MUVI: Automatically Inferring Multi-Variable Access Correlations and Detecting Related Semantic and Concurrency Bugs

Shan Lu¹, Soyeon Park¹, Chongteng Hu¹, Xiao Ma¹, Weihang Jiang¹
Zhenmin Li²,³, Raluca A, Popal, Yuanyuan Zhou¹
¹University of Illinois, ²CleanMake Inc., ³MIT
{shenlu,soyeon,chu7,xiaomao2,wiang3,zl4,yyzhou}@uiuc.edu, ³ralucap@mit.edu

ACM SOSP 2007

Presented by:
Ignacio Laguna

Semantic and Concurrency bugs—Two of the most difficult to detect

• Variable Access Correlations can be exploited to detect these bugs
  – Many variables are correlated
  – Correlated variables need to be accessed together in a consistent manner
  – Failing in updating correlated variables may lead to inconsistent views
Multi-Variable Access Correlations: Example 1  
(MySQL-5.2.0)

- thd->db_length describes the length of the string thd->db
- Semantic connection:
  - Whenever thd->db is modified, thd->db_length needs to be updated accordingly (or at least to be checked)

---

Multi-Variable Access Correlations: Example 2  
(Mozilla-0.8)

- A flag variable (cache->empty) indicates whether an array variable (cache->table) is empty
- Semantic connection:
  - Whenever an item is inserted or removed from the table, (empty) needs to be updated accordingly
Why are multi-variable access correlations important?

• They usually exist only in programmer’s mind
  – They are too tedious to document
  – Can easily be violated by other programmers

• Existing techniques cannot extract such correlations
  – Compiler analysis cannot catch them

• Violating correlations can lead to two types of bugs:
  – Inconsistent updates bugs
  – Concurrency bugs

Bug Type 1:
Multi-Variable Inconsistent Updates

Example 1

```
1721  int Event_job_data::compile(THD* thd)
1722  {  ------
1820  thd->db->old_db;
1833  }   Compile an event*
```

(d) Bug (violating the access correlation)

• If programmer forgets the correlation, he/she may update one variable and forget to update the other correlated variable

• Remember:
  – Whenever thd->db is modified, thd->db_length needs to be updated accordingly
Bug Type 1: Multi-Variable Inconsistent Updates (Cont’d)

Example 2

- The actual string length (str_length) should never go beyond the length allocated for it (Allocated_length)

- Every modification to (str_length) requires a corresponding check or update to (Allocated_length)

Bug Type 2: Multi-Variable Concurrency Bug

- The execution may violate access correlation due to interleaving across threads
- The correct way:
  - To access correlated variables atomically (within the same atomic region)
Contributions of this Work

1) First tool to automatically identify multi-variable access correlations in large programs
   - MUVI (Multi-Variable Inconsistency) tested with latest versions of Linux, Mozilla, MySQL and PostgreSQL
   - Detected 6449 correlations in 19–175 minutes with 83% accuracy
   - Detected bugs were confirmed by the developers

2) MUVI automatically detects inconsistent updates bugs

3) MUVI address limitations of previous methods to detect multi-variable concurrency bugs

---

Real World Examples of Multi-Variable Correlations

<table>
<thead>
<tr>
<th>ID</th>
<th>Source</th>
<th>Variables with access correlation</th>
<th>Variable definitions</th>
<th># of functions they are together (not)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Linux net-device.h</td>
<td>struct net_device_state {</td>
<td>u4 rx_bytes; /* # of received bytes <em>/ u4 rx_packets; /</em> # of received packets */</td>
<td>49 (1)</td>
</tr>
<tr>
<td>b</td>
<td>PostgreSQL time.h</td>
<td>struct tm {</td>
<td>/* second */</td>
<td>25 (0)</td>
</tr>
<tr>
<td>c</td>
<td>Linux f.h</td>
<td>struct fb_var_screeninfo {</td>
<td>u32 red_msb; /* red <em>/ u32 blue_msb; /</em> blue <em>/ u32 green_msb; /</em> green <em>/ u32 transparent_msb; /</em> transparency */</td>
<td>11 (1)</td>
</tr>
<tr>
<td>d</td>
<td>Linux hibiscus.h</td>
<td>struct ccs_session {</td>
<td>lock /* lock */</td>
<td>20 (0)</td>
</tr>
<tr>
<td>e</td>
<td>Linux list.h</td>
<td>struct list_node {</td>
<td>next; /* next */</td>
<td>32 (0)</td>
</tr>
<tr>
<td>f</td>
<td>MySQL street.h</td>
<td>struct st_fs_file* cur_file;</td>
<td>file_stack; /* cur_file points to the top of stack */</td>
<td>69 (0)</td>
</tr>
</tbody>
</table>
Real World Examples of Multi-Variable Correlations (Cont’d)

