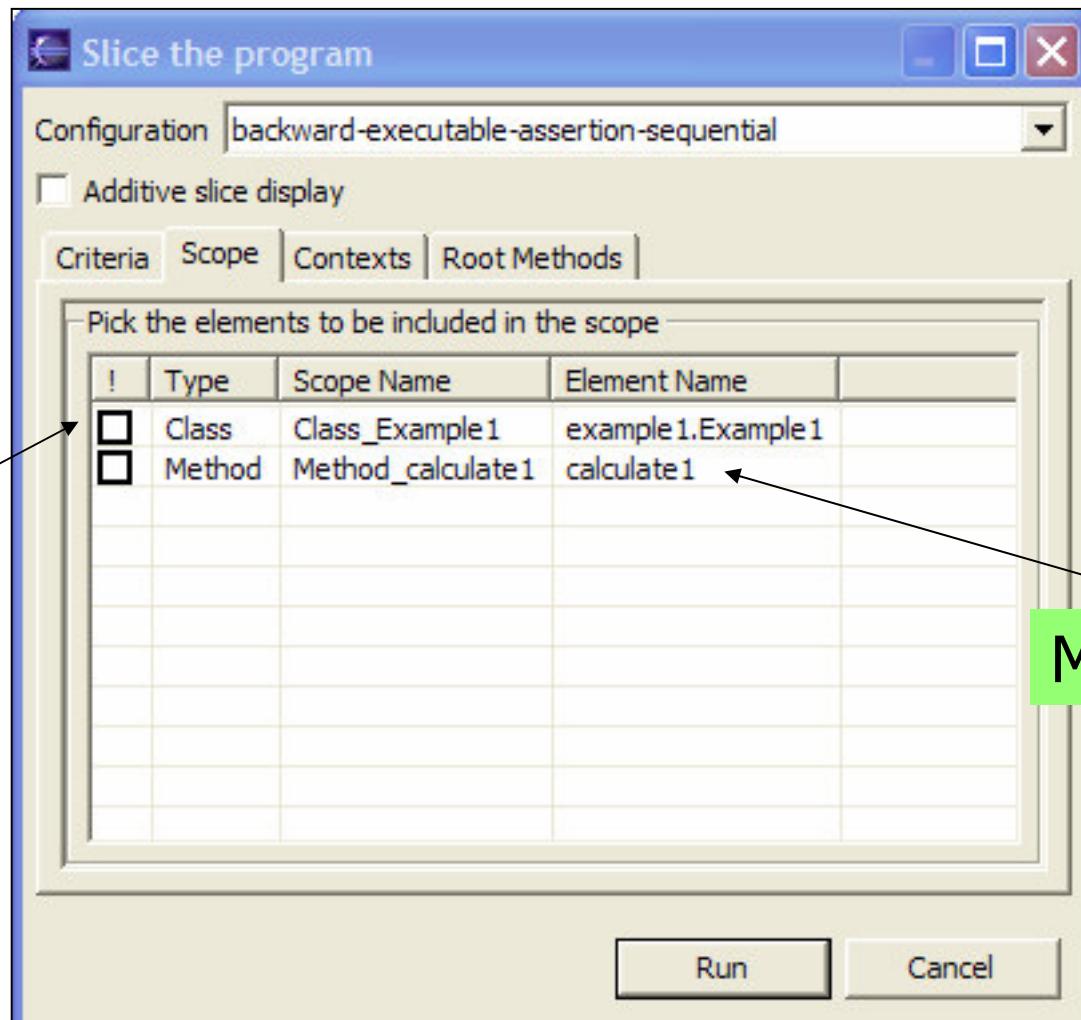
Demo

Scoped Slicing

- Scoped slicing allows the user to restrict entities that are considered for the slice
- Entities that can be added to the scope include Classes and Methods



Scoped Slicing



Class Scope

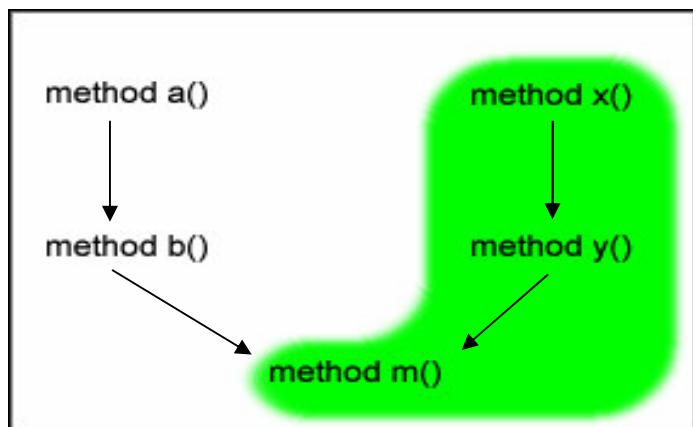
Method Scope

Demo

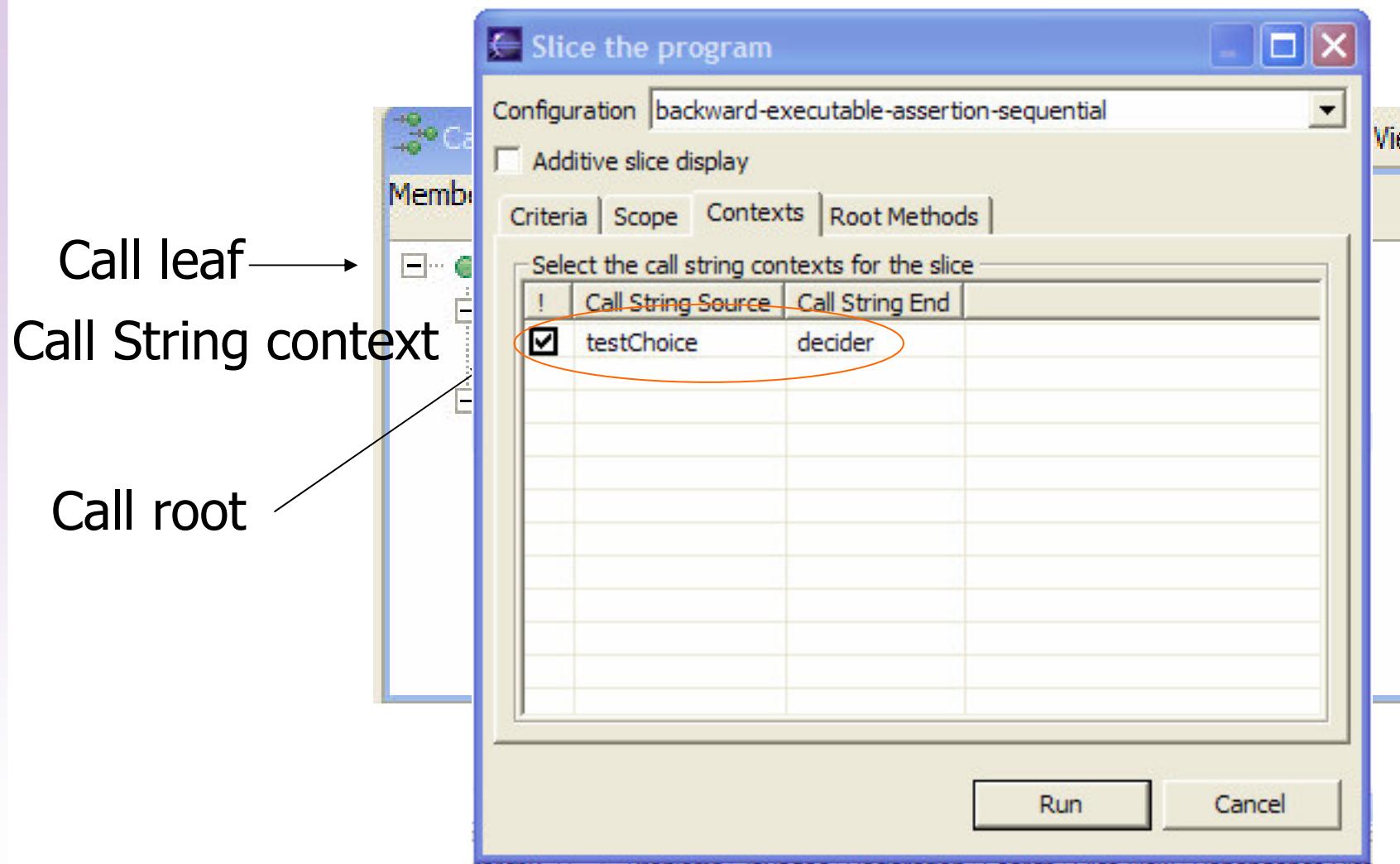
Call graph context-bounded slice

Imagine that your program has just crashed and you have a stack dump showing the call chain that lead to the error

- You could slice at the error point to help you identify the coding flaw
 - ...but that will show dependence throughout the program (including other call paths that arrived at the error point).
- Indus/Kaveri provides context-bounded slicing



Context-bounded slice



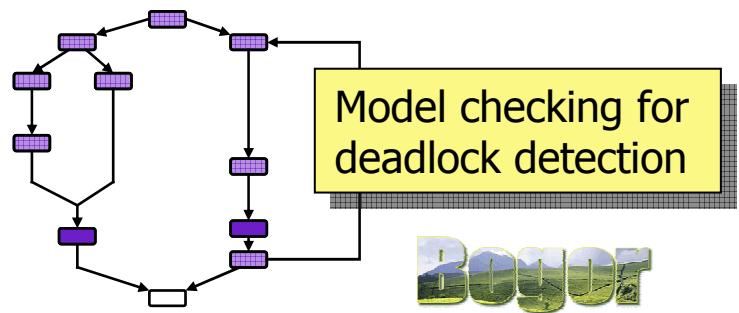
Demo

Outline

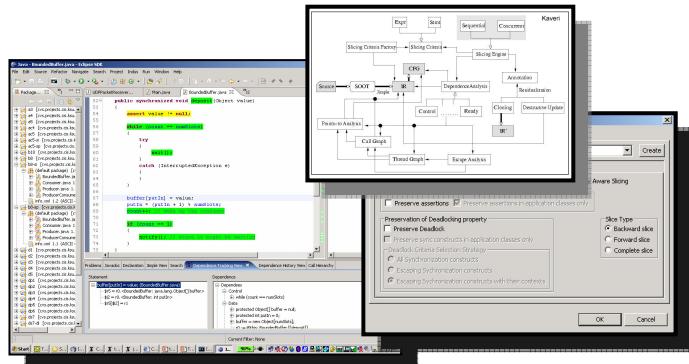
I. Program Dependences for Java

```
public synchronized void deposit(Object value) {  
    while (count == numSlots) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
    buffer[putIn] = value;  
    putIn = (putIn + 1) % numSlots;  
    count++;  
    if (count == 1)  
        notify();  
}  
  
public synchronized Object fetch() {  
    Object value;  
    while (count == 0) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
    value = buffer[takeOut];  
    takeOut = (takeOut + 1) % numSlots;  
    count--;  
    if (count == (numSlots - 1))  
        notify();  
    return value;  
}
```

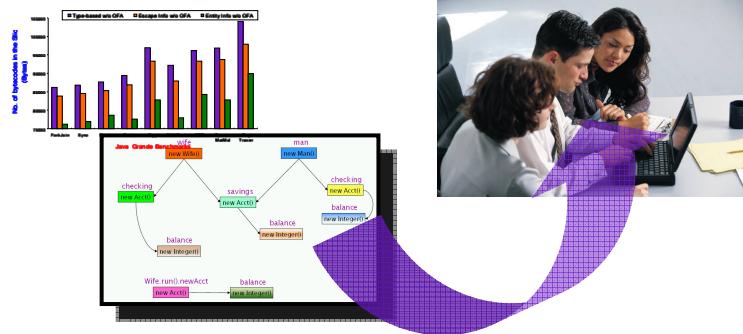
III. Application to Verifying Concurrent Java Programs



