# 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

## ABSTRACT

Software defects significantly reduce system dependability. Among various types of software bugs, semantic and concurrency bugs are two of the most difficult to detect. This paper proposes a novel method, called MUVI, that detects an important class of semantic and concurrency bugs. MUVI automatically infers commonly existing multi-variable access correlations through code analysis and then detects two types of related bugs: (1) inconsistent updates—correlated variables are not updated in a consistent way, and (2) multi-variable concurrency bugs—correlated accesses are not protected in the same atomic sections in concurrent programs.

We evaluate MUVI on four large applications: Linux, Mozilla, MySQL, and PostgreSQL. MUVI automatically infers more than 6000 variable access correlations with high accuracy (83%). Based on the inferred correlations, MUVI detects 39 **new** inconsistent update semantic bugs from the latest versions of these applications, with 17 of them recently confirmed by the developers based on our reports.

We also implemented MUVI multi-variable extensions to two representative data race bug detection methods (lock-set and happens-before). Our evaluation on five real-world multi-variable concurrency bugs from Mozilla and MySQL shows that the MUVI-extension correctly identifies the root causes of four out of the five multi-variable concurrency bugs with 14% additional overhead on average. Interestingly, MUVI also helps detect four **new** multi-variable concurrency bugs in Mozilla that have never been reported before. None of the nine bugs can be identified correctly by the original race detectors without our MUVI extensions.

**Categories and Subject Descriptors:** D.2.5 [Software Engineering]: Testing and Debugging-Diagnostics

**General Terms:** Languages, Reliability.

**Keywords:** Bug detection, concurrency bug, variable correlation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*SOSP'07,* October 14–17, 2007, Stevenson, Washington, USA.
Copyright 2007 ACM 978-1-59593-591-5/07/0010 ...$5.00.

## 1. INTRODUCTION

### 1.1 Motivation

Software defects significantly reduce system dependability. Among various types of bugs, semantic bugs and concurrency bugs are two of the most difficult to detect. This paper focuses on detecting an important class of semantic and concurrency bugs. Our work is based on a novel observation that is general across different software and can be exploited to detect related semantic and concurrency bugs. Our observation is that various *access correlations* commonly exist among multiple variables. In other words, many variables are inherently correlated and need to be accessed together with their correlated peers in a consistent manner. These variables need to be either updated together consistently or accessed together to give the program a consistent view instead of a partial view. For simplicity of description, we refer to those variables that share such an access correlation as *correlated variables.* We also use *variable access correlations* and *variable correlations* interchangeably in this paper.

Variable correlation is a fundamental aspect of program semantics. Consciously or unconsciously, programmers rely on variable correlation to emulate the inherent correlation in the real world and ease their programming. For example, programmers may use correlated variables to represent correlated real-world entities; use one variable to specify the other's property, state or constraints; or use multiple variables to describe different aspects of a complex object.

Figure 1 and Figure 2 give four real-world variable access correlation examples from MySQL and Mozilla. In the example shown in Figure 1(a), `thd->db_length` describes the length of a string (`thd->db`). The semantic connection determines their access correlation: whenever `thd->db` is modified, `thd->db_length` needs to be updated accordingly or at least be checked to see if it is still consistent, as done in Figure 1(b). Similarly, in the Mozilla example shown in Figure 2(a), a flag variable (`cache->empty`) indicates whether an array variable (`cache->table`) is empty. Whenever an item is inserted into or removed from the table, `empty` needs to be updated accordingly as shown in Figure 2(b), so that subsequent execution can decide whether the table can be accessed or not. Section 3 and Section 7 will show more multi-variable correlation examples from real-world applications such as Linux and Mozilla.

Although they are very important, most semantic multi-variable access correlations usually exist only in programmers' mind, because they are too tedious to document. As

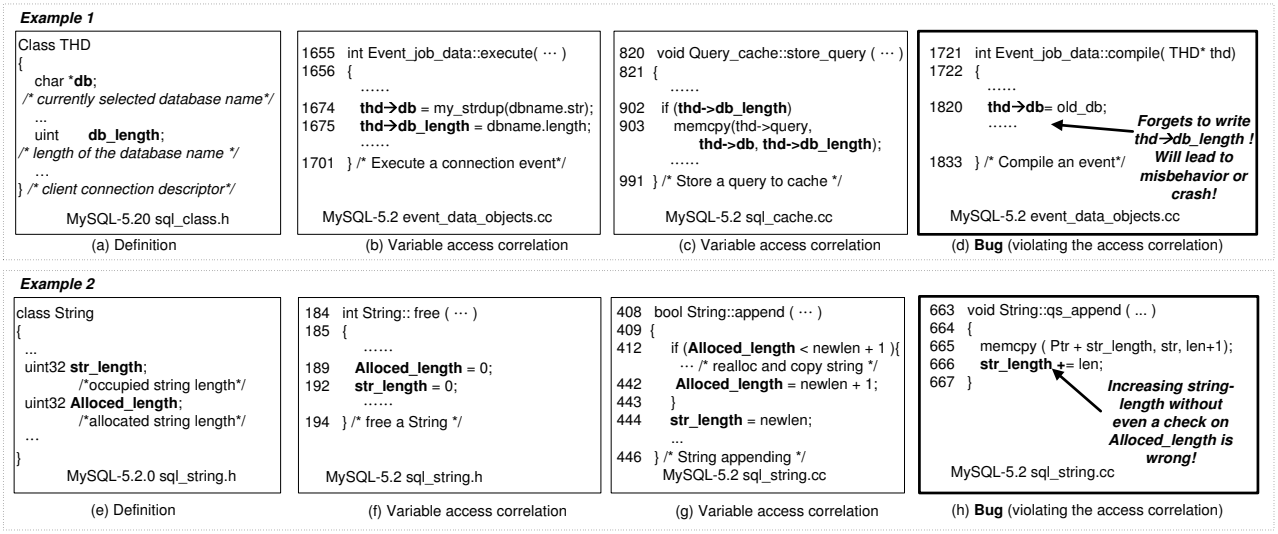


**Figure 1: Two multi-variable access correlation examples and the related inconsistent update bugs from MySQL-5.2.0 (Both are new bugs detected by MUVI from the latest version MySQL and are recently confirmed by developers.)**

a result, access correlations can be easily violated by other programmers or even the same programmers due to miscommunication or careless programming.

Unfortunately, existing techniques cannot effectively extract such semantic correlations. Traditional compiler analysis cannot catch them, because many correlated variables are just *semantically* correlated and do not necessarily have data dependencies, such as the variable `empty` and the variable `table` shown in Figure 2 (a).

Violating multi-variable access correlations can lead to two types of bugs: (1) multi-variable inconsistent update bugs and (2) multi-variable related concurrency bugs. The former is general to both sequential and concurrent programs, but has never been studied before. The latter is specific to concurrent programs, and has not been well addressed by previous concurrency bug detection tools.

**Bug Type 1: Multi-Variable Inconsistent Updates** If a programmer is unaware of or forgets about a multi-variable access correlation, he/she may update only one variable and forget to update or check other correlated variables to make sure that they are still consistent. We call this type of bugs "multi-variable inconsistent updates".

Figure 1(d) gives a real-world bug example from the latest version of MySQL. This bug violates the access correlation between `thd->db` and `thd->db_length`. The string variable is updated with a new value, but the length variable is not updated. Such inconsistency can lead to a crash or other program misbehavior. We detected this bug using our tool and reported it to the MySQL developers who later confirmed it as a true bug.

Figure 1(h) gives another inconsistent update bug example, also detected by our tool, from the latest version of MySQL. The variable `Alloced_length` is correlated with `string_length` since a string's actual length should never go beyond the length allocated for it. Every modification to `string_length` requires a corresponding check or update to `Alloced_length` as shown in Figure 1 (f) and (g). However, function `qs_append` simply updates the `string_length` without any update or check to `Alloced_length`. This mistake can corrupt the `String` object. This bug has also been confirmed by MySQL developers based on our report. More multi-variable inconsistent update bug examples will be shown in Section 7.

**Bug Type 2: Multi-Variable Concurrency Bugs** Unfortunately, in concurrent programs, even if programmers put correlated accesses together everywhere, the execution may still violate the access correlation due to the interleaving across threads. The correct way is to access the correlated variables *atomically* —within the same atomic region. Otherwise, a remote access (read/write) from another thread could interleave between these correlated accesses, either getting an inconsistent view (in case of remote reads) or producing inconsistent final results (in case of remote writes).

Figure 2 (a–b) shows a real-world concurrency bug example from Mozilla. This bug violates the access correlation between `cache->table` and `cache->empty`. Actually, the program does update the two variables consistently everywhere within each thread, as shown in function `js_FlushPropertyCache` and `js_PropertyCacheFill`. However, due to the lack of proper synchronization, concurrent execution of these code segments can still violate the access correlation. As shown in the figure, thread 1 executes `js_FlushPropertyCache`, nullifying the whole `table` and setting `empty` to be true. Unfortunately, these two actions can be interleaved by another thread's `js_PropertyCacheFill` operation. As a result, `empty` is false, but `table` is all-zero. Subsequent execution will reference the object in this empty table based on the empty-flag's value (FALSE), and cause a program to crash.

