A Serializability Violation Detector for Shared-Memory Server Programs

Min Xu
Electrical & Computer Engr. Dept.
University of Wisconsin-Madison
mxu@cae.wisc.edu

Rastislav Bodík
Computer Science Division, EECS
University of California, Berkeley
bodik@eecs.berkeley.edu

Mark D. Hill
Computer Sciences Dept.
University of Wisconsin-Madison
markhill@cs.wisc.edu

Abstract

We aim to improve reliability of multithreaded programs by proposing a dynamic detector that detects potentially erroneous program executions and their causes. We design and evaluate a *Serializability Violation Detector* (SVD) that has two unique goals: (I) triggering automatic recovery from erroneous executions using backward error recovery (BER), or simply alerting users that a software error may have occurred; and (II) helping debug programs by revealing causes of error symptoms.

Two properties of SVD help in achieving these goals. First, to detect only erroneous executions, SVD checks serializability of atomic regions, which are code regions that need to be executed atomically. Second, to improve usability, SVD does not require *a priori* annotations of atomic regions; instead, SVD approximates them using a heuristic. Experimental results on three widely-used multithreaded server programs show that SVD finds real bugs and reports modest false positives.

Categories and Subject Descriptors. D.2.5 [**Software Engineering**]: Testing and Debugging—*Diagnostics*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*;

General Terms. Algorithms, Languages, Reliability

Keywords. Multithreading, Serializability, Race Conditions

1. Introduction

1.1. Objective

In shared-memory programs, bugs often manifest themselves only under specific thread interleavings, and sometimes at the worst time. For example, the 2003 U.S.-Canada power outage went unnoticed for a crucial period of time; the reason was a data race in the grid-monitoring system [28,35].

Fortunately, the non-deterministic nature of these timing-dependent bugs can be exploited. If one can detect erroneous executions on-the-fly, then backward error recovery (BER) can be used to roll back a part of the erroneous execution; subsequent reexecution with a conservative thread scheduling may avoid recurrence of the software error. When BER is not available, detecting erroneous executions can alert that a software error may have happened. In the 2003 blackout, an early warning may have prompted the power

grid operators to reboot the grid-monitoring system, which would then warn about the power outage before it had become a disaster.

The goal of this paper is to develop a detector suitable for (I) BER-based avoidance of erroneous program executions; and (II) alerting users as software errors occur. We argue that such a detector should have the following two properties.

Detect *only* erroneous executions. Typical detectors strive for best coverage by detecting *potential* bugs, i.e., bugs that may manifest in executions not directly examined. In contrast, our detector should detect only erroneous program executions, because it is to be deployed in the following scenarios:

- *Bug avoidance with BER.* Imagine a detector with low overhead, perhaps implemented in hardware. When an erroneous execution is detected, the execution rolls back to a safe checkpoint and reexecutes (more) serially [30,34]. In this scenario, detecting only erroneous executions helps reducing the performance lost in unnecessary rollbacks. Similarly, detecting only manifested errors helps avoid overloading operators with false alarms. Because unnecessary rollbacks or alarms occur on each instance of a *false positive*, the detector should strive to reduce *dynamic false positives*, which include dynamic instances of identical warnings.

- *From symptoms to bugs.* Imagine we have captured a failing multithreaded execution with a deterministic recorder [4,29,38]; how do we now find the bug in the execution? Replaying the execution with the detector will point to an error that actually happened in this execution; this error is likely the cause of the failure. Detecting causes of erroneous executions avoids reporting errors that may exist in some other executions, thus improving understanding of the execution at hand. In this scenario, the detector should strive to reduce *static false positives*, which are false warnings related to the same piece of code.

Do not require *a priori* program annotation. Typical detectors require *a priori* program annotations, such as identification of synchronization constructs or annotations of atomic regions, which are code regions that need to be executed atomically. The annotation effort is non-trivial for server programs, because the source code is large and sometimes not fully available, especially for commercial software. Instead of requiring *a priori* annotations, we believe that a dynamic detector that enables *a posteriori* examinations is more applicable to server programs, because the examination is limited to the program trace of a single execution.

1.2. Our Solution

We propose a new detector, called *Serializability Violation Detector* (SVD), that seeks to detect only erroneous program executions

and does not require *a priori* program annotation. Three ideas underlie the design of SVD.