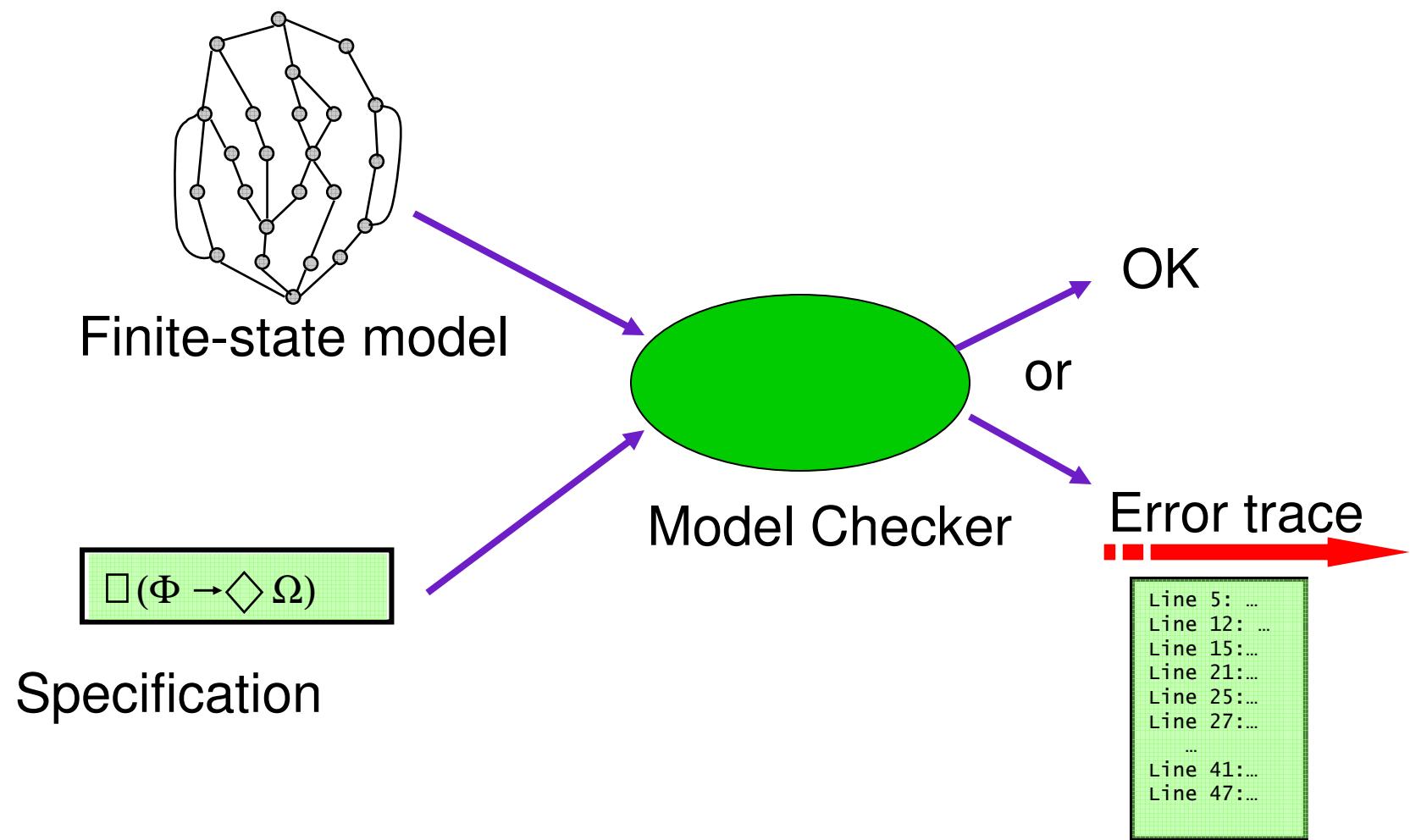
II. Indus & Kaveri



IV. Analysis optimization and mining analysis results



Model Checking



Why Try to Use Model Checking for Software?

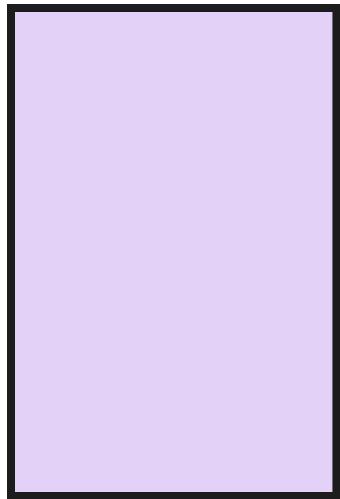
- Automatically check, e.g.,
 - invariants, simple safety & liveness properties
 - absence of dead-lock and live-lock,
 - complex event sequencing properties,
- In contrast to testing, gives complete coverage – even in presence of concurrency -- by exhaustively exploring all paths in system,
- It's been used for years with good success in hardware and protocol design

“Between the window open and the window close, button X can be pushed at most twice.”

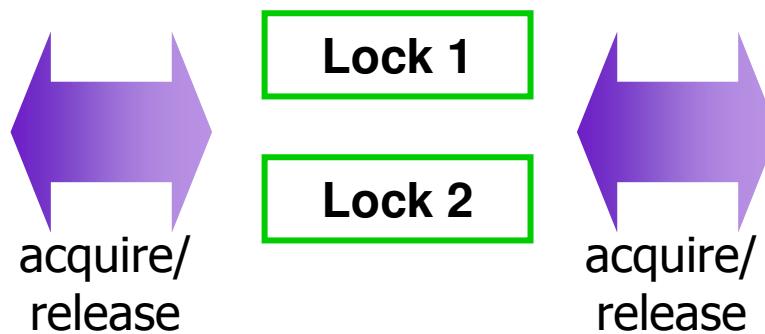
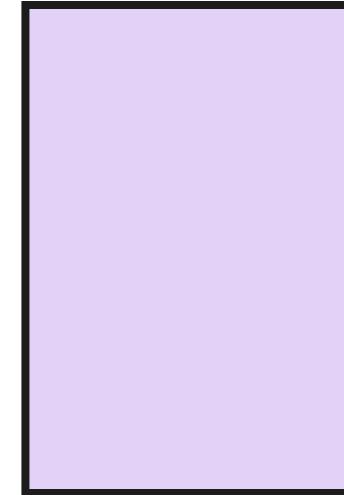
This suggests that model-checking can complement existing software quality assurance techniques.

Simple Deadlock Example

Process 1



Process 2



Simple Deadlock Example

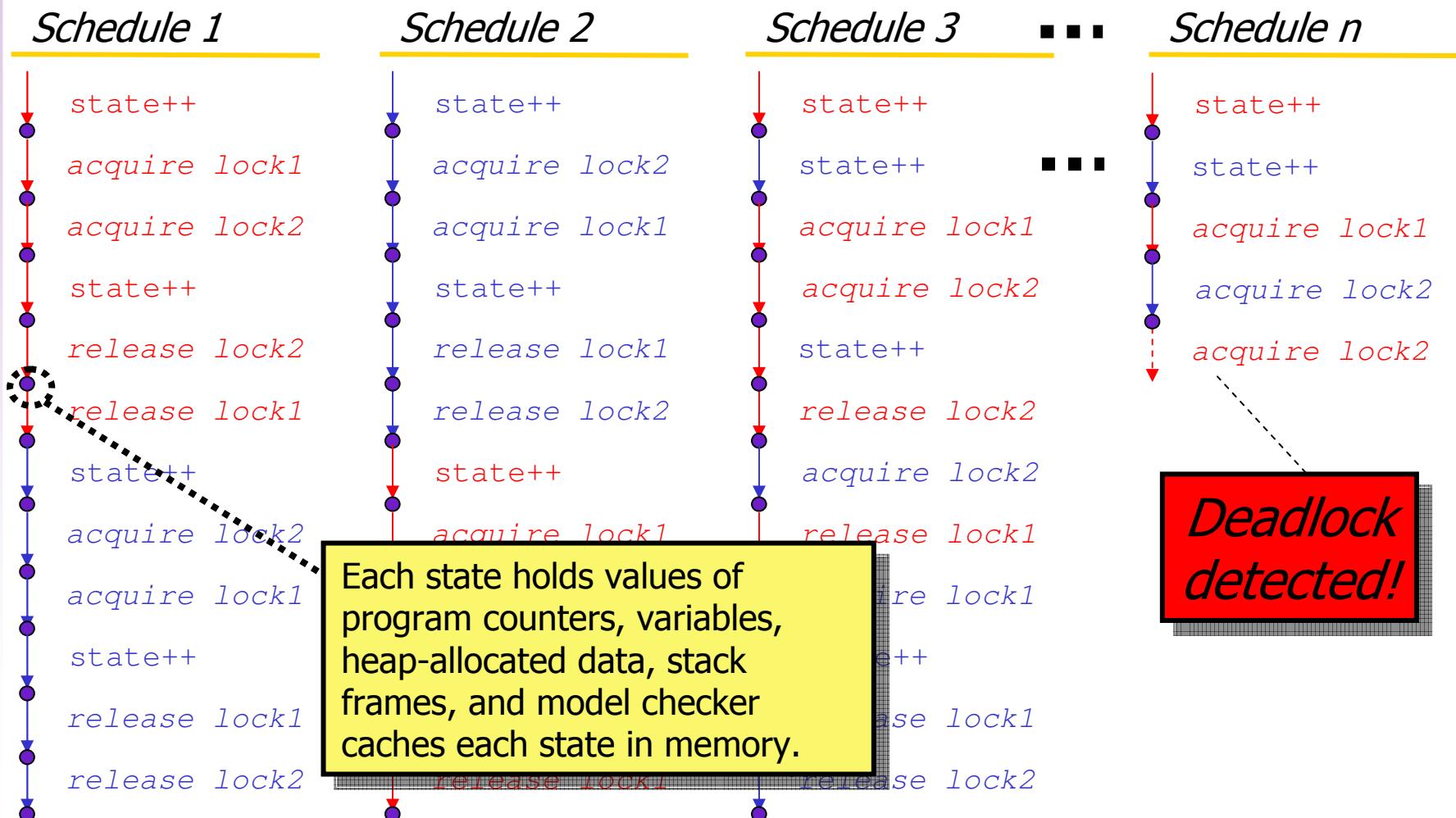
```
public class Deadlock {  
    static Lock lock1;  
    static Lock lock2;  
    static int state;  
  
    public static  
    void main(String[] args) {  
        lock1 = new Lock();  
        lock2 = new Lock();  
        Process1 p1  
            = new Process1();  
        Process2 p2  
            = new Process2();  
        p1.start();  
        p2.start();  
    }  
  
    class Lock {}
```

```
class Process1 extends Thread {  
    public void run() {  
        Deadlock.state++;  
        synchronized (Deadlock.lock1) {  
            synchronized (Deadlock.lock2) {  
                Deadlock.state++;  
            }  
        }  
    }  
  
    class Process2 extends Thread {  
        public void run() {  
            Deadlock.state++;  
            synchronized (Deadlock.lock2) {  
                synchronized (Deadlock.lock1) {  
                    Deadlock.state++;  
                }  
            }  
        }  
    }
```

Simple Deadlock Example

Process 1

Model checking explores all schedules/interleavings Process 2



Model Checking Realistic Systems

Goal

model check **implementations** of **realistic Java systems** against a particular property or collection of properties



Complexities leading to state explosion

- concurrency, heap intensive data, extensive use of layering, design patterns, etc.