Multi-variable concurrency bug has not been well studied so far. The only piece of previous work (to the best of our knowledge) [3] that tries to automatically detect multiple-variable involved data race bugs uses a lock-based heuristic: if two variables, `x` and `y`, are ever accessed within one lock critical-section, they should never be separately accessed in different critical sections throughout the program. Although this work raises the issue about multi-variable concurrency bugs, its solution does not work well: all the reported bugs in their experimental results turned out to be false positives. The reason is that two variables being accessed inside one critical section once does not imply that they always need to be accessed in the same critical section. Furthermore,

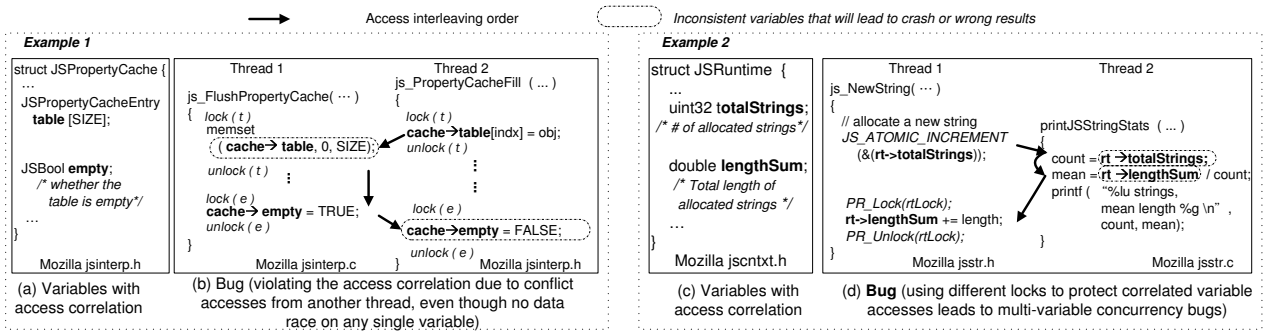


**Figure 2: Two multi-variable access correlation examples and related concurrency bug examples. (a)(b) shows a real example from Mozilla-0.8. Without the locks, the previous race detector may detect each individual access as a race but will NOT suggest that these two accesses need to be protected within the *same* atomic section. If both accesses were protected individually using locks as shown on (b), it is still a bug but cannot be detected by previous concurrency bug detectors. (c-d): This is a *new* concurrency bug detected by our MUVI tool from Mozilla-0.9. Even if each single variable update is protected using locks, the bug still exists since the right implementation should protect the two correlated variables in the *same* atomic section.**

the two variables might coincidently appear in one critical section that is set up for other nearby accesses.

Most of the existing tools on concurrency bug detection cannot deal with above multi-variable related concurrency bugs. The well-known lock-set data race detectors [7, 35], happens-before race detector [30] and various (static or dynamic) enhancements of them [5, 8, 28, 43] are all designed to detect *single* variable races. Specifically, they only check whether the concurrent accesses to each *single* variable are synchronized, i.e. using the same lock or having strict happens-before order among each other. Simply doing race detection at a coarse granularity, such as for each shared object [37], cannot solve this problem since it cannot deal with access correlations among variables belonging to different objects. It can also cause many false positives since not all fields from one object are correlated and need to be accessed within the same atomic region [43]. Atomicity violation bug detection tools [11] check the atomicity of certain code regions, which could include accesses to multiple variables. However, when inferring atomic code regions, existing techniques [25, 40] still focus on single variable. For example, AVIO's access interleaving invariants are associated with each single variable only. In summary, multi-variable access correlations were not considered in previous tools and therefore multi-variable concurrency bugs cannot be correctly identified.

Of course, if one of the correlated variables is not synchronized properly, previous tools may detect it but they would suggest an *inaccurate* root cause that could result in two problems: (1) The programmers simply ignore the bug, if they find the single variable race to be benign or confusing. (2) The programmers give an incorrect or incomplete fix: protecting each single variable separately, instead of the right fix—protecting accesses to the multiple correlated variables *together within the same atomic region*. Such a fix will pass the checking of previous tools, but the bug still exists!

Multi-variable concurrency bugs might exist even if accesses to every single variable are well synchronized. Previous tools would fail to detect such bugs, as shown in Figure 2(b). Figure 2(c–d) shows a **new** multi-variable concurrency bug detected by our tool in Mozilla. In this example, programmers have used locks to protect *all* updates to `rt->totalStrings` and `rt->lengthSum`. However, since different locks are used, the correlated updates are still *not* protected in the same atomic region so the bug still occurs.

While data races may be eliminated or substantially reduced by the emerging transactional memory trend [16, 27], multi-variable concurrency bugs can persist and cause software failures. The reason is that if the updates of two correlated variables are separated into different transactions, the atomicity is still not guaranteed. Recent work such as AtomicSet [36], Colorama [4] recognized this issue and proposed letting programmers manually specify explicitly which variables are correlated. However, this will require significant *manual* effort and may miss those correlations of which programmers are not consciously aware.

With the increasing popularity of concurrent programs driven by the multi-core architecture reality, it is important to address this fundamental limitation associated with concurrency bug detection. This is particularly important because concurrency bugs are notoriously hard to reproduce and diagnose due to their non-deterministic property [19]. An effective tool that can help detect more types of concurrency bugs would be highly demanded by programmers.

## 1.2 Our Contributions

This paper proposes an innovative and practical approach called MUVI (MUlti-Variable Inconsistency) to *automatically* identify multi-variable access correlations from programs, and detect both multi-variable related inconsistent updates and concurrency bugs. We evaluate our ideas and our tool using several large open-source applications, including Linux, Mozilla, MySQL and PostgreSQL. Specifically, our paper makes the following contributions:

*(1) The first tool (to the best of our knowledge) to automatically identify the commonly existent multi-variable access correlations in large programs*: Our tool combines static program analysis and data mining techniques to automatically infer multi-variable correlations. MUVI also automatically prunes false positives and ranks correlations using various techniques. Our experimental results with Linux, Mozilla, MySQL, and PostgreSQL (with 0.8–3.6 million lines of code) show that MUVI identifies a total of 6449 multi-variable access correlations efficiently (within 19–175 minutes) with an accuracy of around 83%.

The automatically-inferred variable correlations can be used in three different ways: (1) They can be stored in a specification database so that programmers can refer to it

to avoid mistakes and encapsulate correlated accesses to improve the code modularization. (2) They can be used to automatically annotate the source code so that other tools can leverage such semantic information. For example, the recently proposed AutoLocker [27] can use this correlation information to assign the same lock to correlated variables. It can also help provide variable grouping information needed in Colorama [4] and AtomicSet [36]. (3) They can be used to detect program bugs—violations to these correlations, as demonstrated in our work (summarized below).

*(2) The first tool to automatically detect multi-variable inconsistent update bugs*: Based on the inferred multi-variable correlations, MUVI also automatically scans the source code to detect places where correlated variables are not updated consistently. MUVI applies code analysis and other techniques to prune false positives and rank bug reports. Our experimental evaluation shows that MUVI has detected a total of 39 (22, 7, 9 and 1, respectively) new bugs from the latest version of Linux, Mozilla, MySQL and PostgreSQL with 17 bugs recently confirmed by the corresponding developers based on our bug reports. Moreover, we have also detected 20 places with bad programming practices that can easily introduce bugs later. Our inconsistent update bug detection false positive rate is reasonable (41% on average).

*(3) Address the fundamental multi-variable limitation of previous concurrency bug detection methods.* We extend two classic race detection methods (lock-set and happens-before) to detect multi-variable related data races. We also discuss how to extend other concurrency bug detectors such as AVIO [25], RaceTrack [43], and RacerX [8] to deal with multi-variable concurrency bugs. We evaluate the above multi-variable extensions using five real-world multi-variable concurrency bugs, whose root causes can *not* be detected by the original race detection tools. Our extensions successfully help previous tools to identify the correct root causes of four out of the five tested bugs. Interestingly, our multi-variable extensions also help detect *four new multi-variable concurrency bugs* in Mozilla that have never been reported before. None of the nine bugs can be identified correctly by the original race detectors without our multi-variable extensions.

## 2. VARIABLE CORRELATIONS

Correlation, originating from statistics, means a pair of items' departure from independence. Many items in real-world are not independent among each other, neither are program variables in our software. Variable access correlation is inherent in program semantics. The following shows the typical semantic reasons of variable access correlations:

- *Constraint Specification:* A variable specifies a constraint, a property or a state of its correlated peers. For example, in Figure 1(a), `thd->db_length` describes the length property of the string `thd->db`; in Figure 2(a), `cache->empty` records the state of `cache->table`.
- *Different Representation:* Two variables represent the same information in different ways. As shown in Table 1(a), `rx_bytes` and `rx_packets` record the incoming network traffic in different units: number of bytes and number of packets. They are accessed together [1] in 49 functions except for one, which is a new inconsistent update bug detected by MUVI and confirmed by the Linux developers.