1) **Computational units.** SVD automatically infers approximations of atomic regions. We call the inferred atomic regions *Computational Units (CU's)*, because the inference heuristic relies on computational patterns, namely data and control dependences. Because our heuristic inference does not rely on any synchronization constructs, it identifies atomic regions even when locks are mistakenly omitted in the example shown in Section 2.1. Thanks to the inference, no *a priori* annotation is needed.

2) **Serializability.** SVD detects erroneous executions by determining whether the execution violated serializability of the inferred CU's. Traditionally, serializability is defined for database transactions. In databases, serializability means that the execution of a group of transactions is *logically* equivalent to a serial execution of the same group of transactions. In shared-memory programs, we apply serializability to atomic regions, which are approximated by inferred CU's. A serializable execution is correct, because it appears that each atomic region executed atomically. Executions that are not serializable are often erroneous. We use an efficient yet approximate serializability test, which ensures that the memory locations read by a CU are not overwritten by another thread before the CU ends.

3) *A posteriori* **examination.** Because inferred CU's may differ from the atomic regions, SVD is able to produce a log of CU's for the programmer to examine *a posteriori*. This way, programmers can discover erroneous executions that are missed online by SVD due to inherent limitations of inferring CU's without any *a priori* annotations.

We design and evaluate SVD in a post-mortem debugging scenario with deterministic replay. Our results show that SVD helps find erroneous executions caused by timing-dependent bugs in Apache [3] and MySQL [23], without requiring *a priori* program annotations. Experimental results on these two server programs show that SVD reports *far fewer* dynamic false positives and *modestly fewer* static false positives than a data race detector. Experimental results on PostgreSQL [27], a relatively mature server program, show that although SVD reports more false positives than the data race detector, the absolute false positives rate is low. We contribute in the following aspects.

- We propose a novel detector that seeks to detect only erroneous program executions caused by timing-dependent bugs, without requiring *a priori* program annotations.

- We propose a novel method for approximately inferring atomic regions.

- By applying the detector to large shared-memory server programs, we show the new detector is suitable for avoiding (unknown) bugs with BER.

The rest of this paper is organized as follows. In Section 2, we use three examples to illustrate the key ideas of SVD. In Section 3 we formalize SVD. In Section 4, we present two versions of SVD algorithms and outline a hardware implementation. We qualitatively analyze SVD in Section 5. After describing our evaluation methodology in Section 6, we evaluate SVD in Section 7. We present related work in Section 8 and conclude in Section 9.

2. Overview of SVD

2.1. Inferring Computational Units

To understand how SVD operates, it is useful to pretend first that programs come with *a priori* annotations that specify code regions to be executed atomically. We call these code regions *atomic regions.* The programmer implements atomic regions with lock-based critical sections or by means of some other synchronization mechanisms, such as signal, monitor or thread fork. A program is buggy when some atomic regions are not implemented correctly, e.g., when their critical sections are placed incorrectly or are entirely missing. SVD seeks to detect if a bug in an incorrect implementation of an atomic region manifested itself in a given execution. SVD performs the detection by verifying that atomic regions were executed in a serialized way.

Because we assume that *a priori* annotations of atomic regions are actually not available, our approach is to infer these regions. The key feature of our inference is that it does not rely on the synchronization mechanisms used in the program; such inference would likely infer atomic regions that were as buggy as the synchronizations, whose bugs we want to identify in the first place. Instead, SVD infers atomic regions from how shared variables are used and from data and control dependences.

The result of SVD inference are computational units (CU's), which approximate dynamic instances of atomic regions. That is, CU's are execution paths through atomic regions. (Henceforth, atomic regions refer to dynamic instances of atomic regions.) As we shall show, SVD computes CU's online, automatically, without requiring *a priori* program annotations, and without being affected by (incorrect) synchronization constructs.

Informally, a CU is the largest group of dynamic program statements that follow the following two-part *region hypothesis*.

1) A shared variable written in an atomic region is not read again in the same atomic region. In other words, true (read-after-write) dependences through shared variables between statements happen only across atomic regions, not within them.

2) Program statements are related through true dependences or control dependences in atomic regions. In other words, an atomic region does not perform multiple unrelated computations. The precise meaning of "related" is given in Section 3.2.