- *...there's just a lot of code!*
 - libraries, infrastructure code, etc.

Slicing for Model Reduction

Folklore: program slicing is useful for model-reduction

Basic Thesis:

“...often, significant amounts of the program are irrelevant wrt (independent of) the property being checked...”

Observation:

Program slicing is a static analysis technique for...

- reasoning about program (in)dependences
- eliminating program statements that are irrelevant to computations at particular program points.



Idea:

Use program slicing to prune away parts of the system that are irrelevant wrt property being checked.

Properties and Program Features

Property:

- absence of deadlock

Program features to drive slicing (slicing criterion):

- synchronized statements
(lock acquires/releases)

```
class Process1 extends Thread {  
    public void run() {  
        Deadlock.state++;  
        synchronized (Deadlock.lock1) {  
            synchronized (Deadlock.lock2) {  
                Deadlock.state++;  
            }}}} }  
  
class Process2 extends Thread {  
    public void run() {  
        Deadlock.state++;  
        synchronized (Deadlock.lock2) {  
            synchronized (Deadlock.lock1) {  
                Deadlock.state++;  
            }}}} }
```

*Slicing
criteria*

Slicing action:

- finds all program statements that influence the computation of the synchronized statements (removes the rest)

Sliced Deadlock Example

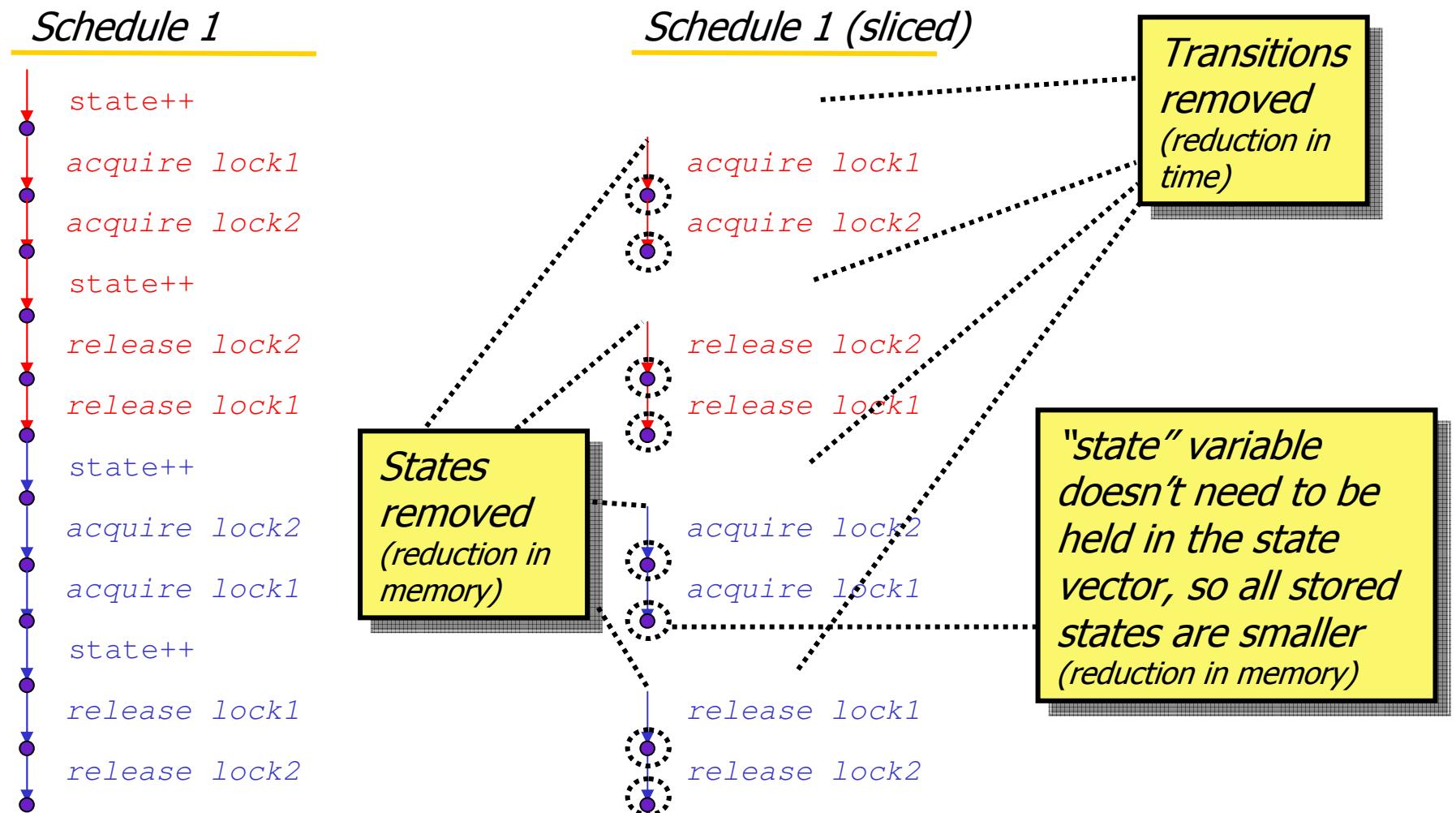
```
public class Deadlock {  
    static Lock lock1; / Variable sliced away  
    static Lock lock2; /  
    static int state; /  
  
    public static void main(String[] args) {  
        lock1 = new Lock();  
        lock2 = new Lock();  
        Process1 p1  
            = new Process1();  
        Process2 p2  
            = new Process2();  
        p1.start();  
        p2.start();  
    }  
  
    class Lock {}
```

```
class Process1 extends Thread {  
    public void run() { / statements (transitions) sliced away  
        Deadlock.state++;  
        synchronized (Deadlock.lock1) {  
            synchronized (Deadlock.lock2) {  
                Deadlock.state++;  
            }}}} }  
  
class Process2 extends Thread {  
    public void run() {  
        Deadlock.state++;  
        synchronized (Deadlock.lock2) {  
            synchronized (Deadlock.lock1) {  
                Deadlock.state++;  
            }}}} }
```

Green stuff = "in the slice"

Simple Deadlock Example

Effects of slicing on model checking costs



Expected Benefits of Slicing

Statements Removed

- fewer transitions
- fewer interleavings
- fewer states

Unreachable Methods/Classes Removed

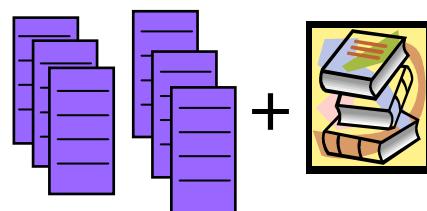
- static initializers removed
- reduces preprocessing / translation overhead

