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

Shan Lu[†], Soyeon Park[†], Chongfeng Hu[†], Xiao Ma[†], Weihang Jiang[†]
Zhenmin Li[†‡], Raluca A. Popa[§], Yuanyuan Zhou[†‡]
[†]University of Illinois, [‡] CleanMake Inc., [§] MIT
{shanlu,soyeon,chu7,xiaomao2,wjiang3,zli4,yyzhou}@uiuc.edu, [§]ralucap@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)

```
Class THD
{
   char *db;
 /* currently selected database name*/
   …
   uint       db_length;
/* length of the database name */
   …
} /* client connection descriptor*/
```

MySQL-5.20 sql_class.h

```
int Event_job_data::execute( … )
{
   ……
   thd→db = my_strdup(dbname.str);
   thd→db_length = dbname.length;
   ……
} /* Execute a connection event*/
```

MySQL-5.2 event_data_objects.cc

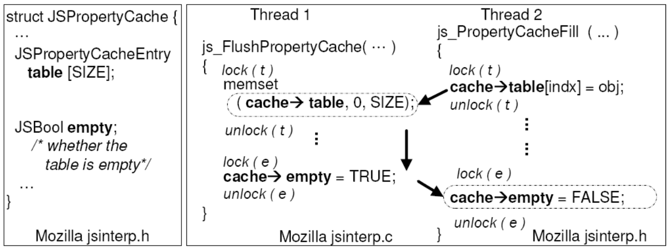
(a) Definition

(b) Variable access correlation

- 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)



```
struct JSPropertyCache {
  …
  JSPropertyCacheEntry
    table [SIZE];


  JSBool empty;
     /* whether the
        table is empty*/
   …
}
```

Mozilla jsinterp.h

Thread 1

```
js_FlushPropertyCache( … )
{  lock ( t )
   memset
    ( cache→ table, 0, SIZE);

   unlock ( t )

   lock ( e )
   cache→ empty = TRUE;
   unlock ( e )
}
```

Mozilla jsinterp.c

Thread 2

```
js_PropertyCacheFill  ( ... )
{
  lock ( t )
  cache→table[indx] = obj;
  unlock ( t )


   lock ( e )
  cache→empty = FALSE;
   unlock ( e )
}
```

Mozilla jsinterp.h

(a) Variables with access correlation

(b) Bug (violating the access correlation due to conflict accesses from another thread, even though no data race on any single variable)

- 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?

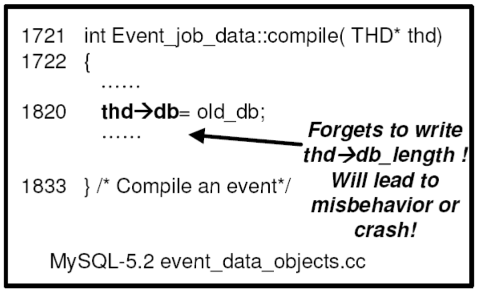
- They usually exist only in programmer's mind
  - They are too tedious to document
  - Can easily be violated by other programmers

- 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*/
```

Forgets to write thd→db_length ! Will lead to misbehavior or crash!

MySQL-5.2 event_data_objects.cc

(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

```
class String
{
  ...
  uint32 str_length;
         /*occupied string length*/
  uint32 Alloced_length;
         /*allocated string length*/
  ...
}
       MySQL-5.2.0 sql_string.h
```

(e) Definition

```
void String::qs_append ( ... )
{
   memcpy ( Ptr + str_length, str, len+1);
   str_length += len;
}
```

Increasing string-length without even a check on Alloced_length is wrong!

MySQL-5.2 sql_string.cc

(h) **Bug** (violating the access correlation)

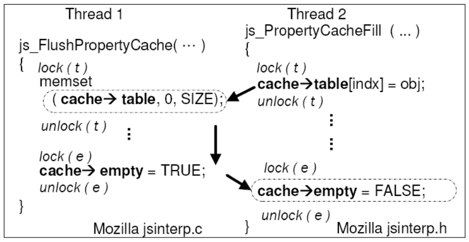
- The actual string length (str_length) should never go beyond the length allocated for it (Allocated_length)
- Every modification to (str_length) requires a corresponding check or update to (Allocated_length)

Bug Type 2:
Multi-Variable Concurrency Bug

```
struct JSPropertyCache {
  ...
  JSPropertyCacheEntry
    table [SIZE];

  JSBool empty;
     /* whether the
        table is empty*/
   ...
}
       Mozilla jsinterp.h
```

(a) Variables with access correlation



Thread 1