To evaluate the power of region hypothesis, we manually examined 14 atomic regions from real programs. We observed that region hypothesis held on the common paths of all 14 regions. For example, it holds for the atomic region in JDK 1.4 StringBuffer class, which contains a subtle synchronization bug [16]. It did not hold on some rare paths. We discuss in Section 2.3 how to deal with these limitations using *a posteriori* examinations.

Figure 1 shows an example of a CU. The simplified code in Figure 1(a) is taken from the MySQL database server, which uses file system locks to guard database tables. In the figure, one such lock is info→internal_lock. SVD correctly infers the atomic region, which is implemented correctly as a critical section guarded by info→internal_lock. The inferred CU contains four statements, represented in the figure as an oval. To understand the inference, observe first that the shared variable info→tot_lock is read and subsequently written in the CU, but it is not read in the CU again. Second, note that the four statements are related via dependences as shown in Figure 1(b). Statements 1.05, 1.06 and 1.07 are control-dependent on 1.03, and statement 1.06 is true-
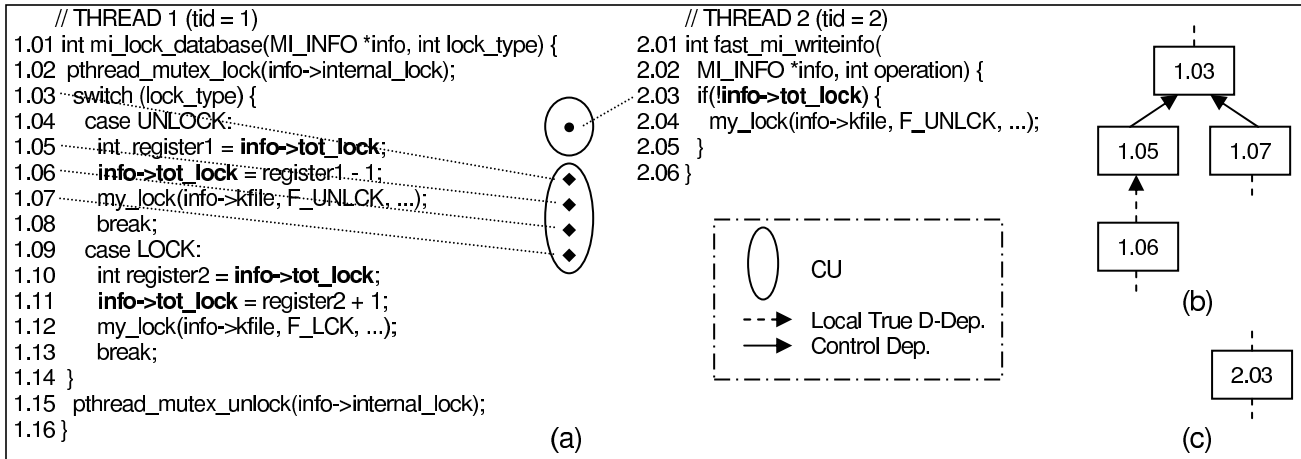
Figure 1: SVD computes CU's by observing dependences between program statements; SVD avoids reporting some false positives that are reported by race detectors. (a) MySQL table locking code. Inferred CU's are big enough to cover atomic regions implemented correctly with the lock. The code also contains a harmless data race on shared variable info→tot_lock. While race detectors will report this false positive, SVD will not, because CU's in the execution are serializable. (b) and (c) Data and control dependences between statements within a CU. Note that only statements that are executed are included in a CU.

dependent on 1.05 via a local variable register1. (The true-dependence predecessor of statement 1.03 is not in the CU because it belongs to the preceding CU; see Section 3.2). Note that SVD performs the inference without using the lock info→internal_lock. (The shared variable info→tot_lock is not a lock, despite what its name may suggest.)

To illustrate how SVD computes a CU in a buggy program, consider the example in Figure 2, where a lock is mistakenly omitted. The simplified code shown in Figure 2(a) is taken from the log_config module of the Apache web server. The log_config module buffers log messages, generated from multiple threads, in a shared memory buffer before writing them to a file. In this shown execution, two threads execute function ap_buffered_log_writer() simultaneously. Variable len is thread-local. Thread-local pointer buf points to a shared memory buffer

(buf→bufout) and a shared index variable (buf→outcnt). In order to avoid corrupting data in buf→bufout, the memcpy() operation (3.08) and the update to buf→outcnt (3.09) should be guarded within a critical section, which is not implemented.