[1]More formal discussion of *together*ness is in Section 3.1.

| | ID | source | Variable definitions | # of functions they are together (not) |
|---|---|---|---|---|
| Variables **With** access correlation | a | Linux net-device.h | struct net_device_stats {<br>u64 rx_bytes; /* #of received bytes */<br>u64 rx_packets; /* #of received packets*/<br>} | 49 ( 1 ) |
| | b | PgSQL time.h | struct tm {<br>int tm_sec; /* second */<br>int tm_min; /* minute */<br>} /* time */ | 25 ( 0 ) |
| | c | Linux fb.h | struct fb_var_screeninfo {<br>u32 red_msb; /* red */<br>u32 blue_msb; /* blue */<br>u32 green_msb; /*green*/<br>u32 transp_msb; /*transparency*/<br>} /* for color display */ | 11 ( 1 ) |
| | d | Linux libiscsi.h | struct iscsi_session {<br>spinlock_t lock; /* lock */<br>int state; /* critical data */<br>} | 20 ( 0 ) |
| | e | Linux list.h | struct hlist_node {<br>struct hlist_node *next; /* next */<br>struct hlist_node **pprev; /* pevious */<br>} /* linked list */ | 32 ( 0 ) |
| | f | MySQL mysql-test.c | struct st_test_file* cur_file;<br>struct st_test_file* file_stack;<br>/* cur_file points to the top of stack */ | 69 ( 0 ) |
| Variables **WithOUT** access correlation | g | Linux net-device.h | struct net_device_stats {<br>u64 rx_bytes; /* #of received bytes */<br>u64 tx_aborted_erros; /* #of transfer aborts*/<br>} | 4 ( 68 ) |
| | h | MySQL sql_ class.h | Class THD {<br>NET net; /* client connection descriptor */<br>uint db_length; /*length of database name*/<br>} | 3 (87 ) |

**Table 1: Six examples with multi-variable correlation and two examples with no access correlation, and their access patterns (the number of times they are accessed together within the same function). The numbers in () are the numbers of times that the correlated variables are *not* accessed together.**

- *Different Aspects:* The correlated variables specify different aspects of a complex data to emulate correlated real-world entities. For example, `tm_min` and `tm_sec` in Table 1(b) represent the minute and the second of a certain moment. They are accessed together in 25 functions and never separated. Table 1(c) shows four correlated fields, `red_msb`, `blue_msb`, `green _msb` and `transp_msb`, that represent the red, blue, green and transparency information for color screenplay. They are accessed together in 11 functions with only one exception, which is also a recently confirmed new bug detected by MUVI.
- *Implementation-demand:* The correlated variables cooperate with each other in order to implement a specific functionality of the program. Table 1(d) shows that the field `lock` is used to protect the critical data `state` in structure `iscsi_session`; therefore accesses to `state` are always together with accesses to `lock`. Similar examples can be seen in Table 1(e) (double-linked list data structure) and Table 1 (f) (stack data structure).

Obviously, not any two variables from a program are access-correlated. For global variables, such claim is intuitive. For multiple fields from the same structure, this also holds. The (g), (h) in Table 1 provide two examples: although the fields in each pair belong to the same structure, they do not have access correlation as they are accessed together in only 3–4 functions and are accessed separately in 68 or 87 functions.

Since correlated variables are connected semantically, violating an access correlation poses the risk of breaking the semantic connections and consistency, and might threaten

| Constraint | Definition | Example |
|---|---|---|
| read($x$) ⇒read($y$) | Every read to $x$ semantically requires a read to $y$ | Figure 2: read to `cache->table` has to be preceded by checking `cache->empty` |
| write($x$) ⇒write($y$) | Every write to $x$ semantically requires a write to $y$ | Figure 1: update to a string variable (`THD::db`) will bring update to its length variable (`THD::db_length`), vice versa. |
| write($x$) ⇒AnyAcc($y$) | Every write to $x$ semantically requires an access to $y$ | Table 1(d): write to `state` has to check or grab the lock `lock` |
| AnyAcc($x$) ⇒AnyAcc($y$) | Every access to $x$ semantically requires an access to $y$ | Table 1(c): accesses to `fb_var_screeninfo`'s `green, blue, read, transp` fields are always together. |

**Table 2: Examples of access constraints in correlations. `AnyAcc` means either read or write. There are totally nine types of access constraints. Here we only show four types for demonstration. The other five types are (1) read($x$) ⇒write($y$), (2) read($x$) ⇒AnyAcc($y$), (3) write($x$) ⇒read($y$), (4) AnyAcc($x$) ⇒read($y$), and (5) AnyAcc($x$) ⇒write($y$).**

the program correctness, as demonstrated in the bug examples shown in Section 1 and 7.

**Access constraints in correlations** Access correlations do not necessarily mean that correlated variables only need to be *updated* together. As shown in Table 2, sometimes, correlated variables are always accessed (both read and write included) together; sometimes, reading a variable should be preceded by checking (reading) another variable; in some other cases, writing one variable requires checking (reading) *or* writing its correlated variables.

Similarly, some multi-variable access correlations are not necessarily symmetric. Modifying one may require modifying or checking the others accordingly, but the other way around is not necessarily true. In the example (d) shown in Table 1, updating `state` requires accessing the lock variable, but accessing `lock` does not need to modify `state`.

Based on the above two observations, there are nine different types of access constraints for two correlated variables (four of which are illustrated in Table 2). For simplicity of description, for two variables $x$ and $y$, we use the following notation to represent an access correlation formally:

$$A1(x) \Rightarrow A2(y)$$

where $A1$ and $A2$ can be any of the three: "read", "write" or "AnyAcc" (either read or write). For example, an access correlation, `write`($x$) ⇒`read`($y$), means that every time $x$ is modified, the program needs to read the value of $y$ together. Similarly, `AnyAcc`($x$) ⇒`AnyAcc`($y$) means that if $x$ is accessed (either read or written), $y$ needs to be accessed together. The "togetherness" notion is defined in the next section.

# 3. VARIABLE CORRELATION ANALYSIS

Variable access correlations are typically too many and tedious for programmers to specify manually. Therefore, if we can automatically infer such access correlations, we can use them as specifications, annotations and help bug detection. This section presents how MUVI automatically infers variable correlations from programs.

## 3.1 Correlation Analysis Overview

Similar to much previous work on extracting information and invariants from source code [9, 20, 22] and dynamic execution [10, 15], we assume that the target program is reasonably mature, i.e., it is not at its initial development stage. Almost all open source and commercial software meet this requirement, so it does not significantly limit the applicability of our work.

Since our goal is to extract multi-variable *access* correlations, not arbitrary correlations, we base our correlation analysis on variable access patterns and examine what variables are usually read or written together. For example, if every time when variable $x$ is updated, variable $y$ is also read together, it is very likely that some underlying program semantic (access-correlation) connects them together. In this case, we claim that `write`($x$) ⇒`read`($y$).

**"Access Together" Definition:** A non-trivial question that immediately emerges is how we claim that two accesses are "together". There could be many possible measures. For example, we can use source code distance (measured in terms of lines of code) or dynamic execution distance (measured in the dynamic instruction trace) as a metric. Although no single measure is absolutely the best in all cases, in our scenario, dynamic execution distance is obviously not a good measure. The reason is that two correlated accesses can easily be separated by a loop or a function invocation, and thus have a large dynamic distance. In contrast, static code distance does not suffer from this limitation. It is also more aligned with programmers' coding process, which is usually centered around semantic correlations and functionalities.

Obviously, simply counting the absolute source code line gap between accesses is not enough, because that naive counting will consider two accesses in two adjacent functions as "together", which is unreasonable. Therefore, a good static distance based measure should also consider code structures such as basic blocks, functions, and files. Comparing all these units, a basic block is too small, while a file is too large. A function is the right unit since, from the programmers' point of view, a function is usually the basic unit to perform a certain task, which fits the correlation semantic.

Based on all these considerations, 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.

Our current prototype uses a cutoff threshold to determine whether two accesses are *together*. It is also conceivable to use a scalar metric, ranging from 0 to 1, to measure the "togetherness". Such scalar metric can also be used for ranking and false positive pruning.

**"Access Correlation" Definition:** Now we can formally define access correlation: $x$ has access correlation with $y$, i.e. A1($x$) ⇒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, where *MinSupport* and *MinConfidence* are tunable parameters, A1 and A2 can be, respectively, any of the three: read, write or AnyAcc, and together-ness is defined as above.

**Correlation Inference Steps:** MUVI's correlation analysis is then to find out all A1($x$) ⇒A2($y$) from the target program. It is conducted in three steps:

(1) *Access Information Collection:* MUVI parses the source code and collects each function's variable access information, including the set of variables accessed within each function, the access types and locations in source code. The information is stored in an `Acc_Set` database.

(2) *Access Pattern Analysis:* MUVI uses a frequent pattern mining technique to process the `Acc_Set` database and finds out all the variable sets that frequently appear together. This step produces a pool of variable access correlation candidates.

(3) *Correlation generation, pruning and ranking:* The last step starts from the correlation candidates produced at step 2. It uses the detailed information stored in `Acc_Set` to generate, prune and rank different types of correlations.