- Not any two variables from a function are always access-correlated

<table>
<thead>
<tr>
<th>ID</th>
<th>source</th>
<th>Variable definitions</th>
<th># of functions they are together (not)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Linux net-device.h</td>
<td>struct net_device_stats { u64 rx_bytes; /* # of received bytes <em>/ u64 tx_aborted_errors; /</em> # of transfer aborts */ }</td>
<td>4 (68)</td>
</tr>
<tr>
<td>h</td>
<td>MySQL sql_class.h</td>
<td>Class THD { NET net; /* client connection descriptor <em>/ uint db_length; /</em> length of database name */ }</td>
<td>3 (87)</td>
</tr>
</tbody>
</table>

Inferring Variable Access Correlations

- Notation:
  - Access correlation: A1(x) ⇒ A2(y)

  - Where, x and y are variables, A1 and A2 can be any of the three: “read”, “write” or “AnyAcc” (either read or write)

  - Example: write(x) ⇒ read(y): every time x is modified, the value of y has to be read together
What does it mean “Access Together”?

• Accesses to variables are measured to be together in terms of source code distance
  – Measured in terms of lines of code

• MUVI defines “access together” as:
  – if two accesses (reads or writes) appear in the same function with less than MaxDistance statements apart, these two accesses are considered together, where MaxDistance is an adjustable threshold

“Access Correlation” Definition

• Variable $x$ has access correlation with variable $y$ (i.e., $A_1(x) \Rightarrow A_2(y)$):
  – Iff $A_1(x)$ and $A_2(y)$ appear together at least MinSupport times, AND
  – Whenever $A_1(x)$ appears, $A_2(y)$ appears together with at least MinConfidence probability

  – MinSupport and MinConfidence are tunable parameters
Database of Variable Access Information

- MUVI parses source code to collect each function’s variable access information
  - Information is stored in an Acc_Set database

- MUVI considers only common variables like global variables and structure/class fields
  - It avoids short-lived correlations with scalar local variables

- The database stores both direct and indirect accesses to variables through different function calls

Access Pattern Analysis

- **Goal:** to identify variables that are accessed in the same function more than a threshold number of times
  - Each set of variables that satisfy this is an “access pattern”
  - *Note:* an “access pattern” is not an “access correlation” (but is a good candidate)

- MOVI uses the frequent item-set mining algorithm FPClose

- FPClose is applied to the database that is the Acc_Sets of all functions in the program
  - Output: the set of access patterns that are frequent
The Final Step: Correlation Generation and Pruning

- MOVI takes the access patterns to generate correlations
  - It prunes false positives and ranks the results

- Given an access pattern \((x, y)\), it may indicate different correlations \(A_1(x) \Rightarrow A_2(y)\) or \(A_1(y) \Rightarrow A_2(x)\)

- For each possibility, MOVI determines which access correlation holds based on:
  - Support—number of functions in which \(A_1(x)\) and \(A_2(y)\) are together
  - Confidence—conditional probability: given \(A_1(x)\) in a function, \(A_2(y)\) is performed nearby in the same function

Detecting Inconsistency Bug Updates

- An inconsistent update bug is caused by violations to \(write \Rightarrow AnyAcc\) correlations
  - The programmer updates one variable, but forget to update or check its correlated variable

- Basic detection algorithm:
  - For any \(write(x) \Rightarrow AnyAcc(y)\) correlations, examine the violations of it

- Pruning is performed to eliminate false bug candidates
  - Example: suppose we have a bug candidate function \(F\), which misses the access to \(y\)
  - If \(y\) is accessed in \(F\)’s caller or callee functions, it is unlikely to be a bug
Detecting Multi-Variable Concurrency Bugs

- Extensions to two previous data race detectors:

  1) **Lock-set algorithm**: reports a data race bug when it does not find a common lock when accessing a shared memory location
     - **Extension**: check if correlated accesses are protected by a common lock

  2) **Happens-before algorithm**: detects data-race bugs by comparing the logic timestamps of accesses from different threads

Evaluation Mythology

- The latest version of the following applications were used: Linux, MySQL, PostgreSQL, Mozilla.