Variables Removed

- smaller state vector

Experiments

I. Code Bases



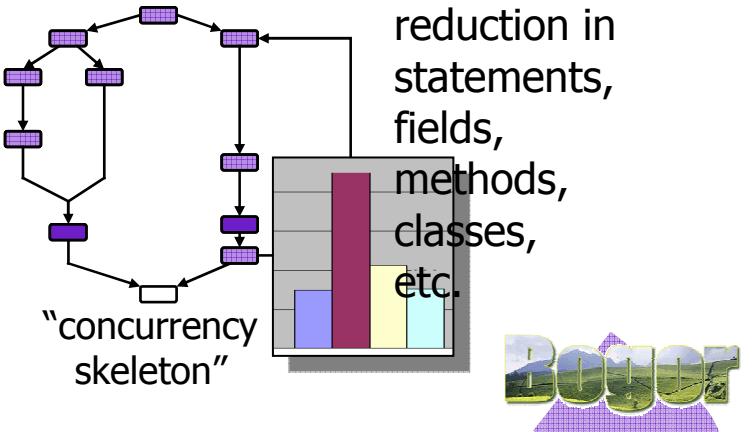
Java
application
code

+
Source version
of Java
libraries

*slice on
synchronization
statements for
deadlock checking*

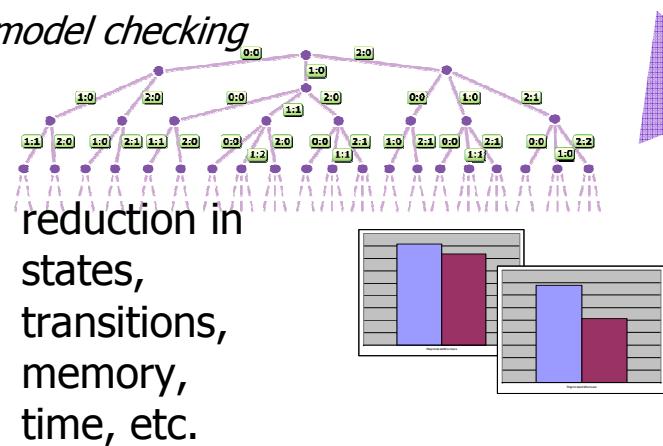
Indus

II. Slicing – Static Metrics



III. Slicing – Dynamic Metrics

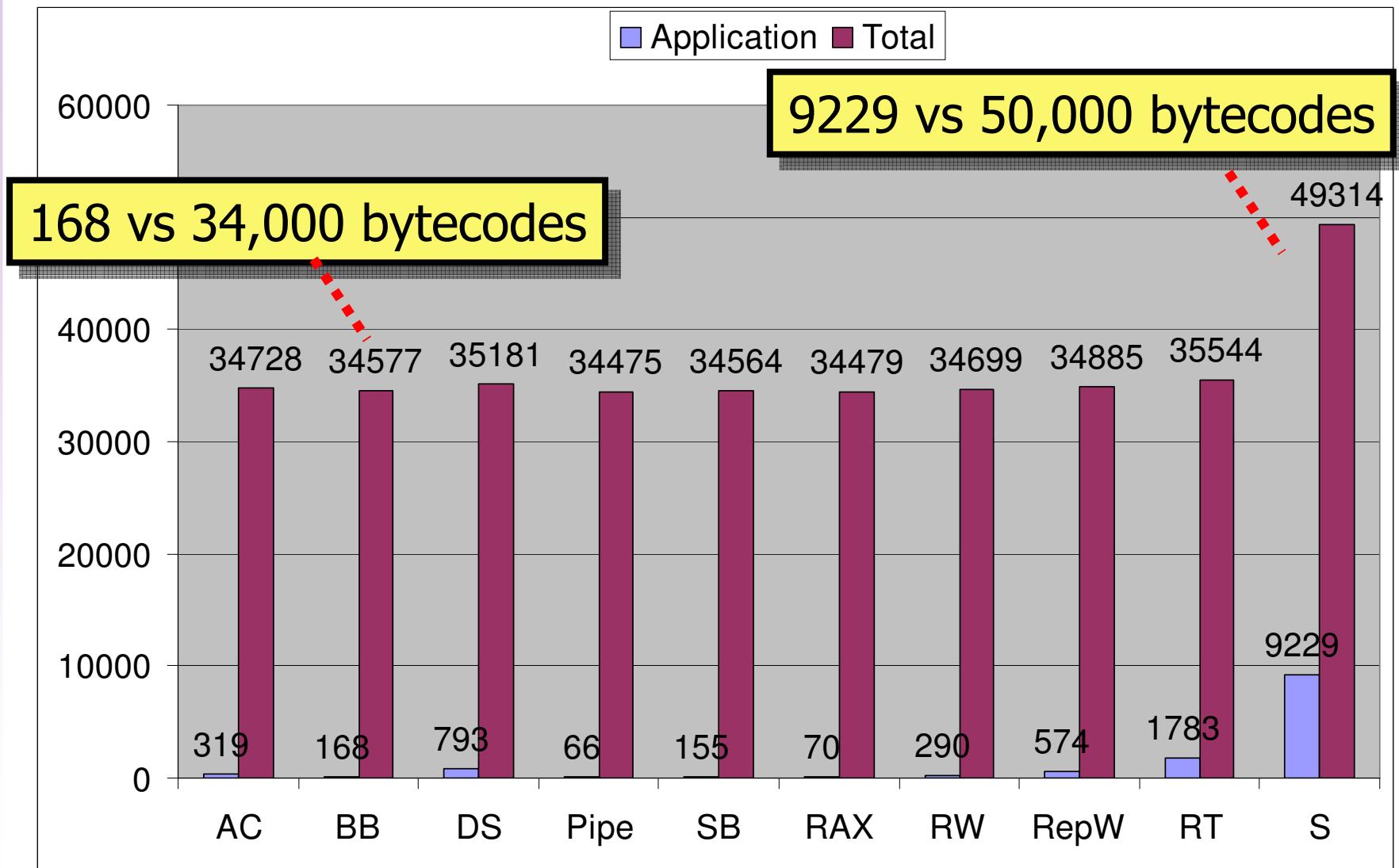
model checking



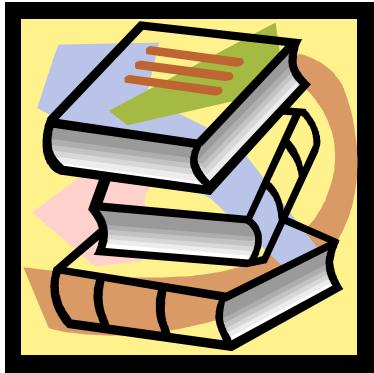
Classes of Examples

- Algorithm Sketches (used elsewhere in the literature)
 - Bounded Buffer
 - Pipeline
 - Sleeping Barbers
 - Readers/Writers
 - RAX (distillation of NASA Remote Agent Bug)
- Small Applications (significantly more heap intensive)
 - Disk Scheduler
 - Alarm Clock
- Larger Applications
 - Replicated Workers (client/server data distribution framework)
 - Ray Tracer (Java Grande)
 - Siena (internet-scale publish/subscribe middleware)

Size in Bytecodes



Libraries -- Assessment



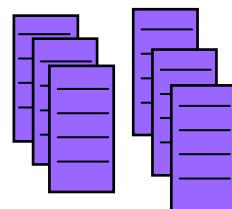
- Significant portion of program bytecodes come from libraries
- Some library code may be *unreachable during execution* but still contribute to state vector
- Some library code may be reachable but *irrelevant to property*
- Much unreachable code can be pruned away using a precise call-graph analysis (based on precise “points-to” analysis)
 - To avoid “straw man” arguments, we will take CallGraph-reachability-pruned code-bases as the baseline of slicing/model-checking experiments

Comparing slicing to an already “smart” framework!

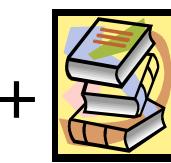
Experiments



I. Code Bases



Java
application
code

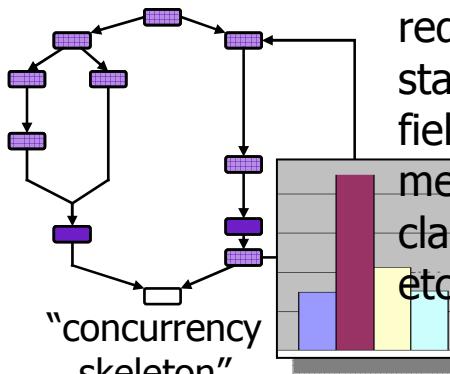


Source version
of Java
libraries

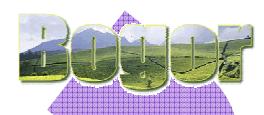
*slice on
synchronization
statements for
deadlock checking*

Indus

II. Slicing – Static Metrics

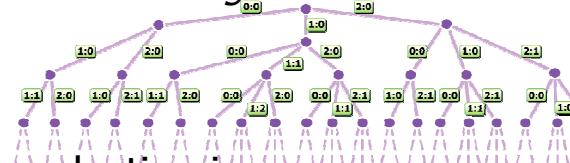


reduction in
statements,
fields,
methods,
classes,
etc.

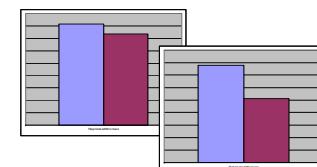


III. Slicing – Dynamic Metrics

model checking



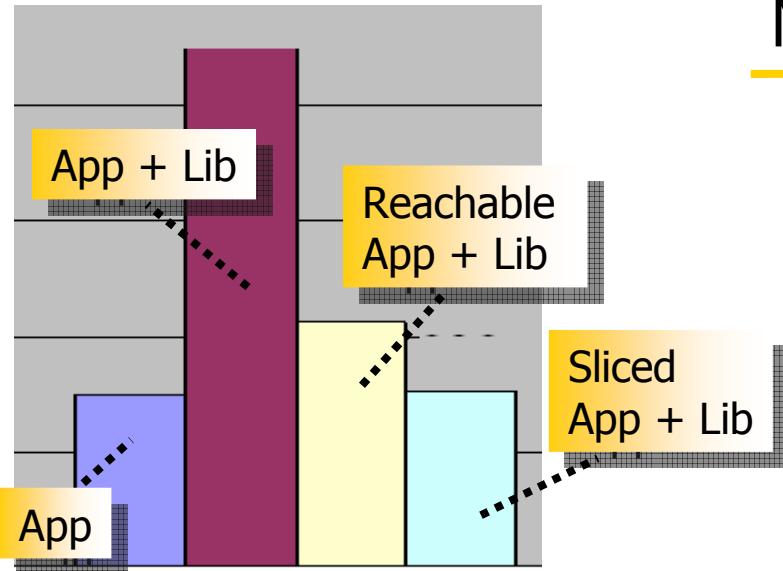
reduction in
states,
transitions,
memory,
time, etc.



Static Metrics for Slicing



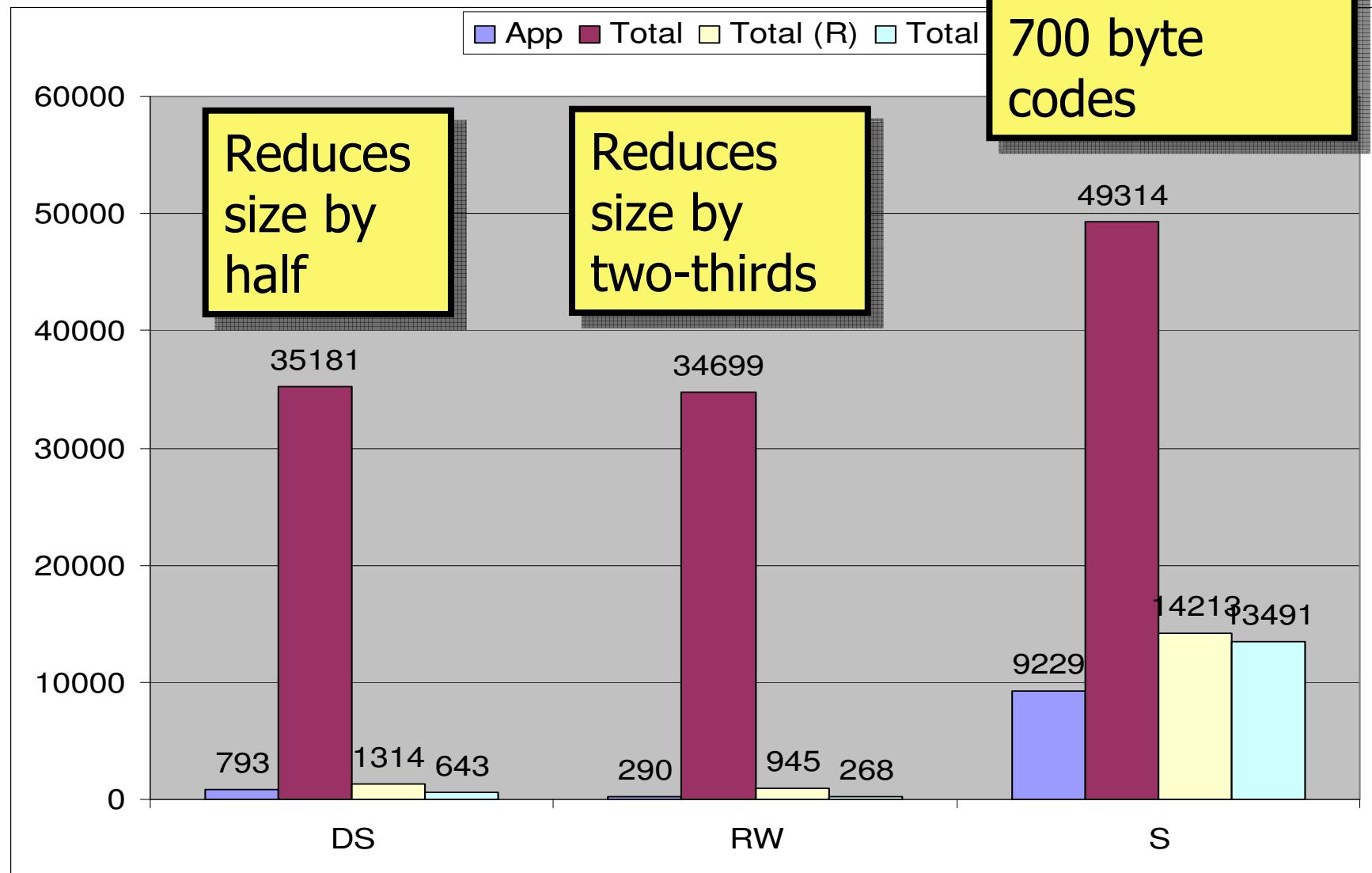
What is the effect of Call-Graph-reachability and slicing on the static program model structure?