## 3.2 Access Information Collection

MUVI conducts flow-insensitive and inter-procedural static analysis to collect variable access information from every functions. In other words, the goal of this step is to compute the `Acc_Set` for each function in the program. To achieve this goal, MUVI needs to address several issues:

*(1) which variables are we interested in?* Theoretically, all variables could be involved in some correlation relationships. However, correlations involving structure/class fields and global variables are usually more common and more important than those short-lived correlations involving only scalar local variables. Therefore, MUVI considers two types of variables: global variables and structure fields (regardless of which object instance the field is associated with), represented by structure/class-name::field-name(e.g. THD::db). These structures can be globally defined, locally defined or dynamically allocated. For simplicity of description, we refer to both global variables and structure fields as "variables" in the remainder of the paper.

*(2) what detailed access information do we need?* For each variable access, we need the following detailed information to conduct further analysis: access type (read or write) for classifying different types of correlations; source code position (file name and line number) for measuring the "togetherness" of two accesses; whether an access is from a function itself or its callee functions for pruning purposes.

*(3) how to handle function calls?* A function can access a variable directly (referred to as a *direct access*) or via its callees (referred to as an *indirect access*). The `Acc_Set` of a function should include both direct and indirect accesses. Otherwise, some access correlations would be missed, especially in cases when a variable is read or written inside some utility functions, such as `get()` or `put()`, for the purpose of encapsulation. To achieve this, MUVI first builds a call graph of the target program and then traverses the call graph bottom up starting from the leaf nodes. All direct accesses made in a function are added to this function's `Acc_Set`. If function $F$ calls function $f1$, $f1$'s *direct* accesses are also added to $F$'s `Acc_Set` (as shown in Figure 3), but we do not propagate $f1$'s accesses any further along the call chain, i.e. not to $F$'s callers. The rationale is that, if two accesses are several functions apart in the call chain, the chance that they are correlated is small (otherwise it is difficult for programmers to maintain the code). Therefore, if we propagate $f1$'s direct accesses too many levels upward, we can easily introduce many false correlations. As a future work, we can make this scheme more flexible: allow propagations across many levels of function calls, but use higher weights for more direct accesses and lower weights for less direct ones.

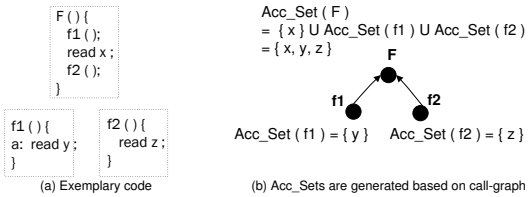


**Figure 3: `Acc_Set` collection for an example call graph**

For a direct access $a$ in a function $f1$, its source code position is the line number where this access is made. However, when this access is propagated to $f1$'s caller $F$, the access's source code position in $F$'s `Acc_Set` is the source line where $F$ calls $f1$ so that $F$'s direct accesses are still relatively close to this access $a$.

**Issues and Extensions** The above algorithm used in our current access information collection is context-insensitive, i.e., it does not consider the caller's effect during the analysis of `Acc_Set`. To be context-sensitive, we will need a parameterized `Acc_Set` summary for each function, so that different call sites of a function will get different `Acc_Set` instantiations.

Since MUVI relies on source code parsing, we cannot get information regarding accesses made inside a library whose source code is unavailable. For those common and important library calls (e.g. strlen, strcpy, memset), we can manually specify their `Acc_Sets`, i.e. the access type of a library call to its parameters in a similar way to previous work [41]. We can also extend our analysis to extract correlations from binary code, which remains as our future work.

Since MUVI analyzes access correlations for fields in structures (regardless the object instance), pointer aliasing of structure objects does not affect our analysis. For global variables, pointer aliasing could affect the accuracy, but our empirical results suggest that its effect is insignificant.

## 3.3 Access Pattern Analysis

The goal of this step is to identify variables that are accessed in the same function (i.e., appear in the same `Acc_Set`) for more than a threshold number of times. For each set of variables that satisfies this property, we refer to it as an access pattern. Note that a pattern is not an access correlation. Instead, it may imply a set of candidate access correlations of different types, such as `read`$(x)$ ⇒`read`$(y)$, `read`$(x)$ ⇒`write`$(y)$, etc.

Given the `Acc_Set` of each function, different approaches can be used to extract such access patterns. One solution is to count the number of `Acc_Sets` containing both $x$ and $y$ for every pair of variables $(x, y)$. Although this solution is relatively simple, it cannot scale to large programs with millions of lines of code, such as Linux. Moreover, it would be hard to extend this algorithm to consider access correlations that involve more than two variables such as `write`$(x)$ & `write`$(y)$ ⇒`read`$(z)$.

Since access correlations involving more than two variables do exist in real-world programs (e.g., Table 1(c) and more examples in Section 7.4), we do not use the above method. Instead, we leverage a well-studied data mining technique: frequent itemset mining [13]. Frequent itemset mining examines a database where each entry is an itemset,

i.e. a set of items, and tries to *efficiently* discover which sub-itemsets (subsets of an itemset) are *frequent*, i.e. contained in more than a threshold (called *MinSupport*) number of database entries. For example, in an itemset database $\mathbb{D}$,

$$\mathbb{D} = \{\{w, y, z\}, \{v, w, y, z\}, \{w, x, y\}\},$$

if *MinSupport*=3, the mining result will show that itemsets $\{w\}$, $\{y\}$, $\{w, y\}$ are frequent. If *MinSupport*=2, itemsets $\{w\}$, $\{y\}$, $\{z\}$, $\{w, y\}$, $\{w, z\}$, $\{y, z\}$, $\{w, y, z\}$ are frequent.

The specific frequent itemset mining algorithm used in MUVI is called *FPclose* [13]. It is one of the most efficient frequent itemset mining algorithms. Due to space limit, we do not present the details of the FPclose algorithm here.

We apply FPclose to our `Acc_Set` database, consisting of the `Acc_Sets` of all functions from the target program. The FPclose algorithm outputs the frequent sub-itemsets, i.e., access patterns—sets of variables that are accessed (regardless of their access types) in more than *MinSupport* number of functions. For example, at the threshold *MinSupport* = 10, MySQL's variable pair {`THD::db`, `THD::db_length`} (Figure 1) is included in our candidate list, because they appear together in 39 functions. On the other hand, the non-correlated variable examples shown in Table 1(g)(h) in the previous section will not be selected as candidates, since they are accessed in only a few functions together.

## 3.4 Correlation Generation and Pruning

In this final step, MUVI takes the access patterns to generate correlations, prune false positives and rank the results. Basically, given a pattern such as $(x, y)$, it may indicate a total of 18 correlations in the form A1($x$) ⇒A2($y$) or A1($y$) ⇒A2($x$), where A1 and A2 can be read, write, or AnyAcc. For each of the above possibilities, MUVI determines whether the access correlation holds by mainly considering two basic metrics, *support* and *confidence*, plus some other considerations.

- *Support.* Given a correlation **C**: A1($x$) ⇒ A2($y$), its support, denoted as *support*(**C**), is the number of functions in which A1($x$) and A2($x$) are together (based on the definition of togetherness in Section 3.1). Such a function is called a *supporter* of this correlation. If a correlation candidate has fewer than *MinSupport* number of supporters, it is pruned out.
- *Confidence.* The confidence of a correlation **C**: A1($x$) ⇒ A2($y$) measures the conditional probability that, given A1($x$)'s presence in a function, A2($y$) is performed nearby in the same function. It is calculated via *support*(**C**) / *support*(A1($x$)), where *support*(A1($x$)) is the number of functions that perform A1($x$). Obviously, even if a correlation candidate has many supporters, a low confidence would make the correlation not trustworthy. Therefore, MUVI uses a threshold *MinConfidence* to prune out correlation candidates with too low confidence.
- *Other considerations.* In addition to the above two metrics, we also differentiate direct function supporters from indirect function supporters to improve the accuracy of correlation analysis. The former directly access variables involved in the correlation, while the latter access the involved variables via their callee functions. Clearly, direct supporters carry more weight than indirect supporters. Due to this concern, MUVI counts the number of direct supporters and prunes out correlation candidates with lower than a threshold *MinDirectSupport* number of direct supporters. It is also conceivable to give different weights to direct and indirect supporters when counting the support and confidence. Furthermore, we also prune out false correlations caused by popular variables, such as `stdout` and `stderr`. These variables are accessed in many functions, and therefore can easily be falsely inferred as correlating to many other variables. To address this problem, we prune out correlations that involve variables that appear in more than a threshold number of functions.

**Ranking** Large software will have a long list of correlations. In order to help programmers prioritize their efforts, MUVI ranks the correlation results based on *support* and *confidence*. Specifically, when the *support* is large enough, indicating that its *confidence* is statistically reliable, we rank the correlations based on their *confidence*; if the *support* is not large enough, we rank the correlations based on *support*.

**Parameter setting** Setting above threshold parameters needs to consider the tradeoffs between false positives and false negatives. Our default parameter setting (section 6) should provide a good initial balance point for most applications as shown in our experiments on four applications (Section 7). Users can start with the default setting and tune it based on their empirical experience. If users have concerns with the false positive number, they can increase the threshold parameters, such as *MinConfidence* and *MinSupport*. If users can tolerate more false positives, they can decrease the parameters to get more inference results.