- Evaluation of MUVI in terms of:
  - Correlation analysis
  - Inconsistent update bug detection
  - Concurrency bug detection capability

- Parameters settings:
  - MinSupport = 10
  - MinConfidence = 0.8
  - MaxDistance = 10 lines of code
Experimental Results:
Variable Access Correlation Analysis

<table>
<thead>
<tr>
<th>App.</th>
<th>#Access-Correlations</th>
<th>#Involved Variables</th>
<th>#Involved Structures</th>
<th>% False Positive</th>
<th>Analysis Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>3353</td>
<td>3038</td>
<td>587</td>
<td>19%</td>
<td>175m2s</td>
</tr>
<tr>
<td>Mozilla</td>
<td>1431</td>
<td>1380</td>
<td>394</td>
<td>16%</td>
<td>157m40s</td>
</tr>
<tr>
<td>MySQL</td>
<td>726</td>
<td>703</td>
<td>209</td>
<td>13%</td>
<td>19m25s</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>939</td>
<td>833</td>
<td>277</td>
<td>15%</td>
<td>98m23s</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>6449</td>
<td>5954</td>
<td>1467</td>
<td>17%</td>
<td>450m30s</td>
</tr>
</tbody>
</table>

Table 5: Variable correlations inferred by MUVI. The correlations presented here include only \texttt{anyAcc} $\Rightarrow$ \texttt{anyAcc} and the other types are presented in Table 8. * The false positive here means the average false positive rate.

---

Experimental Results:
Inconsistent Update Bug Detection

<table>
<thead>
<tr>
<th>App.</th>
<th>#MUVI Bug Report</th>
<th>#New Bugs Found</th>
<th>#New Bugs Confirmed</th>
<th>#Bad Programming</th>
<th>#False Positives</th>
<th>False pos. sources</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S1 S2 S3</td>
</tr>
<tr>
<td>Linux</td>
<td>40</td>
<td>22</td>
<td>12</td>
<td>5</td>
<td>13</td>
<td>6 3 4</td>
</tr>
<tr>
<td>Mozilla</td>
<td>30</td>
<td>7</td>
<td>0</td>
<td>8</td>
<td>15</td>
<td>8 7 0</td>
</tr>
<tr>
<td>MySQL</td>
<td>20</td>
<td>9</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5 2 1</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>5 0 0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>100</td>
<td>39</td>
<td>17</td>
<td>20</td>
<td>41</td>
<td>24 12 5</td>
</tr>
</tbody>
</table>

Table 6: Inconsistent update bugs detected by MUVI. #New bugs confirmed means that the bugs are already confirmed by the corresponding developers after we reported these errors. “S1” stands for semantic exception, “S2” for wrong correlation, and “S3” for no future read.
New Inconsistent Update Bugs Detected in Latest Version of Linux

(a) A new (confirmed) bug found by MUVI in latest version Linux driver framebuffer component

(b) A new (confirmed) bug found by MUVI in latest version Linux driver network component

Experimental Results: Concurrency Bug Detection

<table>
<thead>
<tr>
<th>Bug</th>
<th>Lock-set\textsubscript{MV}</th>
<th>False Overhead*</th>
<th>Happens-before\textsubscript{MV}</th>
<th>False Overhead*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moz-js1</td>
<td>Y</td>
<td>39.9%</td>
<td>Y</td>
<td>21.2%</td>
</tr>
<tr>
<td>Moz-js2</td>
<td>Y</td>
<td>39.8%</td>
<td>Y</td>
<td>1.0%</td>
</tr>
<tr>
<td>Moz-imap</td>
<td>Y</td>
<td>13.2%</td>
<td>Y</td>
<td>1.0%</td>
</tr>
<tr>
<td>MySQL-log</td>
<td>Y</td>
<td>6.5%</td>
<td>Y</td>
<td>5.0%</td>
</tr>
<tr>
<td>MySQL-bug</td>
<td>N</td>
<td>5.9%</td>
<td>N</td>
<td>3.2%</td>
</tr>
</tbody>
</table>

Note: In addition to the above existing concurrency bugs, we detected four new multi-variable concurrency bugs that have never been reported before.
Sensitivity Analysis: 
How to Select *MinConfidence* and *MinSupport*?

- Configuration parameters are taken as the points where false alarm rate changes dramatically
  - Example: when confidence reaches 80%, false positive rate changes from 50% to 20%

Summary

- MUVI proposes source code analysis and data mining techniques to:
  - Automatically infer variable access correlations
  - Detect related bugs

- MOVI extracted 6449 access correlations from Linux, Mozilla, MySQL and PostgreSQL with 83% accuracy

- MOVI detected 39 new bugs (17 already confirmed)