Figure 2(a) shows the CU's of the two threads. (The broken oval shows the serializability is violated.) The shared variables are shown in bold. Figure 2(b) shows that statements 3.05, 3.08 and 3.09 are true-dependent on statement 3.04 via the thread-local variable len. Figure 2(c) shows a similar CU of thread 2. Note that the inferred CU's include statements from the atomic region, even though the programmer did not implement them correctly (the locking is missing).

So far, we have not discussed the actual inference algorithm. SVD partitions a thread execution into CU's by following dependences in execution order. It groups related instructions into a CU, starting
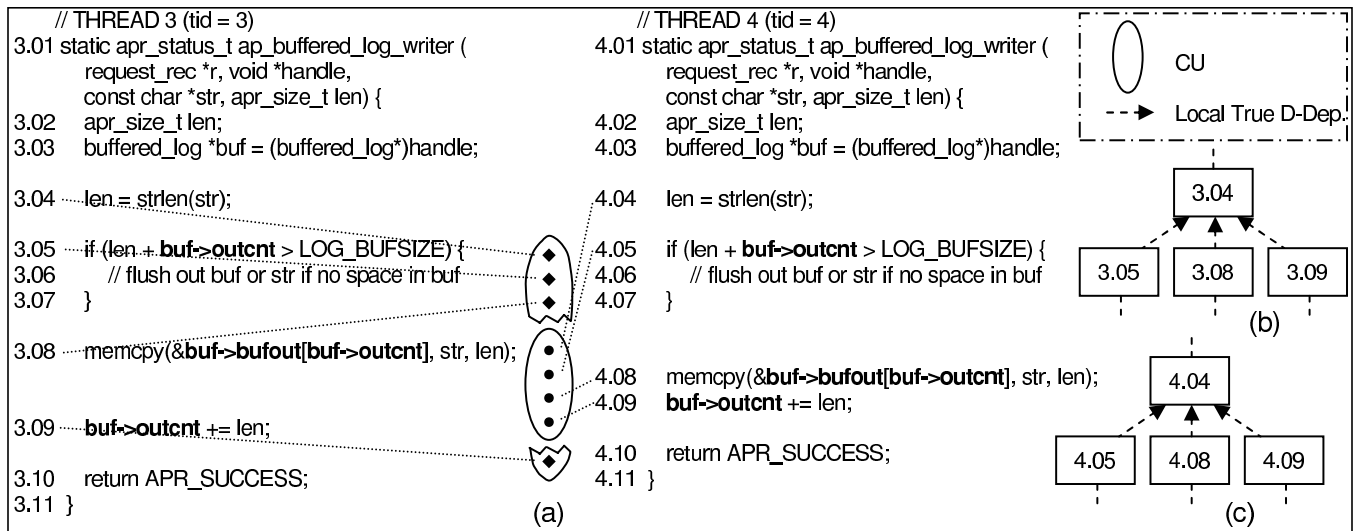