Parameter setting also depends on the targeting program properties and how users plan to use the inferred correlation. For example, small applications can use relatively small *MinSupport*. If we want to use the correlations as strict requirements and report all violations to them as bugs, we would better be conservative and pick large parameters. If we only use the correlations as hints to help with bug detection, such as in the case of the multi-variable concurrency bug detection in Section 5, we can be more aggressive and choose relatively small parameter values.

# 4. INCONSISTENT UPDATE BUG

As discussed in Introduction, violating access correlations can lead to two types of bugs, inconsistent update bugs and multi-variable concurrency bugs. This section presents how MUVI detects the first type of bugs, and the next section will explain how MUVI detects the second type.

**What is an inconsistent update?** Inconsistent update bugs are caused by violations to write ⇒AnyAcc access correlations. That is, sometimes, the programmer updates one variable, but forgets to update or check its correlated variable. As a result, the memory states of the correlated variables become inconsistent. Such mistakes can be easily made by programmers due to careless programming or miscommunication (as demonstrated by many bugs detected by MUVI in the latest version of Linux, Mozilla, etc). We do not consider violations to access correlations that start with a read access, because read does not directly change memory state and usually does not cause severe damages by itself.

**How to detect?** Based on MUVI 's correlation analysis results, the basic algorithm of detecting inconsistent updates is now straightforward. For any write($x$)⇒AnyAcc($y$) correlation, we examine the violations to it. All the functions that only update $x$ without accessing $y$ are treated as inconsistent update bug candidates.

**Ranking and pruning** We first prune out likely false bug candidates based on caller–callee consideration. Given 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. In our current prototype, MUVI checks two-level caller and callee functions. Of course, we can examine more levels, but our empirical results with several large software find it unnecessary.

After pruning, we rank the remaining bug candidates based on the following considerations:

**(1)** Violations to the strong write ⇒write correlations are ranked at the top. Intuitively, write ⇒write provides the most rigorous consistency requirements. If an update to $y$ is "required" after each update to $x$, the violation, which neither updates nor checks $y$, is most likely to cause memory state inconsistency.

**(2)** The more violations a correlation has, the lower rank it gets. Similar to previous rule inference work [20, 22], if there are too many violations to a correlation, it is unlikely for those violations to be all bugs. The rationale is that in mature software, programmers are less likely to introduce many bugs with the exact same root cause.

**(3)** The more trustworthy a correlation is, the more likely a violation to it is a bug. After the previous two steps, we rank the remaining violations based on the ranks of the corresponding correlations.

As an example, Linux function `velocity_receive _frame` violates the access correlation described in Table 1(a), write (`net_device_stats::rx_packets`) ⇒write (`net_device_stats ::rx_bytes`). Since it is the only violation to a highly ranked write⇒write correlation, it is ranked number 1 in our bug report, and it has been confirmed as a true bug by the Linux developers.

**Discussion** Of course, MUVI inconsistent update bug detection cannot solve all the multi-variable inconsistency problem. Since MUVI only considers access types (read or write) and not specific variable values, both false positives and false negatives could occur due to special variable values.

It is possible that when `x` is assigned a certain value, the update to `y` is unnecessary. For example, in MySQL, `SHOW_VAR::type` describes the type of data stored in `SHOW_VAR ::value`. Usually, they are updated together. However, when `type` is assigned to be `UNDEF`, there is no need to update or check `value`. It is also possible that although two correlated variables are updated together, the values assigned to them are inconsistent. Both are out of the scope of our approaches and can potentially be solved by combining value invariant techniques such as DIDUCE [15] and DAIKON [10] with our variable correlation analysis.

# 5. MULTI-VARIABLE CONCURRENCY BUG

Concurrent execution is another major source of multi-variable access inconsistency. This section first briefly describes the existing concurrency bug detection techniques and then discusses how to extend them to detect multi-variable concurrency bugs.

## 5.1 Background

**Data Race** Data-race is the best studied type of concurrency bugs. It occurs when two accesses, at least one of which is a write, from different threads to **one** shared variable (called *conflict accesses*) are not synchronized [35]. As discussed in the Introduction, previous race detectors focus on concurrent accesses to each **single** variable and miss synchronization problems among **multiple** correlated variables. In the following, we briefly describe the two classic race detection algorithms: the lock-set algorithm [7, 8, 35, 43] and the happens-before algorithm [6, 30, 32]; and we will demonstrate how to extend them in the next subsection.

The lock-set algorithm reports a data race bug when it finds that there is no common lock held during accesses to a shared memory location. To perform such a check, the algorithm maintains the set of locks currently held by each thread (called the thread *Lock Set*), and the set of locks that have been used to protect each variable so far (called the *Candidate Set*). A candidate set is initialized as all possible locks, and updated upon every access to the corresponding variable by intersecting with the thread lock set. A data race is detected when the candidate set is empty. In our study, we extend an existing dynamic implementation (by open source developers) of this algorithm in Valgrind [29].

The happens-before algorithm [6] detects data-race bugs by comparing the logic timestamps of accesses from different threads to the same shared variable. If the timestamps do not indicate a happens-before order among these accesses, a race is reported. Here the logic timestamp is calculated based on thread interaction and synchronization. In our study, the basic happens-before algorithm is implemented using a binary instrumentation tool, PIN [26].

## 5.2 Multi-Variable Race Detection

Multi-variable concurrency bugs cannot be solved by single variable race detection. As shown in the real bug examples shown in Figure 2 and later in Figure 7, multi-variable concurrency bugs could occur when there is no race or only benign races on each single variable.

The basic idea of multi-variable extension to previous race detectors is to, for any pair of conflicting accesses to the same variable, in addition to examining their locksets or their order, we also examine if either access has correlated accesses, and if their correlated accesses share the same lock or are also ordered with respect to the conflicting accesses.

### 5.2.1 Multi-Variable Extensions to Lock-Set

Following the above basic idea, the multi-variable extension to the lock-set algorithm, referred to as lock-set$_{\text{MV}}$, is as follows. For every pair of accesses, A1($x$) and A2($x$) (one is a write), from different threads to the same shared variable $x$, we check if they are protected by at least one common lock (the basic lock-set algorithm), *and also* check if any correlated access A3($y$), of A1($x$) or A2($x$), is also protected by a common lock with A1($x$) *and* A2($x$), as shown in Figure 4.

Note that we require only *correlated accesses*, instead of all accesses, to $x$ and $y$ to be protected by a common lock with their conflicting accesses. For example, if $x$ and $y$ only have a write-write correlation, then only those write accesses to $x$ and $y$ that appear together are checked.

An implementation challenge is that the access correlations are inferred from source code in the format of (global) variable names and structure/class field names, while the Valgrind lockset implementation works at binary code level. Translation is therefore required to bridge this gap. For global variables, we get their memory addresses from the compiler and feed them to lock-set$_{\text{MV}}$ prior to the detection run. For shared structure/class-objects, their dynamic allocation and deallocation raise extra challenges. In order to

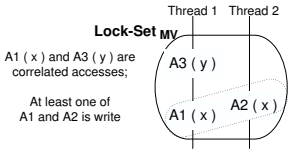


**Figure 4: Multi-variable extension to the lock-set algorithm**

dynamically update the mapping from structure/class fields to memory locations for lock-set$_{\rm MV}$, we use source-to-source code translation to wrap `malloc`-like and `free`-like functions with structure type information. Lock-set$_{\rm MV}$ dynamically intercepts these wrappers to get the correlation information.

### 5.2.2 Multi-Variable Extensions to Happens-before

Extending the happens-before algorithm to consider variable correlations is quite similar with our extension to the lock-set algorithm. The logic timestamp calculation for each memory access is the same as that in the traditional happens-before algorithm. The difference is that happens-before$_{\rm MV}$ will compare the logic timestamp between not only accesses to the same memory location but also correlated accesses. If the timestamp comparison shows no happens-before relation with the above accesses, meaning that they can happen in arbitrary orders, happens-before$_{\rm MV}$ reports a bug.

## 5.3 Extensions to Other Detectors

Even though we only implemented and evaluated the multi-variable extensions to lock-set and happens-before, it is straightforward to follow the same idea to extend other concurrency bug detectors.

**Extending other race detection tools** Extending other dynamic race detectors, in particular those hybrid ones such as RaceTrack [43], is straightforward, since they combine the lock-set and happens-before algorithms. Extending static race checkers such as RacerX [8] and Chord [28] is even easier, since MUVI 's correlation information is collected from source code, and thus is much easier to feed to static checkers than to dynamic ones.

**Extending atomicity violation detection** Atomicity violation is another type of important concurrency bugs. It occurs when certain code region's programmer-intended atomicity, also called serializability [2], is not maintained during execution.

An important and challenging problem in atomicity violation detection is to infer which code regions are intended to be atomic. Existing solutions rely on either manual annotation [11, 27, 36] or inference based on *single*-variable-centric access patterns [25, 40].