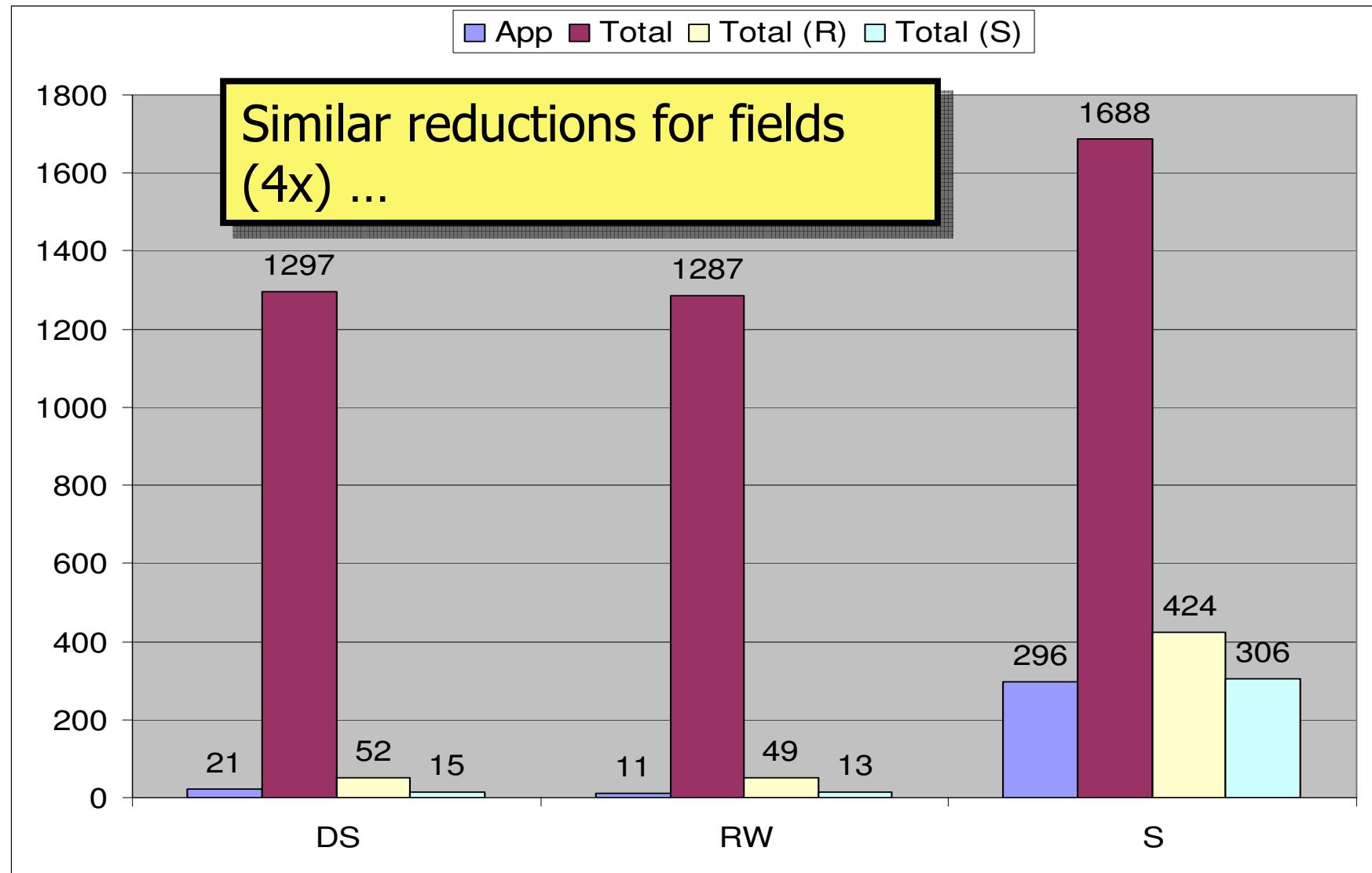
Measurements (see paper)

- bytecodes
- classes
- methods
- fields
- new (allocations)
- threads
- exceptions
- synchronization statements
- wait
- notify

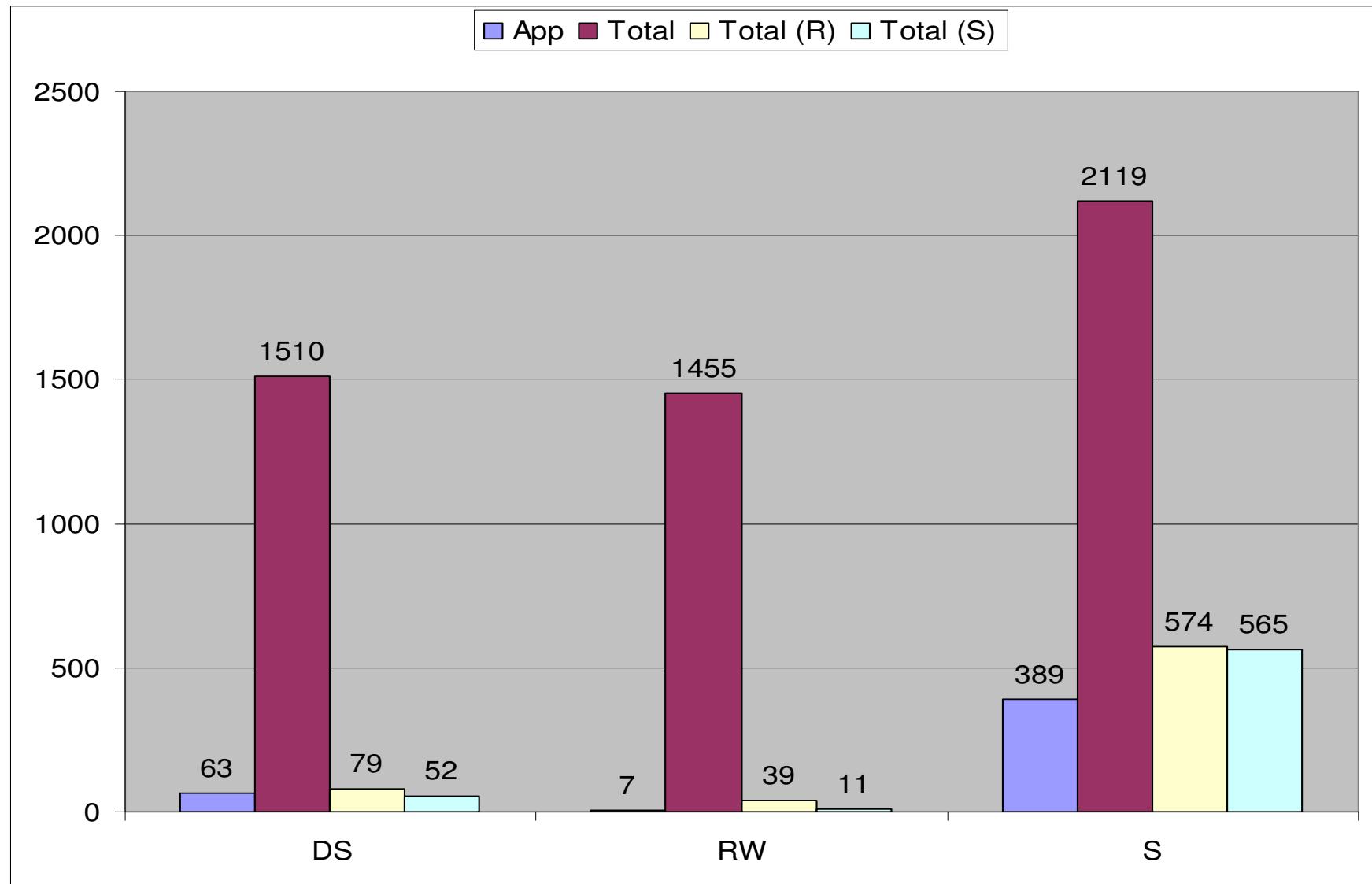
Static Metrics - Byte-codes



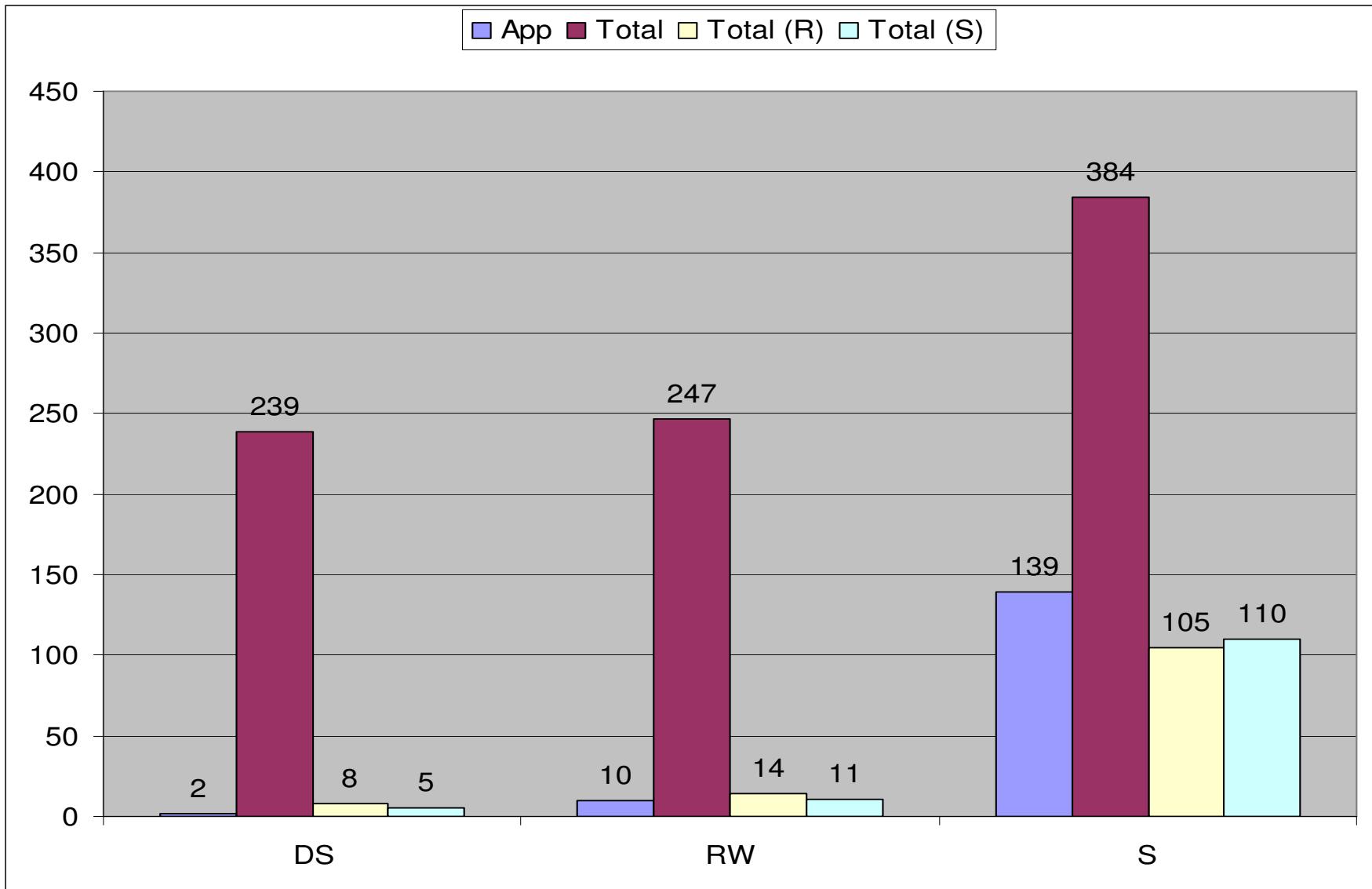
Static Metrics - Fields



Static Metrics - New

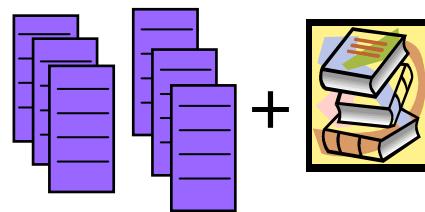


Static Metrics - Sync



Experiments

I. Code Bases



Java
application
code

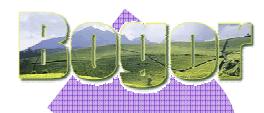
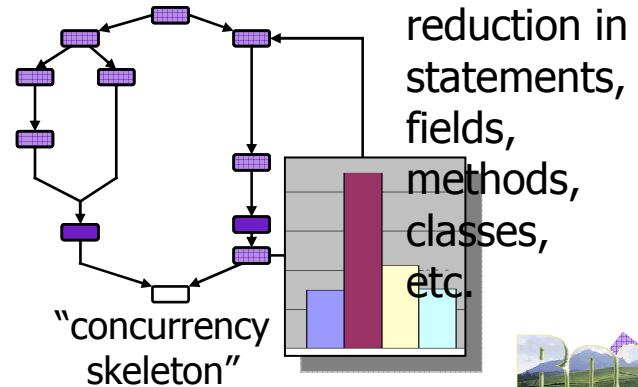


Source version
of Java
libraries

*slice on
synchronization
statements for
deadlock checking*

Indus

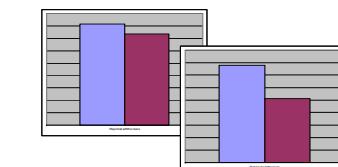
II. Slicing – Static Metrics



III. Slicing – Dynamic Metrics

model checking

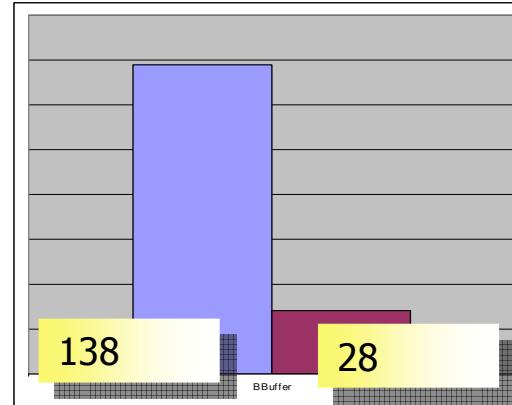
reduction in
states,
transitions,
memory,
time, etc.