```
js_FlushPropertyCache( ··· )
{  lock ( t )
   memset
    ( cache→ table, 0, SIZE);
   unlock ( t )   :

   lock ( e )
   cache→ empty = TRUE;
   unlock ( e )
}
```

Mozilla jsinterp.c

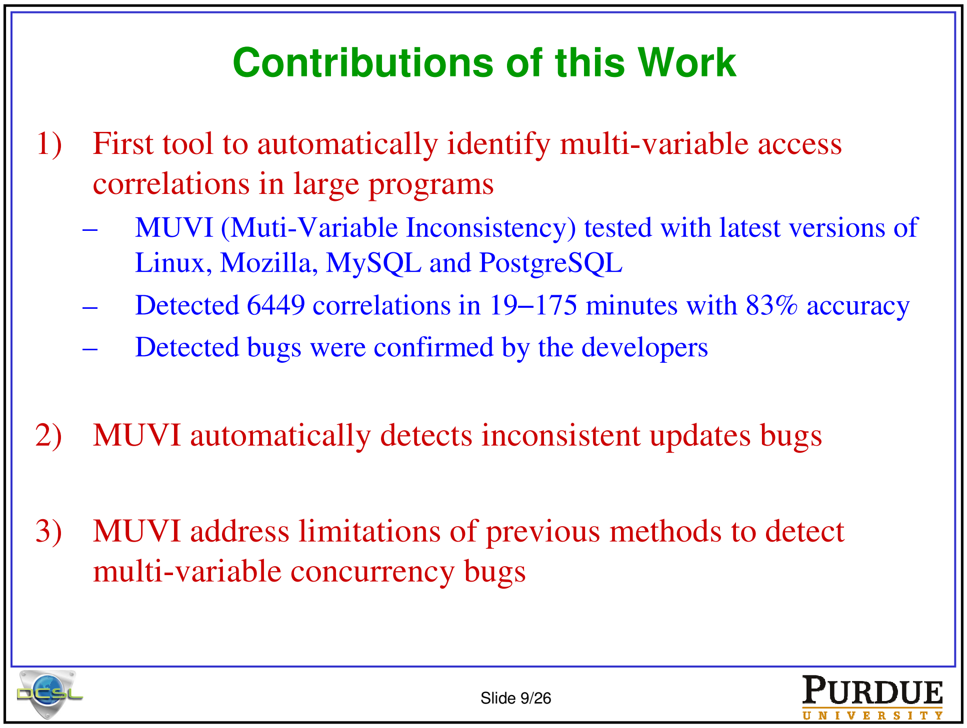
Thread 2

```
js_PropertyCacheFill ( ... )
{
   lock ( t )
   cache→table[indx] = obj;
   unlock ( t )
        :
        :
    lock ( e )
   cache→empty = FALSE;
    unlock ( e )
}
```

Mozilla jsinterp.h

(b) Bug (violating the access correlation due to conflict accesses from another thread, even though no data race on any single variable)

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

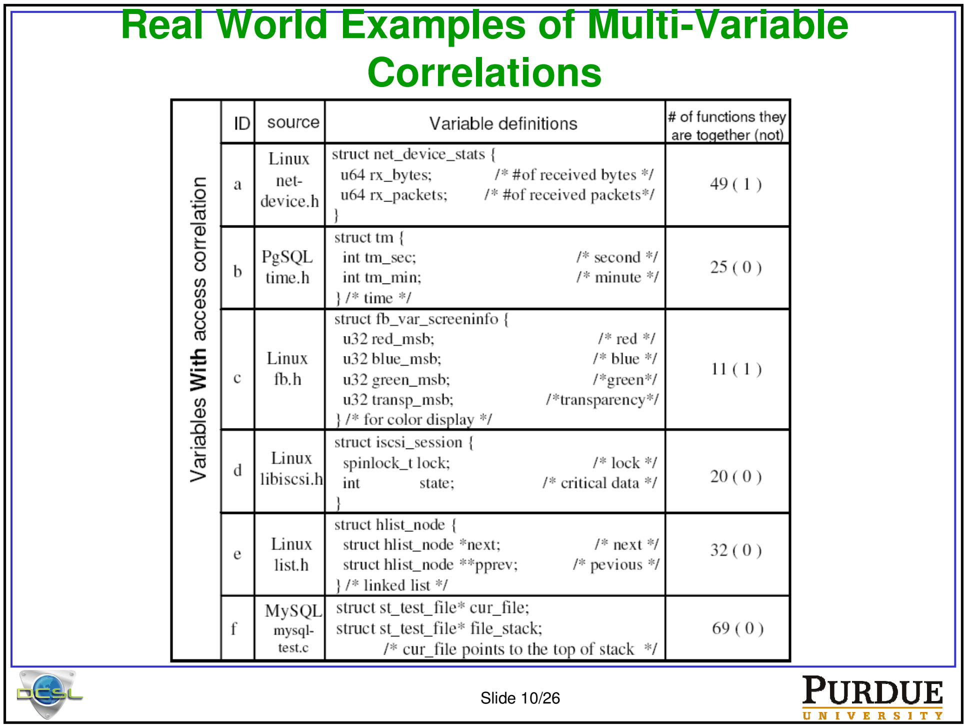


Contributions of this Work

1) First tool to automatically identify multi-variable access correlations in large programs
   - MUVI (Muti-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

	ID	source	Variable definitions	# of functions they are together (not)

Variables **With** access correlation	a	Linux net-device.h	struct net_device_stats {
			u64 rx_bytes; /* #of received bytes */
			u64 rx_packets; /* #of received packets*/
			}
				49 (1)

	b	PgSQL time.h	struct tm { int tm_sec; /* second */ int tm_min; /* minute */ } /* time */	25 (0)
	c	Linux fb.h	struct fb_var_screeninfo { u32 red_msb; /* red */ u32 blue_msb; /* blue */ u32 green_msb; /*green*/ u32 transp_msb; /*transparency*/ } /* for color display */	11 (1)

	d	Linux libiscsi.h	struct iscsi_session {
			spinlock_t lock; /* lock */
			int state; /* critical data */
			}
				20 (0)

	e	Linux list.h	struct hlist_node { struct hlist_node *next; /* next */ struct hlist_node **pprev; /* pevious */ } /* linked list */	32 (0)
	f	MySQL mysql-test.c	struct st_test_file* cur_file; struct st_test_file* file_stack; /* cur_file points to the top of stack */	69 (0)

Real World Examples of Multi-Variable Correlations (Cont'd)

	ID	source	Variable definitions	# of functions they are together (not)

Variables **WithOUT** access correlation	g	Linux net-device.h	struct net_device_stats {
			u64 rx_bytes; /* #of received bytes */
			u64 tx_aborted_erros; /* #of transfer aborts*/
			}
				4 (68)

	h	MySQL sql_ class.h	Class THD {
			NET net; /* client connection descriptor */
			uint db_length; /*length of database name*/
			}
				3 (87)

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

---

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., $A1(x) \Rightarrow A2(y)$):
  - Iff $A1(x)$ and $A2(y)$ appear together at least *MinSupport* times, AND
  - Whenever A1(x) appears, A2(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 A1(x) $\Rightarrow$ A2(y) or A1(y) $\Rightarrow$ A2(x)

- For each possibility, MOVI determines which access correlation holds based on:
  - Support—number of functions in which A1(x) and A2(y) are together
  - Confidence—conditional probability: given A1(x) in a function, A2(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

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

Table 5: Variable correlations inferred by MUVI. The correlations presented here include only `AnyAcc⇒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

App.	#MUVI Bug Report	#New Bugs Found	#New Bugs Confirmed	#Bad programming	#False Positives	False pos. sources		
						S1	S2	S3
Linux	40	22	12	5	13	6	3	4
Mozilla	30	7	0	8	15	8	7	0
MySQL	20	9	5	3	8	5	2	1
PgSQL	10	1	0	4	5	5	0	0
Total	100	**39**	**17**	20	41	24	12	5

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