Figure 2: SVD computes CU's via dependences even when buggy code contains incorrect synchronization; SVD detects erroneous program executions. (a) Apache's log_config module contains a data race that corrupts the log messages stored in a shared buffer. SVD detects when corruptions happen by observing serializability of CU's is violated. (b) and (c) Data and control dependences between statements within a CU. Note that only statements that are executed are included in a CU.
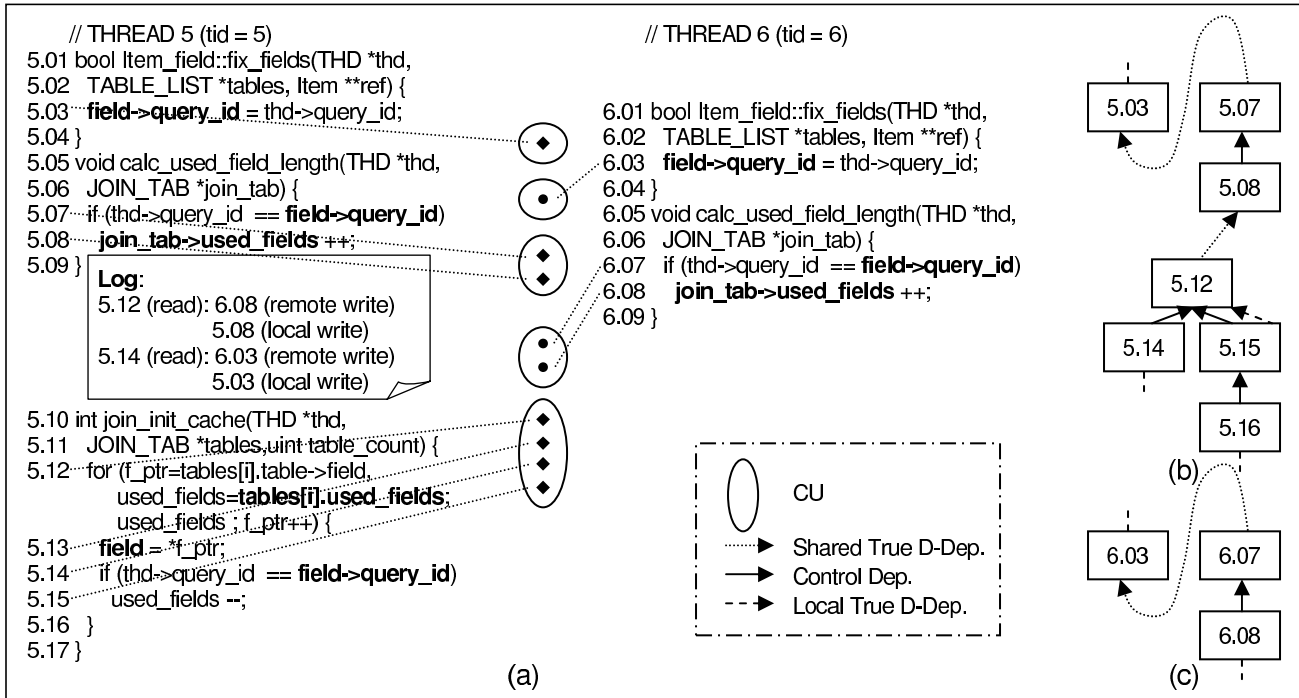
Figure 3: SVD can miss erroneous executions (i.e. false negatives), but it generates a log for an *a posteriori* examination to mitigate the problem. The buggy code is taken from MySQL, which executes prepared SQL queries. (a) SVD stops growing CU's when shared dependences are observed. Variables `field→query_id` and `join_tab→used_fields` are *mistakenly* shared. Because true dependences are observed on the shared variables, several small CU's are computed according to the second rule of region hypothesis. Therefore, SVD fails to detect this erroneous execution online. We also show a log generated by SVD that enabled an *a posteriori* examination, which discovered this bug. (b) and (c) Data and control dependences between statements within a CU. Note that only statements that are executed are included in a CU.

a new CU whenever it encounters a read from a shared variable. SVD uses a heuristic to find shared variables. A variable is shared if it is accessed by more than one thread *after* it is accessed by a CU and *before* the CU ends. Figure 1 and Figure 2 do not show shared dependences that end CU's. Yet one can imagine when one of the shared variables (info→tot_lock and buf→bufout, buf→bufcnt) is read back by a statement *s*, SVD concludes that a CU ends just before *s*. The end of the CU is conservative, because the atomic region may have ended earlier than *s*. Figure 3 in Section 2.3 shows some examples of shared dependences.

2.2. Detecting Serializability Violations

After SVD computes CU's, it reports erroneous program executions whenever CU's are not serializable. To detect serializability violations, SVD observes the *temporal order* between memory accesses from different threads (i.e., thread interleaving) and *conflicts* between the accesses. Two accesses conflict if and only if they access the same variable from different threads and at least one access is a write.

2.2.1 Finding Erroneous Program Executions

To check serializability, SVD uses a heuristic that tests if conflicts have happened on *input variables* of a CU. Input variables are locations not written within the same CU before its first read by the CU. This check is performed whenever a CU performs a write. This heuristic is a relaxation of a conservative serializability detection algorithm presented in Section 3.3. It helps SVD achieve low false positive rate by allowing unlikely false negatives (Section 4).