The access correlations extracted by MUVI can serve for atomicity violation detection well. An access correlation essentially indicates that the correlated accesses need to be done atomically. Taking the concurrency bug in Figure 2 as an example, correlation `write(cache->empty)` ⇒`write(cache->table)` indicates that the writes to the two variables need to be atomic. The violation to this atomicity is exactly the root cause for that real bug from Mozilla.

The multi-variable atomicity information above can well complement existing tools like AVIO [25] and SVD [40]. These tools infer atomic regions based on 'code unit' composed of two consecutive accesses from one thread to *one* shared variable or read-write data dependency, respectively. Both of them would miss above atomic region composed of accesses to `cache->empty` and `cache->table`, where two different variables with no data-dependency are involved. In the following, we demonstrate how to extend AVIO to take advantage of the MUVI access correlation information.

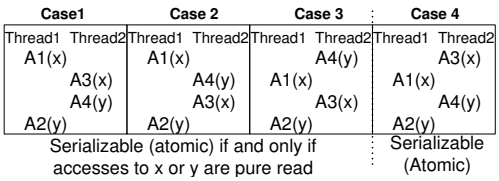


**Figure 5: Serializability of two variable access interleaving. The atomicity of above code regions composed of two accesses from one thread to correlated variables $x$ and $y$ is violated if interleaving follows case 1–3 with at least one write to both $x$ and $y$.**

The first step of our extension is straightforward: MUVI can extend AVIO's code-unit by including accesses, which have correlations among each other. Afterward, the challenge is to decide the atomicity (serializability) of concurrent accesses to *two* variables. This is much more complicated than the single-variable case in the original AVIO, because multiple variables create many more possible access interleaving combinations. Due to the space limit, we omit the serializability analysis details and simply list the atomicity condition results based on our analysis of 64 different access interleaving combinations in Figure 5.

# 6. EVALUATION METHODOLOGY

**Evaluated Applications** We have evaluated MUVI correlation analysis and inconsistent update bug detection using the **latest** versions of four applications: Linux (drivers), Mozilla , MySQL, and PostgreSQL as listed in Table 3.

| Application | Version | LOC | Description |
|---|---|---|---|
| Linux (drivers) | 2.6.20 | 3.6M | Operating System |
| Mozilla-Firefox | 2.0.0.1 | 3.4M | Web browser |
| MySQL | 5.2.0 | 1.9M | Database Server |
| PostgreSQL | 8.2.3 | 832K | Database Server |

**Table 3: Applications (latest versions) used in MUVI correlation analysis and inconsistency bug detection.**

In order to evaluate MUVI 's multi-variable concurrency bug detection capability, we use five known **real-world** [3] bugs from Mozilla and MySQL, as shown in Table 4.

**Platform** All of our experiments are conducted on a machine with a 2.4GHz Pentium processor, 512 KB L2 cache, 1GB of memory, running Linux 2.4.20 as the OS. We extend the EDG [14] compiler front-end for static code analysis.

**Parameter setting and sensitivity analysis** The default parameters in MUVI variable access correlation analysis are set as follows: *MinSupport* is 10, *MinDirectSupport* is 5, *MinConfidence* is 0.8, and *MaxDistance* is 10 lines of

[2]A property for several concurrently executed actions, when their data manipulation effect is equivalent to that of a serial execution of them.

[3]Since the race detectors are dynamic, we need to reproduce the bug during the execution to examine whether the detectors can catch them. But interestingly, we also found *four* **new** *multi-variable concurrency bugs* that have never been reported before.

| BugId | App. | Description |
|---|---|---|
| Moz-js1 | Mozilla-suite v0.9 | Wrong-ordered concurrent updates make empty table's *empty* flag false; leads to system crash (Figure 1) |
| Moz-js2 | Mozilla-suite v0.8 | Wrong-ordered read/write to `gcPoke` flag leads to reading wrong `liveAtoms`; makes garbage collection failure |
| Moz-imap | Mozilla-Thunder-bird v1.7 | Wrong-ordered concurrent updates make URL-in-progress flag true, but URL string NULL; system crash |
| MySQL-log | MySQL-v4.0.16 | Un-atomic read to log-file's name and log-file are interleaved by remote thread switching file; log-file failure |
| MySQL-blog | MySQL-v3.23.56 | Un-atomic table deletion and logging are interleaved by remote thread's table insertion and logging; security problem (Figure 8) |

**Table 4: Concurrency bugs tested with lock-set$_{MV}$ and happens-before$_{MV}$. It does not include the four *new* multi-variable concurrency bugs detected by our tools– one of them is shown on Figure 2(d); another will be shown in the next section.**

code. We choose these values based on our sensitivity analysis. In the next section, we show the parameter sensitivity study for two critical parameters, *MinSupport* and *MinConfidence*. We fix all other parameters using the default settings and change the targeting parameter (*MinSupport* or *MinConfidence*) to measure the accuracy of inferred access correlations. We will also conduct experiments to discuss how the parameter setting would affect the bug detection results and discuss the effect of our function call handling.

**Accuracy measurement** In our evaluation, we *separately* measure the accuracy (false positives and false negatives) of our variable access correlation analysis, inconsistent update bug detection and multi-variable concurrency bug detection.

# 7. EXPERIMENTAL RESULTS

## 7.1 Variable Access Correlation Analysis

Table 5 shows the variable access correlation analysis results. As we can see, variable access correlations are very common in real applications: totally 6449 access correlations (include only `AnyAcc`⇒`AnyAcc`) are inferred, with 5954 variables and 1467 structures involved. The analysis is efficient. For 3–4 million lines of code as Linux kernel has, it takes MUVI only 3 hours to infer 3353 access correlations.

To evaluate the accuracy of the correlations inferred by MUVI, it is too time-consuming to examine all 6449 correlations. Therefore, we take an approach similar to previous work [21] by randomly sampling 100 correlations from each application and manually verifying whether they are true. The results show that the false positive rate is reasonably low, around 17% on average.

| 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.**

The above results indicate that MUVI can efficiently and reasonably accurately infer access correlations, which can be stored in a database as a specification for reference by programmers or leveraged by other tools such as AutoLocker [27], DAIKON [10] and Colorama [4].

**False positives of access correlation inference** MUVI still has around 17% false correlations. They come from two major sources. (1) Macro and inlined functions are replicated by the compiler pre-processor and result in redundant supporters. Pruning these supporters requires special treatment of these macros and inlined functions. (2) In a few cases, variables are just coincidentally accessed together for many times, but there is no real correlation between them. This is especially true for some variables that get most of their support from read together. It is possible to prune out some of them by giving more weight to write-write patterns.

**False negatives of access correlation inference** Similar to previous work, MUVI is definitely not a panacea. It will miss variable access correlations in following cases: (1) true correlation with low supports. This is a common problem for almost all statistics based techniques [9, 22]. To solve this problem, we might need to look for other sources of correlation evidence. (2) *conditional correlation*. Some access correlations might only exist within certain program state (an example, Figure 8, will be discussed in Section 7.3). Current MUVI prototype can not distinguish program contexts and phases, and would miss such correlations.

## 7.2 Inconsistent Update Bug Detection

**Overall** Table 6 shows the inconsistent update bugs detected by MUVI. Out of the MUVI inconsistent update bug reports, we manually examined the top 100 ones, and identified 39 true bugs (17 of them have been confirmed by the corresponding open source developers). All of these bugs are **new bugs in the latest versions** of Linux, MySQL, Mozilla, and PostgreSQL. Almost all of the detected bugs are semantic bugs, and therefore cannot be detected by existing tools such as memory bug detection tools.

In addition to the two examples shown in Figure 1 in Introduction, here we show three more examples of recently confirmed bugs detected by MUVI in Figure 6.

**Bad programming practices** Besides true bugs, there are quite a few violations that are bad programming practices that do not cause problems now but can easily introduce bugs later. For example, in PostgreSQL, pointer `PGconn::inStart` points to the starting point of a message, pointer `PGconn::inCursor` is used as a cursor pointing to a position inside a message during the message reading. In one function, the current message is discarded and therefore `PGconn::inStart` is moved to the next message, but

| App. | #MUVI Bug Report | #New Bugs Found | #New Bugs Confirmed | #Bad program-ming | #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.**

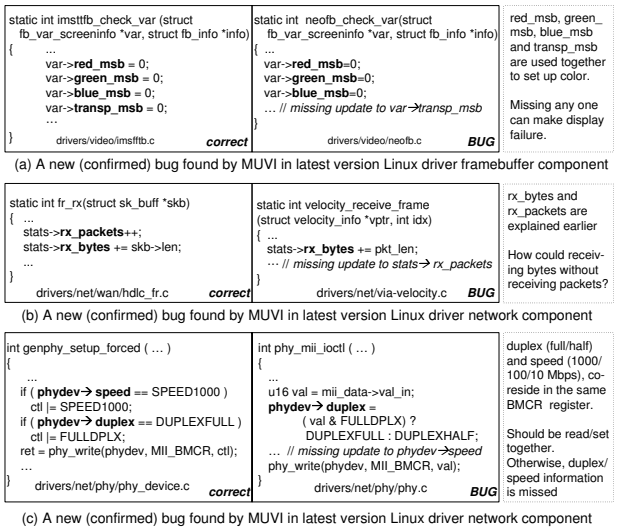


**Figure 6: Examples of *new* inconsistent update bugs detected by MUVI in the latest version of Linux. They have recently been confirmed by the developers.**

`PGconn::inCursor` is not changed, still pointing inside the discarded message. Fortunately, in all other places in the program, the use of `inCursor` is always preceded by a checking of `inStart`, therefore such dangerous inconsistent update does not lead to a bug. However, future code revision may easily introduce a bug as a programmer may assume that these variables are always consistent. Therefore, such cases reported by MUVI can help programmers to clean up the code and improve the software quality.