```
static int imsttfb_check_var (struct
   fb_var_screeninfo *var, struct fb_info *info)
{        ...
         var->red_msb = 0;
         var->green_msb = 0;
         var->blue_msb = 0;
         var->transp_msb = 0;
         …
}
```

drivers/video/imsfftb.c *correct*

```
static int  neofb_check_var(struct
    fb_var_screeninfo *var, struct fb_info *info)
{  ...
   var->red_msb=0;
   var->green_msb=0;
   var->blue_msb=0;
   … // missing update to var→transp_msb
}
```

drivers/video/neofb.c ***BUG***

red_msb, green_msb, blue_msb and transp_msb are used together to set up color.

Missing any one can make display failure.

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

```
static int fr_rx(struct sk_buff *skb)
{   ...
    stats->rx_packets++;
    stats->rx_bytes += skb->len;
    ...
}
```

drivers/net/wan/hdlc_fr.c *correct*

```
static int velocity_receive_frame
(struct velocity_info *vptr, int idx)
{  ...
   stats->rx_bytes += pkt_len;
   … // missing update to stats→ rx_packets
}
```

drivers/net/via-velocity.c ***BUG***

rx_bytes and rx_packets are explained earlier

How could receiving bytes without receiving packets?

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

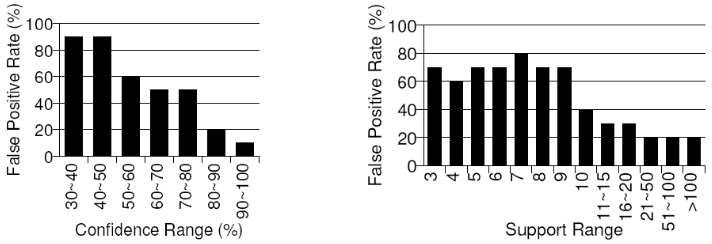
---

Experimental Results: Concurrency Bug Detection

Bug	Lock-set$_{MV}$			Happens-before$_{MV}$		
	Detect Bug?	False Pos.	Over-head*	Detect Bug?	False Pos.	Over-head*
Moz-js1	Y	1	39.9%	Y	1	21.2%
Moz-js2	Y	2	39.8%	Y	5	1.0%
Moz-imap	Y	0	13.2%	Y	0	1.0%
MySQL-log	Y	3	6.5%	Y	6	5.0%
MySQL-blog	N	0	5.9%	N	1	3.2%

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*?



(a) Confidence (b) Support

- 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)