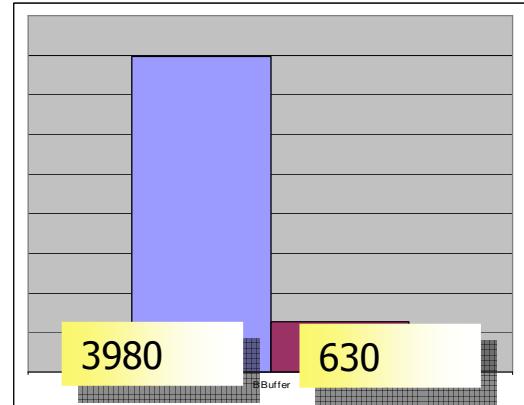
Dynamic Metrics - BBuffer w/POR

Between factor
of 4-5 reduction

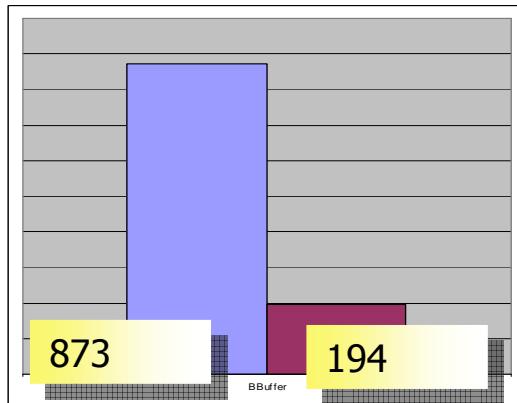
States



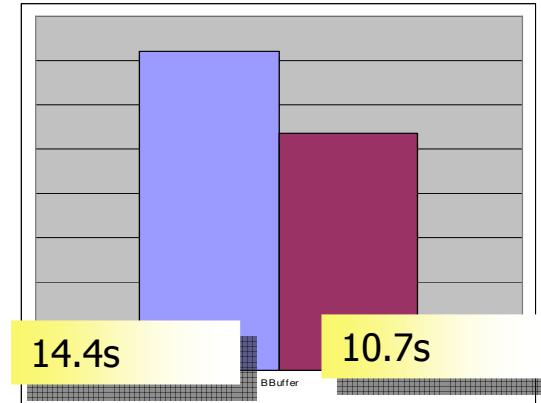
Transitions



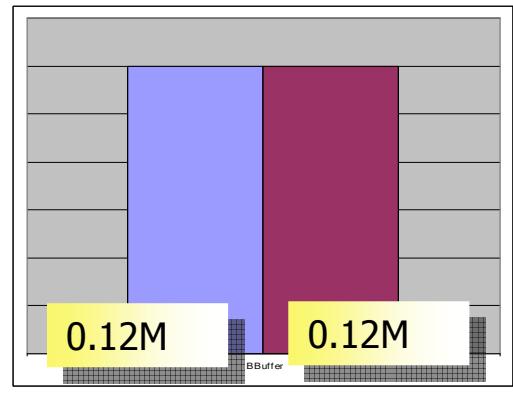
Byte-codes



Time



Memory



Java - BoundedBuffer.java - Eclipse SDK

File Edit Source Refactor Navigate Search Project Indus Run Window Help

Package Explorer UDPPacketReceiver... DatagramPacket.java BoundedBuffer.java Outline

```
1 class BoundedBuffer
2 { // designed for a single producer thread
3     // and a single consumer thread
4
5     //~ Instance variables .....
6
7     protected int numSlots = 0;
8     protected Object[] buffer = null;
9     protected int putIn = 0;
10    protected int takeOut = 0;
11    protected int count = 0;
12
13    //~ Constructors .....
14
15    /*@ invariant count <= numSlots;
16     * @ invariant count >= 0;
17     * @ invariant buffer.length == numSlots;
18     */
19
20    /*@ atomic behavior
21     * @ requires numSlots > 0;
22     * @ assignable this.numSlots, this.buffer, this.putIn, t
23     * @ ensures \fresh(buffer) && buffer.length == numSlots ;
24     * @also
25     */
26
27    public BoundedBuffer(int numSlots)
28    {
29        this.numSlots = numSlots;
30        this.buffer = new Object[numSlots];
31        this.putIn = 0;
32        this.takeOut = 0;
33        this.count = 0;
34    }
35
36    public void deposit(Object object)
37    {
38        synchronized(this)
39        {
40            if(count < numSlots)
41            {
42                buffer[putIn] = object;
43                putIn++;
44                count++;
45            }
46        }
47    }
48
49    public Object fetch()
50    {
51        synchronized(this)
52        {
53            if(count > 0)
54            {
55                Object object = buffer[takeOut];
56                takeOut++;
57                count--;
58                return object;
59            }
60        }
61    }
62
63    public void clear()
64    {
65        synchronized(this)
66        {
67            putIn = 0;
68            takeOut = 0;
69            count = 0;
70        }
71    }
72
73    public int getPutIndex()
74    {
75        return putIn;
76    }
77
78    public int getTakeIndex()
79    {
80        return takeOut;
81    }
82
83    public int getCount()
84    {
85        return count;
86    }
87
88    public void setCount(int count)
89    {
90        this.count = count;
91    }
92
93    public void incrementCount()
94    {
95        this.count++;
96    }
97
98    public void decrementCount()
99    {
100        this.count--;
101    }
102}
```

Problems Javadoc Declaration Jimple View Search Dependence Tracking View Dependence History View Call Hierarchy

Statement Dependence

Start TACAS06 Microsoft Po... johns-notes.t... Indus-0.8 Java - Boun... 96% 6:20 AM

Buffer and indices are sliced away – because synchronization only depends on **count**.

Java - BoundedBuffer.java - Eclipse SDK

File Edit Source Refactor Navigate Search Project Indus Run Window Help

Package Explorer UDPPacketReceiver... DatagramPacket.java BoundedBuffer.java Outline

```
public synchronized void deposit(Object value)
{
    assert value != null;

    while (count == numSlots)
    {
        try
        {
            wait();
        }
        catch (InterruptedException e)
        {
        }
    }

    buffer[putIn] = value;
    putIn = (putIn + 1) % numSlots;
    count++; // wake up the consumer

    if (count == 1)
    {
        notify(); // since it might be waiting
    }
}
```

Problems Javadoc Declaration Jimple View Search Dependence Tracking View Dependence History View Call Hierarchy

Statement Dependence

Writable Smart Insert 1 : 1

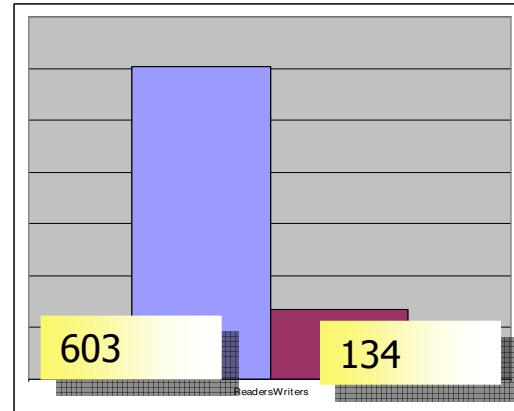
Start TACAS06 Microsoft Po... johns-notes.t... Indus-0.8 Java - Boun... 96% 6:23 AM

Buffer and indices are sliced away – because synchronization only depends on **count**.

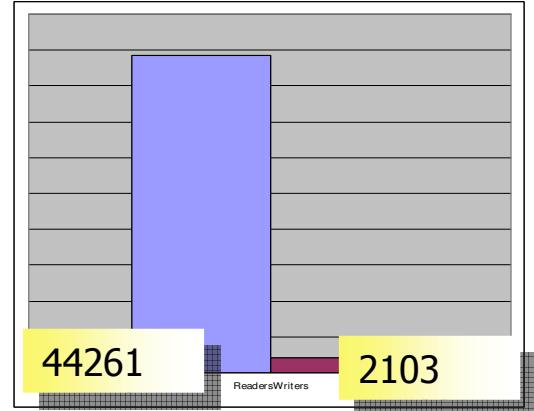
Dynamic Metrics - ReadersWriters w/POR

Significant reductions...

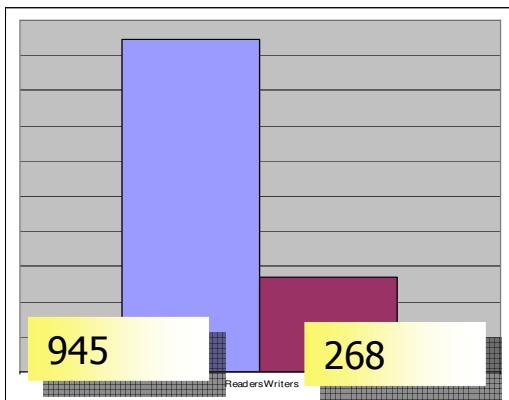
States



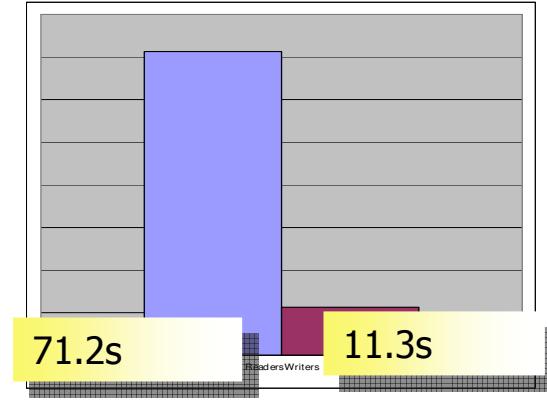
Transitions



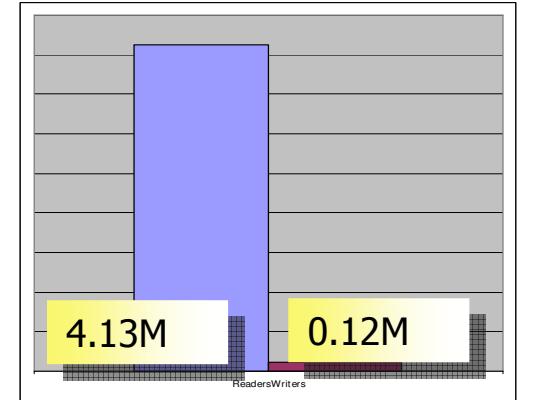
Byte-codes



Time



Memory



Assessment

- Very little application code sliced away (a counter, and `System.out.println`)
- Dramatic amount of library code is sliced away
 - 25 classes, 53 methods, 40 fields, and 683 statements
 - `java.lang.System` gets completely gutted

`java.util.Properties`
`java.io.PrintStream`
`java.io.InputStream`
`java.io.OutputStream`



`java.util.Hashtable`
`java.io.BufferedReader`
`java.io.InputStreamReader,`
`java.io.Reader`
`java.lang.StringBuffer`
`java.io.PrintWriter`
`java.util.Iterator`
...and much more

Slicing Siena Middleware

UDPPacketReceiver.java

```
...  
packet = new DatagramPacket(new byte[1], 1);  
...
```

Not in the slice. Why?