**False positives of inconsistent update bug detection** Although MUVI has pruned some false alarms using interprocedural analysis and confidence filtering, there are still some false positives caused by the following reasons (the breakdowns are also shown on Table 6):

(1) Semantic exceptional cases. Even if the correlations are correct, they can still be violated in cases of special semantic requirements. For example, in Mozilla, `nsHTMLReflowMetrics::height` and `nsHTMLReflowMetrics::width` denote the height and width of an HTML object. They are always read and updated together. However, in function `AdjustForCollapsingCols`, since only col(umns) are collapsed, the program only updates width, but not height.

(2) Wrong correlations. All of the correlations inferred by MUVI are directly fed to bug detection, so wrong correlations result in around one third of the bug false positives.

(3) No future reads. It does not cause problems when a function modifies a variable without accessing the correlated peers if there is no future read to that variable. However, such an assumption about no future read needs to be carefully maintained. Therefore, such false positives can be treated as warnings as they are still helpful to programmers.

Among the above three sources of false positives, it is the easiest to solve the issue (3). False positives caused by it can be pruned by some compiler analysis, such as liveness analysis. Pruning false positives caused by issue (2) requires improving the accuracy of access correlation inference as discussed earlier. The issue (1) contributes the most to the false positives in our experiments. Solving this issue requires automatic inference of special program semantics, which is very challenging and remains as our future work.

| 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.** | | | | | | |

**Table 7: Dynamic multiple-variable concurrency bug detection results. (In detect-bug columns, 'Y' means correct root cause identified; 'N' means not identified. None of these five bugs' correct root causes can be identified without MUVI extension. The numbers of false positives are the *additional* static false positives introduced by our extensions (excluding those introduced by the original race detectors). *: The overhead is the *extra* monitor-run overhead over the original race detectors.)**

**False negatives of inconsistent update bug detection** The false negatives of MUVI inconsistent update bug detection would come from two main sources: (1) some true correlations are missed by MUVI access correlation analysis; (2) some true bugs might be ranked low in MUVI bug reports and are therefore missed by programmers. In our current prototype, this happens when there is relatively big number of violations or small number of supports. Part of this problem can be solved by better ranking algorithms, which we will study in the future.

## 7.3 Concurrency Bug Detection

**Overall** Table 7 shows the evaluation results on five real-world multi-variable concurrency bugs. Both lock-set$_{MV}$ and happens-before$_{MV}$ can correctly identify the root causes of four tested multi-variable concurrency bugs. None of these tested bugs' true root causes, i.e. multi-variable concurrency bugs, can be identified by the original lock-set or happens-before algorithms without MUVI's extensions.

Furthermore, MUVI also detects four *new* multi-variable concurrency bugs that have never been reported before. We have already shown an example in Figure 2(d). Figure 7 shows another new bug we find from Mozilla. In this bug, the function in thread 1 is invoked with lock protection, but the function in thread 2 is not. Actually, were not the correlation between `table->entryCount` and `table->removedCount`, the single-variable race between the two threads seemed benign. But with MUVI 's multi-variable extension, the race detectors correctly identify the problem: the consistency of the *correlated* accesses is broken by remote conflict accesses.

**False positives of multi-variable concurrency bug detection** MUVI extension also introduces a small number (0-6) of false positives. Since the original lock-set algorithm reports races more aggressively than happens-before, our extension introduces slightly more *additional* false positives to happens-before than to lock-set.

MUVI 's false positives come from two sources: wrong correlations and benign multi-variable races. Like benign single variable races, benign multi-variable races also exist due to special program semantics. Some of the benign multi-variable races can be pruned by sophisticated atomicity violation analysis like what we did for AVIO$_{MV}$ in section 5.

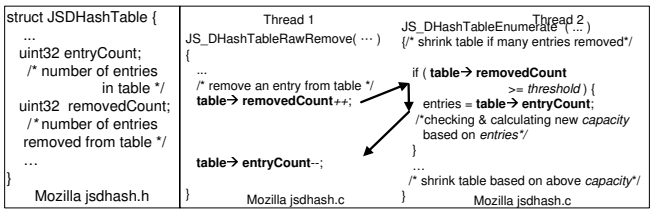


**Figure 7: A *new* multi-variable concurrency bugs found by MUVI in Mozilla. Thread 2 interleaves thread 1's update to `table->entryCount` and `table->removedCount` and reads inconsistent values. As a result, the `table` may not be correctly shrunk. If only considering the race on each single variable, the programmer can easily get confused. However, benefiting from MUVI 's multi-variable extensions, the detectors can identify the correct root cause.**

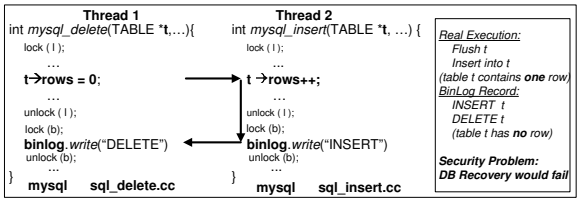


**Figure 8: The false negative (bug MySQL-blog in Table 4) of Lock-set$_{\text{MV}}$ and Happens-before$_{\text{MV}}$. The correlation between `t→rows` and `binlog` is *conditional*, and is therefore missed by MUVI. Specifically, t→rows can be accessed many times, and usually binlog need not be accessed together. Only when a request's processing is at the end and t→rows is given a final value, does binlog need to be consistently modified.**

**False negatives of multi-variable concurrency bug detection** MUVI extension misses one tested bug (bug MySQL-blog), as shown in Figure 8. The reason of this false negative is that MUVI can not detect the *conditional correlation*, which only holds within certain program contexts, associated with this bug. How to combine program contexts with variable correlations remains as future work.

**Performance** Our MUVI extension only adds a small percentage of extra overhead on the original race detectors: 5.9%-40% for lock-set; and 1%-21% for happens-before. Note that, due to the completely different implementations (one using Valgrind and the other using PIN), the extra overhead of lock-set$_{\text{MV}}$ and happens-before$_{\text{MV}}$ are incomparable. Since the original implementations already incur too large overheads (more than 10X) to be used in production runs, our additional overheads have no major impact.

## 7.4 Distribution of Variable Correlations

**How many correlated peers?** Figure 9 shows the distribution of the variables with different numbers of correlated peers. The results from Linux and Mozilla show that most variables are only correlated with a small number of peers: around half of the variables are correlated with only one variable and around 20% are correlated with two variables.

This result indicates that *access correlations do not exist between any two random variables*. Even though most structures contain many fields, only those fields that have true semantic connections have access correlations.

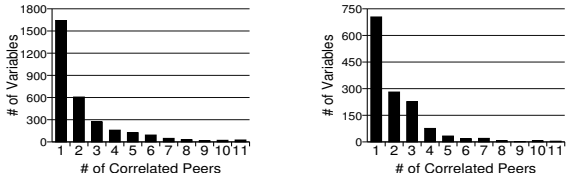


**Figure 9: Distribution of correlated variables with different number of peers in Linux and Mozilla. The results with the other two applications are similar.**

**How many correlations in each type?** Table 8 shows the number of correlations in each type. Around half of the correlations are asymmetric and therefore we need to differentiate the direction in correlation analysis. Table 8 also shows other types of access correlations MUVI finds. The distribution shows that most of the correlations are either read together or written together. read ⇒write and write ⇒read are relatively rare. Variables with these two types of correlations usually also support read ⇒read or write ⇒write. This indicates that the correlated variables should be either used or updated consistently.

| | Sym. | Asym. | rr | rw | wr | ww |
|---|---|---|---|---|---|---|
| Linux | 1595 | 1758 | 1141 | 325 | 408 | 1113 |
| Mozilla | 651 | 780 | 697 | 151 | 237 | 341 |
| MySQL | 316 | 410 | 339 | 61 | 81 | 161 |
| PostgreSQL | 586 | 353 | 365 | 112 | 131 | 269 |

**Table 8: Direction and access types of correlations. (rr is read ⇒read, rw is read ⇒write, and so on. Due to the space limit, the remaining four types are not shown.)**

## 7.5 Sensitivity Analysis

In order to demonstrate how to choose parameters in MUVI, we perform a sensitivity study on *MinConfidence* and *MinSupport*. We only show the results of MySQL variable correlation inference in Figure 10. The other applications also share similar trends.