Figure 2(a) shows an erroneous execution of the Apache web server found by SVD. The diamond and dot patterns shows a logical interleaving of statements from the two threads. Note that we can freely reorder those statements that do not conflict, but not those statements that conflict on shared variables buf→bufout and buf→outcnt. Due to the conflicts on shared variables, the execution of the CU of thread 3 is "broken" by thread 4. SVD detects the serializability violation when 3.09 is writing buf→outcnt by observing a conflict (3.05 vs. 4.09).

2.2.2 Avoiding False Positives w.r.t. Race Detection

Another method for detecting erroneous executions is data race detection. A data race occurs when two threads access the same variable with no *synchronization* between the accesses, where at least one of the accesses is a write [24]. Serializability helps SVD avoid some false positives reported by race detectors, because a program execution can be *serializable* and at the same time contain data races. Although SVD does report false positives that are not reported by race detectors, we found SVD usually reports fewer false positives than race detectors (Section 7.2).

Figure 1(a) shows a correct execution of the table locking code in MySQL that contains data races. Thread 1 updates info→tot_lock within a critical section (guarded by mutex info→internal_lock). Thread 2 reads info→tot_lock without first synchronizing with thread 1. Data races occur due to statements 1.06, 1.11 and 2.03. Existing race detectors would conclude the execution shown in Figure 1(a) is erroneous because of the data races.

However, the data races do not indicate an erroneous MySQL execution; i.e., existing race detectors would report false positives. The code in thread 1 implies info→tot_lock is never zero for shared tables, because shared tables must be locked before they are used and info→tot_lock is initially zero. In other words, the predicate of statement 2.03 is never true for shared tables.[1] Therefore, the data races are not harmful. We found that it requires non-trivial time and effort, even by a programmer who is familiar with MySQL, to determine the races do not indicate a bug.

SVD *avoids* reporting the false positives by observing that serializability is *not* violated in Figure 1(a). In this execution, both CU's are serializable.

2.3. Logging for *A posteriori* Examination

Accuracy of SVD is largely determined by how closely CU's approximate atomic regions. When CU's are larger than atomic regions, SVD may report false positives. When CU's are smaller, SVD may miss erroneous executions (false negatives). This section describes how we mitigate the false negative problem.

SVD logs each statement *s* that reads a variable last written by another thread; it also logs the remote write *rw* and the immediately preceding thread-local write *lw* to the same variable. The triple *(s, rw, lw)* records that *rw* overwrote the value that *lw* may have intended to communicate to *s*. If this local communication is indeed intended, then we find a likely bug. The programmer examines the log in *a posteriori* examination. The examination allows discovering, in a post-mortem fashion, more erroneous executions than SVD can do online. The statement *s* is an input to the CU, so the log effectively records "shapes" of inferred CU's.

For example, Figure 3(a) shows an erroneous execution caused by a MySQL bug that we found during an *a posteriori* examination. Each MySQL query is carried on by a single thread. During executions of MySQL prepared SQL queries, two variables (field→query_id and join_tab→used_fields), which are intended to be thread-local, are shared between threads by mistake. The variable field→query_id is intended to distinguish those fields of a database table (join_tab) that are used by a SQL query. The variable join_tab→used_fields is used to record the total number of fields used by the query. During the execution, thread 5 initially computes number of fields that it uses (5.08). Then, thread 6 updates the values of field→query_id (6.03) and join_tab→used_fields (6.08), which later causes the loop of thread 5 (5.12) to go out-of-bounds based on an inconsistent value of join_tab→used_fields. This bug crashes MySQL server with a segmentation fault.

Because the thread-local variables are mistakenly shared, in order to detect erroneous executions of MySQL, we must detect serializability violations for those atomic regions that provide mutual exclusion to the accesses to these variables. However, variables like field→query_id and join_tab→used_fields are read back within the atomic regions, violating the second rule of region hypothesis. SVD fails to report erroneous executions caused by this bug, because it forms CU's smaller than the atomic regions. Unfortunately, serializability for these small CU's is not violated during the execution as shown in Figure 3(a). Figure 3(a) shows a log that contains statements 5.12 and 5.14. When a programmer

---

1. The predicate of statement 2.03 can be true for thread-local temporary tables, which need not be locked.

examines the log, he can discover this bug by noticing that join_tab→used_fields and field→query_id are mistakenly shared. After he fixes the bug, SVD will no longer break the atomic regions into small CU's in future executions.