DatagramPacket.java

```
...  
public DatagramPacket(final byte[] b,  
                     final int length) {  
    setData(b, 0, length);  
}  
...
```

Passed to constructor,
and then to setData.

```
public void setData(final byte[] data,  
                   final int offset,  
                   final int length)  
{  
    buffer = data;  
    setLength(length);  
}
```

Passed to setData, and
then assigned to buffer
which is not read by any
statement in the slice!

Slicing Siena Middleware

UDPPacketReceiver.java

```
...  
packet = new DatagramPacket(new byte[1], 1);  
...
```

Not in the slice. Why?

DatagramPacket.java

```
...  
public DatagramPacket(final byte[] data,  
                     final int length)  
{  
    setData(b, 0, length);  
}  
...  
  
public void setData(final byte[] data,  
                   final int offset,  
                   final int length)  
{  
    buffer = data;  
    setLength(length);  
}
```

Summary:

Via several intermediate steps, new byte[1] is flowing across multiple methods and classes only into a variable that is never used in the slice.

Very difficult to detect manually – especially on large code bases!

Assessment

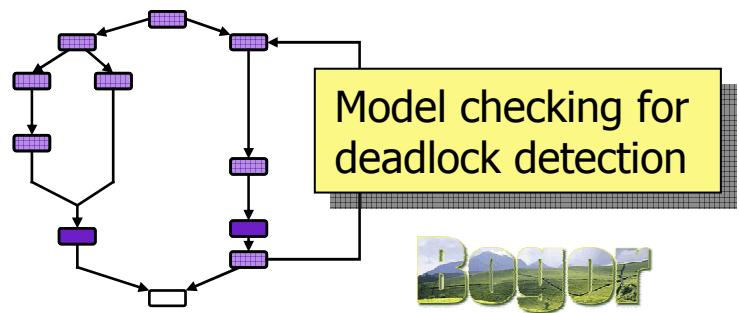
- Applying Indus slicing for Java model checking
 - is easy (completely automatic – no user intervention)
 - is inexpensive (compared to model checking costs)
 - is effective and orthogonal to other reduction strategies (typical reduction factors range from 2x – 5x)
- Key enabling technology: additional notions of dependence related to preserving blocking/deadlocking behavior
- Slicing techniques like Indus seem particularly relevant for real systems that make significant use of libraries

Outline

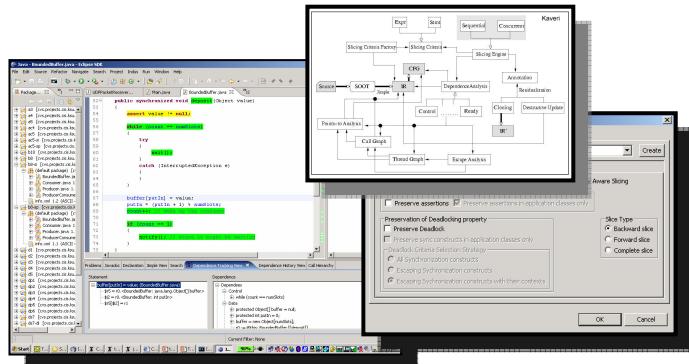
I. Program Dependences for Java

```
public synchronized void deposit(Object value) {  
    while (count == numSlots) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
    buffer[putIn] = value;  
    putIn = (putIn + 1) % numSlots;  
    count++;  
  
    if (count == 1)  
        notify();  
}  
}  
  
public synchronized Object fetch() {  
    Object value;  
  
    while (count == 0) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
  
    value = buffer[takeOut];  
    takeOut = (takeOut + 1) % numSlots;  
    count--;  
  
    if (count == (numSlots - 1))  
        notify();  
}  
}  
return value;
```

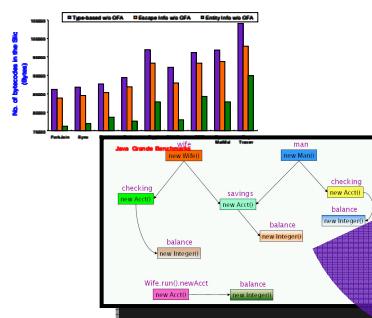
III. Application to Verifying Concurrent Java Programs