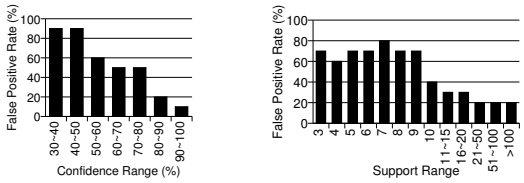


**Figure 10: Parameter sensitivity (MySQL) (For each confidence or support threshold value, the false positive rates are measured by examining 10 randomly selected access correlation candidates with the specified *confidence* and *support* values).**

Figure 10(a) shows how the false positive rate of MUVI variable correlation inference changes with different *MinConfidence* (all other parameters are fixed as default values). The false positive rate dramatically decreases from higher than 50% to around 20% when the confidence reaches 80%. Therefore, we choose 0.8 as the default *MinConfidence*.

Figure 10(b) shows how the false positive rate of MUVI variable correlation inference changes with different *MinSupport* (all other parameters are fixed as default values). Similarly, we can see the dramatic change of false positive

rate around the support range of 10, and therefore we choose 10 as the default *MinSupport*.

Parameter setting also affects the bug detection results. Here, we show the results from MUVI inconsistent update bug detection on Linux and MySQL based on different *MinDirectSupport* for demonstration. Comparing against the bugs detected by MUVI under the default setting (*MinDirectSupport*=5), Figure 11 shows the number of bugs detected under different *MinDirectSupport* (all other parameters use default values). More bugs would be missed with larger *MinDirectSupport*. For example, with *MinDirectSupport* 30, only 1 (out of the total 22) Linux inconsistent update bug and 3 (out of the total 9) MySQL bugs can be detected.

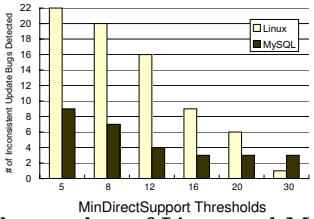


**Figure 11: The number of Linux and MySQL inconsistent update bugs that are detected at different** *MinDirectSupport* **(here we use the set of bugs detected by MUVI default setting as the baseline)**

Apart from the parameter setting, other MUVI design also needs to consider the false positive–negative tradeoff. In our current prototype, not only direct accesses but also indirect accesses of one-level callee functions are considered in the Acc_Set (Section 3.2). Although this design choice results in a few more false positives, it also allows us to extract more true correlations. Take MySQL variable correlation inference as an example. Our experiment shows that MUVI can infer 51 more true correlations [4] than the alternative design where only direct accesses are considered (all other settings are exactly the same), with a reasonable number (15) of additional false positives.

# 8. RELATED WORK

**Inferring specifications from programs** Many studies have been conducted on automatically extracting specifications from programs [1, 2, 20, 22, 38, 39, 42], but most of them focused on procedures and component interfaces instead of variable correlations. For example, Kremenek et al. [20] use probabilistic graph models to infer properties of a function such as ownership, allocator, deallocator, etc.

A study conducted by Engler et al. [9] shares the same high level idea with our work. But their work focuses on detecting inconsistency via *logic reasoning*: for example, some statement indicates that a pointer might be NULL (since it performs a check) but a subsequent statement assumes that this pointer must not be NULL (since it references the pointer directly)—so a conflict. It does not perform pattern analysis except for one type: "function A should be paired with function B". Some other recent studies have been conducted to extract invariants such as value invariants [10, 15] and failure-predicates [23].

Unlike above work, our work detects inconsistency via pattern analysis on multi-variable access correlations. In addition, we also detect multi-variable concurrency bugs and extract multi-variable correlations that can be used to annotate programs for other tools.

**Code mining for patterns** MUVI is also related to previous work that uses data-mining to detect software bugs. DynaMine [24] detects common function API usage patterns by mining software patches. CP-Miner [21] applies data mining to identify copy-pasted code and further detects copy-paste-related wrong variable-name bugs. PR-Miner [22] also uses data mining techniques to find rules, specifically, frequent function call sequences from source code, and then detects bugs caused by missing or wrong *function* calls.

Similarly, MUVI also uses data mining to infer patterns of different types—multi-variable access correlations, and then detect completely different classes of software bugs—multi-variable inconsistent updates and related concurrency bugs.

**Concurrency bug detection** As discussed in earlier sections, much work has been done in detecting data races [5, 6, 8, 28, 30, 31, 32, 35, 37, 43] and atomicity violations [11, 25, 34, 40]. Almost all of them focus on *single*-variable data races or atomicity violations. Model checking is also widely used for concurrency bug detection [12, 17, 33]. Model checking tools first abstract the program state and then systematically explore the whole state space to disclose bugs. In order to shrink the state space, many model checking tools assume that accesses to different variables are *independent* of each other. Therefore, they will similarly miss multi-variable concurrency bugs.

As demonstrated in our extensions to lockset and happens-before, MUVI can help previous concurrency bug detectors to detect multi-variable concurrency related bugs.

**Techniques for easing concurrent programming** Much work has been done to ease the implementation of concurrent programs. Recently, transactional memory [16, 18] is a widely-recognized approach along this direction. While it can significantly eliminate single-variable races, multi-variable concurrency bugs can still occur. AutoLocker [27] proposes using compilers to add locks automatically. It assumes that programmers need to *manually* annotate correlated variables to use the same lock.

MUVI well complements the above work since it can automatically infer variable correlations and provide this information to the above tools.

# 9. CONCLUSIONS AND FUTURE WORK

This paper proposes an innovative approach, MUVI, that combines source code analysis and data-mining techniques to automatically infer variable access correlations and detect related bugs. MUVI extracts 6449 access correlations from Linux, Mozilla, MySQL and PostgreSQL with high (83%) accuracy. Based on these correlations, MUVI detects 39 new bugs (17 already confirmed) from these applications. MUVI extensions to two representative existing race detectors (lock-set and happens-before) correctly identify the root causes of four tested real-world multi-variable concurrency bugs and also detect four new multi-variable concurrency bugs that have never been reported before.

Our results indicate that multi-variable access correlation is a common and important program semantic property in various real-world programs and their violations can cause important semantic bugs in operating system and server code. The access correlations inferred by MUVI can be used to automatically annotate programs for other tools such as AutoLocker [27] and Colorama [4].

[4]Based on our manual examination.

Our work is only a beginning in the multi-variable correlation research. It can be extended in four aspects: (1) Detect other types of multi-variable related bugs, such as read inconsistency, multi-variable atomicity violation bugs, etc. (2) Improve MUVI correlation analysis and bug detection accuracy via better code analysis. (3) Extend MUVI to analyze dynamic traces to get run-time correlation. (4) Evaluate more real-world applications.

## 10. ACKNOWLEDGMENTS

The authors would like to thank the shepherd, Stefan Savage, and the anonymous reviewers for their invaluable feedback, Professor Liviu Iftode for helpful discussions, the Opera group members for discussions and paper proofreading. This research is supported by NSF CCF-0325603 grant, NSF CNS-0615372 grant, NSF CNS-0347854 (career award), DOE Early Career Award DE-FG02-05ER25688, and Intel gift grants. Shan Lu was awarded an SOSP student travel scholarship, supported by Google, to present this paper at the conference. The work by Raluca A. Popa was conducted while she worked at University of Illinois as a part of the CRA-W DMP project.

## 11. REFERENCES

[1] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *POPL*, 2005.
[2] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, 2002.
[3] C. Artho, K. Havelund, and A. Bierre. High-level data races. The First International Workshop on Verification and Validation of Enterprise Information Systems, 2003.
[4] L. Ceze, P. Montesinos, C. von Praun, and J. Torrellas. Colorama: Architectural support for data-centric synchronization. In *HPCA*, 2007.
[5] J.-D. Choi et al. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, 2002.
[6] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *PPoPP*, 1990.
[7] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In ACM/ONR Workshop on Parallel and Distributed Debugging (AOWPDD), 1991.
[8] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, 2003.
[9] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, pages 57–72, 2001.
[10] M. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE*, 2000.
[11] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, 2004.
[12] P. Godefroid. Model checking for programming languages using verisoft. In *POPL*, 1997.
[13] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proceeding of the First IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, Nov 2003.
[14] E. D. Group. EDG C/C++ front end.
[15] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, 2002.
[16] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA*, 2003.
[17] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI '04*, 2004.
[18] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, 1993.
[19] S. T. King, G. W. Dunlap, and P. M. Chen. Operating systems with time-traveling virtual machines. In *Usenix Annual Technical Conference*, 2005.
[20] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *OSDI*, Nov 2006.
[21] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *OSDI*, 2004.
[22] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *FSE*, Sept 2005.
[23] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.
[24] V. B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *FSE*, 2005.
[25] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *ASPLOS*, 2006.
[26] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.
[27] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In *POPL*, 2006.
[28] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI*, 2006.
[29] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *ENTCS*, 2003.
[30] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *PPoPP*, 1991.
[31] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP*, 2003.
[32] D. Perkovic and P. J. Keleher. Online data-race detection via coherency guarantees. In *OSDI*, 1996.
[33] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *PLDI*, 2004.
[34] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *PPoPP*, 2005.
[35] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM TOCS*, 1997.
[36] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006.
[37] C. von Praun and T. R. Gross. Object race detection. In *OOPSLA*, 2001.
[38] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *TACAS*, 2005.
[39] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA*, 2002.
[40] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, 2005.
[41] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, 2006.
[42] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *ICSE*, 2006.
[43] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, 2005.