II. Indus & Kaveri

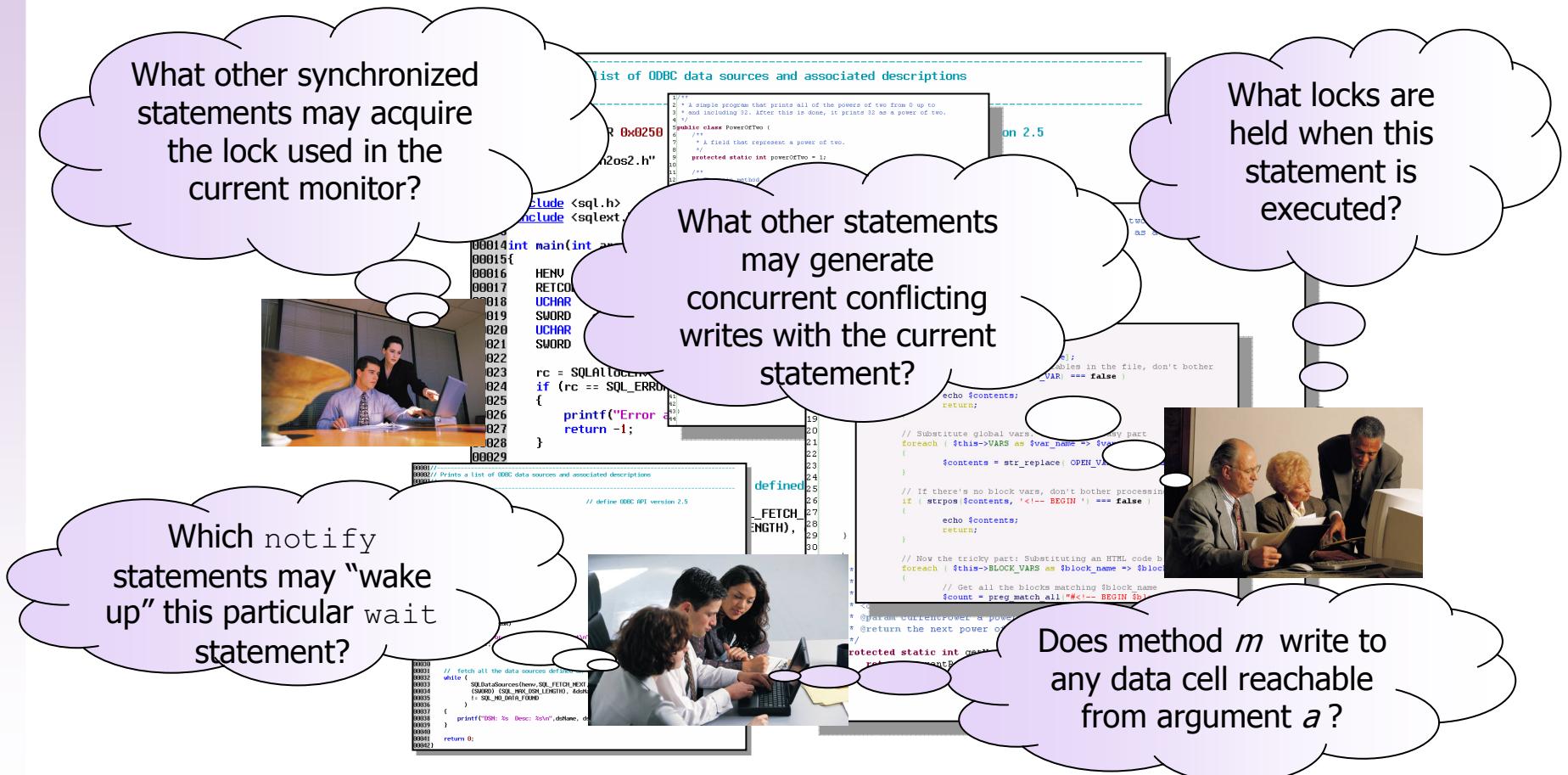


IV. Analysis optimization and mining analysis results

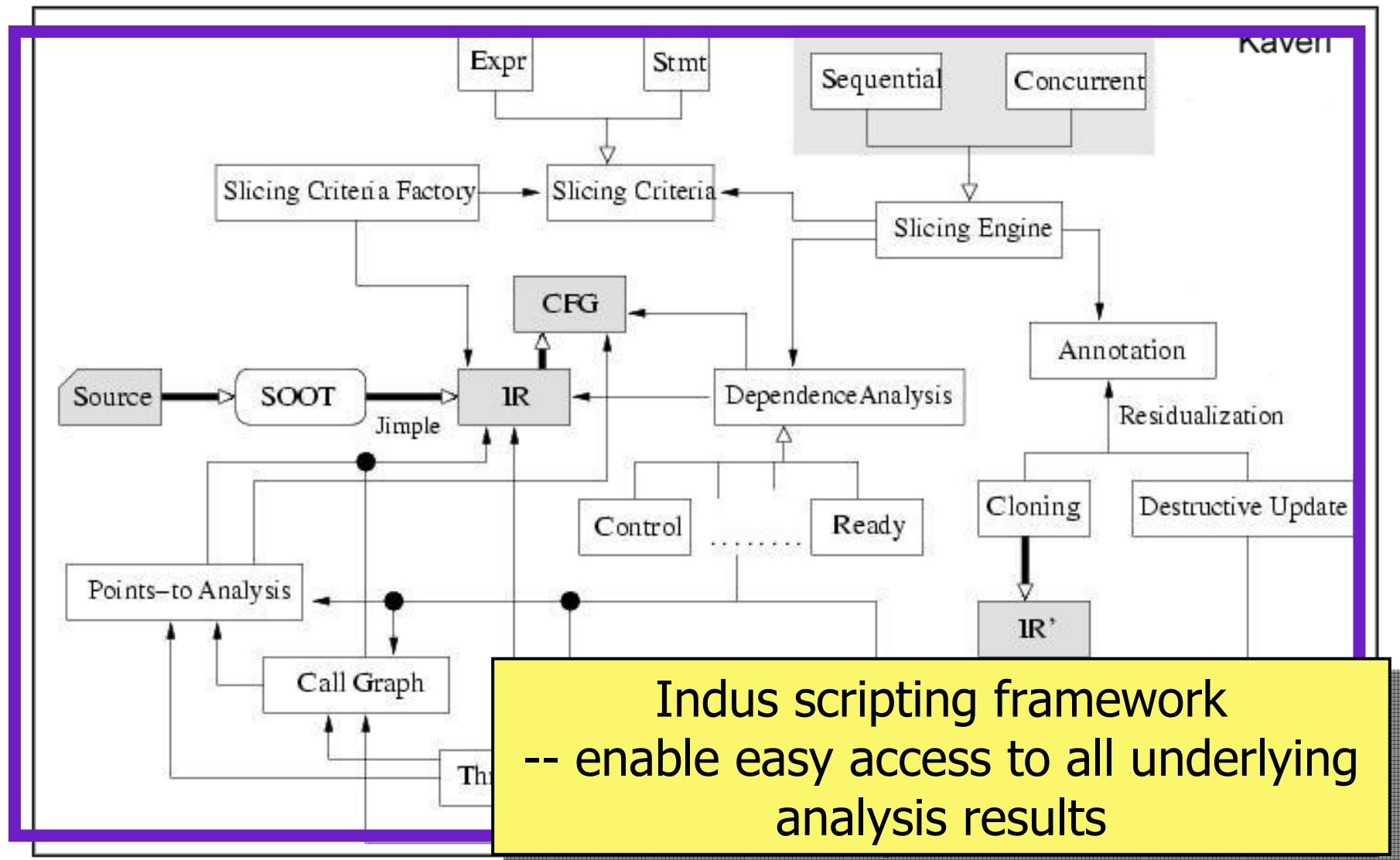


Beyond Traditional Slicing Questions

Mining the results of wide-ranging static analysis for concurrency



Indus / Kaveri Architecture



An Example

```
public class Home {
    public static void main(String[] s) {
        Acct savings = new Acct();
        Thread wife = null;
        Thread man = null;
        wife = new Wife(savings);
        husband = new Man(savings);
        wife.start();
        husband.start();
    }
}

class Acct {
    protected Integer balance;
    Acct() {
        balance = new Integer(100);
    }
}

class Wife extends Thread {
    protected Acct savings;
    protected Acct checking;
    Wife(Acct act) {
        this.savings = act;
        checking = new Acct();
    }
    public void run() {
        try {
            synchronized(savings) {
                savings.wait();
            }
            Acct newAcct = new Acct();
            savings.balance = new Integer(savings.balance.intValue() - 20);
            checking.balance = new Integer(checking.balance.intValue() - 10);
            newAcct.balance = new Integer(newAcct.balance.intValue() + 10);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

class Husband extends Thread {
    protected Acct savings;
    protected Acct checking;
    Husband (Acct act) {
        this.savings = act;
        checking = new Acct();
    }
    public void run() {
        savings.balance = new Integer(savings.balance.intValue() + 20);
        synchronized(savings) {
            savings.notify();
        }
        checking.balance = new Integer(checking.balance.intValue() + 1);
        synchronized(checking) {
            checking.notify();
        }
    }
}
```

An Example

```
class Acct {  
    protected Integer balance;  
    Acct0 {  
        balance = new Integer(100);  
    }  
}  
  
class Wife extends Acct {  
    protected Acct savings;  
    protected Acct checking;  
    Wife(Acct act) {  
        this.savings = act;  
        checking = new Acct();  
    }  
    public void run() {  
        try {  
            synchronized(savings) {  
                savings.wait();  
            }  
            Acct newAcct = new Acct();  
            savings.balance = new Integer(savings.balance.intValue() - 20);  
            checking.balance = new Integer(checking.balance.intValue() - 10);  
            newAcct.balance = new Integer(newAcct.balance.intValue() + 10);  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Objects of account type are shared between threads.

```
public class Home {  
    public static void main(String[] s) {  
        new Acct();  
        ...  
        Man man = new Man(savings);  
        wife.start();  
        man.start();  
    }  
}
```

```
man extends Thread {  
    protected Acct savings;  
    protected Acct checking;  
    Acct act) {  
        this.savings = act;  
        checking = new Acct();  
    }  
}
```

Access to balance may induce inter-thread data interferences.

```
    checking.balance = new Integer(checking.balance.intValue() + 10);  
    synchronized(checking) {  
        checking.notify();  
    }  
}
```

An Example

```
class Acct {  
    protected Integer balance;  
  
    savings account is shared  
    between a man and wife.  
}
```

```
public class Home {  
    public static void main(String[] s) {  
        Acct savings = new Acct();  
        Thread wife = null;  
        Thread husband = null;  
        wife = new Wife(savings);  
        man = new Husband(savings);  
        wife.start();  
        husband.start();  
    }  
}  
  
this.savings = act;  
checking = new Acct();  
}  
public void run() {  
    savings.balance = new Integer(savings.balance.intValue() - 20);  
    checking.balance = new Integer(checking.balance.intValue() - 10);  
    newAcct.balance = new Integer(newAcct.balance.intValue() + 10);  
    synchronized(savings) {  
        savings.notify();  
    }  
    synchronized(checking) {  
        checking.notify();  
    }  
}
```

An Example

```
class Wife extends Thread {  
    protected Acct savings;  
    protected Acct checking;  
    Wife(Acct act) {  
        this.savings = act;  
        checking = new Acct();  
    }  
    public void run() {  
        try {  
            synchronized(savings) {  
                savings.wait();  
            }  
            Acct newAcct = new Acct();  
            savings.balance = new Integer(savings.balance.intValue() - 20);  
            checking.balance = new Integer(checking.balance.intValue() - 10);  
            newAcct.balance = new Integer(newAcct.balance.intValue() + 10);  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Assign the shared savings account in an instance field

Wife has a personal checking account.

Wife waits for a transaction on the savings account

Wife saves some money in a private local account, newAcct.

An Example

```
class Acct {  
    protected Integer balance;  
    Acct() {  
        ba...  
    }  
}  
  
class Wife extends Thread {  
    protected Acct savings;  
    protected Acct checking;  
    Wife(Acct act) {  
        this.savings = act;  
        checking = new Acct();  
    }  
    public void run0 {  
        try {  
            synchronized(sav...  
            savings.wait();  
        }  
        Acct newAcct = ne...  
        savings.balance =  
        checking.balance =  
        newAcct.balance =  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}  
  
class Man extends Thread {  
    protected Acct savings;  
    protected Acct checking;  
    Man(Acct act) {  
        this.savings = act;  
        checking = new Acct();  
    }  
    public void run() {  
        savings.balance = new Integer(savings.balance.intValue() + 20);  
        synchronized(savings) {  
            savings.notify();  
        }  
        checking.balance = new Integer(...  
        synchronized(checking) {  
            checking.notify();  
        }  
    }  
}  
  
public class Home {  
    public static void main(String[] s) {  
        Mas has a personal  
        checking account.  
  
        Man deposits money into the  
        savings account and  
        notifies others about it.  
  
        Man deposits money into the  
        checking account and  
        notifies others about it.  
        But it is a personal  
        account, so the effort is  
        wasted!!  
    }  
}
```

Sociology Issues



Husband using bank account to bet on football



For the good of the family, wife is day-trading on the side and collecting profits in private account

Example Queries

Using escape analysis

- Show all concurrent writes that conflict
- Show other synchronized statements that may be contending for the same lock
- Show all synchronized states encountered along call paths to this line
- At wait, show influencing notifies
- At notify, show dependent waits
- etc.

Example Queries

Using side-effects analysis

- Does method m write to any field reachable from argument a ?
- Does method m read any fields reachable from argument a ?
- Does method m depend on the any static fields?

Conclusions

- Indus provides a wide-ranging collection of capabilities for reasoning about dependences in concurrent Java programs
- Current trends suggest many interesting opportunities for slicing in a concurrent context
- Indus/Kaveri is available for download
 - over 3000 downloads within the past two years
 - working with KSU to release under open source license by end of year

Themes for the Future

- synergistic combination with other techniques
 - dynamic analysis for mining dependences
 - symbolic execution (conditional slicing) to calculate path constraints
 - using dependence information to guide heuristics in model checking
- domain-specific layers on top of Indus that mine dependence information according to specific developer needs
- additional optimizations to aid scalability and precision

Related Technical Papers

- Slicing for model checking reductions (TACAS 06)
- Foundations of Control Dependence (ESOP '05)
- Pruning Interference Dependence using Escape Analysis (CC '04)
- Preserving LTL with Slicing (HOSC '01)
- Notions of Dependence for Java (SAS '99)

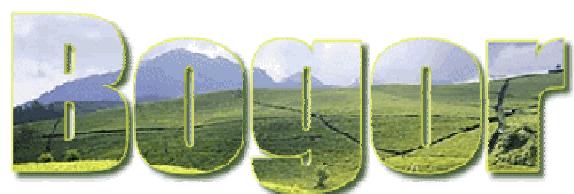
For More Information...



SAnToS Laboratory,
Kansas State University
<http://www.cis.ksu.edu/santos>



Indus Project
<http://indus.projects.cis.ksu.edu>

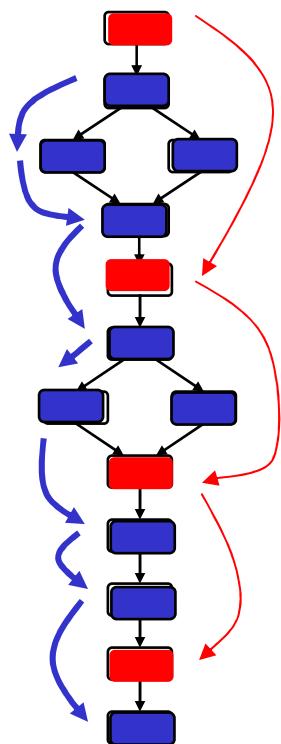


Bogor Project
<http://bogor.projects.cis.ksu.edu>

Applications

Automatic Parallelization

Program CFG

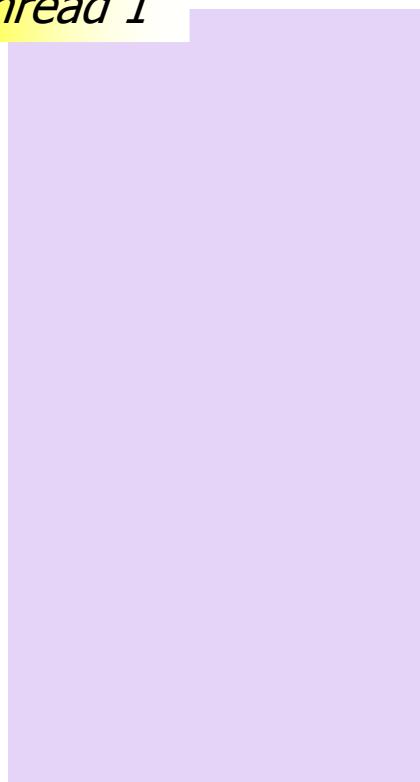


Use slicing to find independent sequences of computation

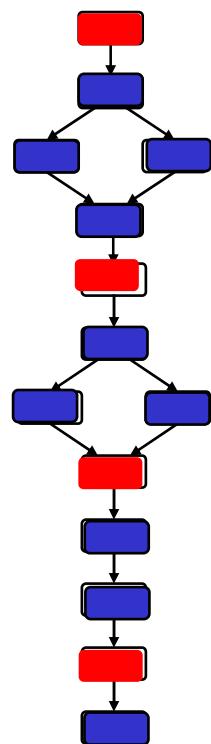
Applications

Automatic Parallelization

Thread 1



Thread 2



Applications

Automatic Parallelization