3. Definitions of CU's and Serializability

This section formalizes the two key ideas behind SVD: inferring atomic regions and detecting serializability violations.

3.1. Dynamic Program Dependence Graph

We define computational units using *dynamic program dependence graph* (*d-PDG*). A *d-PDG* represents dependences on a *program trace* of a multithreaded program execution. The program trace is a sequence of all dynamic statements executed by all the threads, listed in execution order, a total order denoted $\rightarrow$. We say that $a \rightarrow b$ if dynamic statement a is executed before dynamic statement b or if $a = b$. A *thread trace* r of thread t is a subsequence of the program trace such that all dynamic statements in r were executed by t.

A d-PDG is a directed acyclic graph (V, E), whose vertices are dynamic statements and arcs are dependences partitioned into *true* (E_t), *control* (E_c), or *conflict* (E_h) dependences. True and control dependences exist only between vertices of the same thread, while conflict dependences exists only between vertices of different threads. A true dependence arc (a, b) exists whenever (I) $b \rightarrow a$; (II) a location v defined in b is used in a; and (III) v is not defined on the thread trace between b and a in a vertex other than b. Note that v may be defined by some other thread. A control dependence arc (a, b) exists whenever (I) modifying the predicate value of the conditional branch b would result in a not being executed; and (II) no other conditional branch on the thread trace between b and a could be used to bypass the execution of a. A conflict dependence arc (a, b) exists whenever (I) $b \rightarrow a$; (II) a location v is written by b and read or written by a or v is read by b and written by a; (III) v is not written on the program trace between b and a by a vertex other than a and b; (IV) a and b were executed by different threads. Our d-PDG definition is slightly different from the definition given by Miller and Choi [21]. In particular, we include conflict dependence arcs and omit synchronization dependence arcs; we use the former to detect serializability violations (Section 3.3), while we do not rely on the identification of the latter in the SVD algorithm.

To ensure that d-PDG is acyclic, we assume that vertices correspond to atomic operations and are fine-grain enough such that for all arcs (a, b) we have the property $b \rightarrow a$.

In contrast to d-PDG, a *thread* d-PDG (*td-PDG*) represents dependences in a thread trace. A td-PDG thus contains all true and control dependences of a given thread trace and omits all conflict dependences. In other words, td-PDG is a result of partitioning a d-PDG by thread membership of vertices. A td-PDG is identical to a *dynamic dependence graph* defined by Agrawal and Horgan [1].

In order to define a computational unit, we partition true dependences according to the nature of their locations. Dependences on variables not shared among threads are called *true-local* dependences, while remaining true dependences are called *true-shared* dependences. These two sets of arcs are denoted E_l and E_s, respectively. Note that true-shared dependences are intra-thread dependences, even though they involve shared variables.
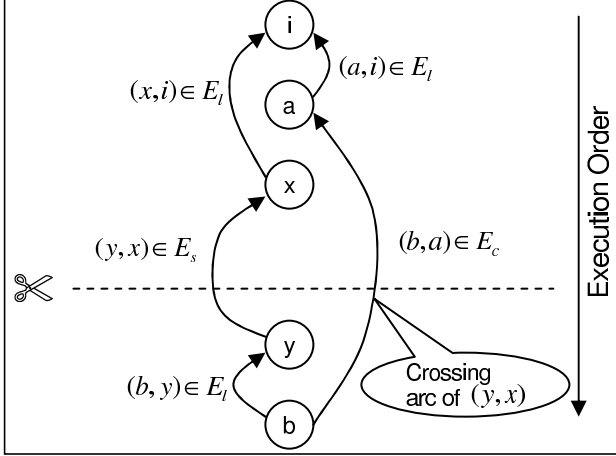
Figure 4: After the crossing arc (b, a) **is removed, the shared arc** (y, x) **is no longer in the same weakly connected component (along local and control dependences).**

3.2. Computational Units (CU)

A computational unit (CU) is an approximation of an atomic region. (Recall that a CU is neither an over- nor an under-approximation of an atomic region.) Specifically, a CU is a partition of a td-PDG determined by region hypothesis. Recall from Section 2 that region hypothesis places two constraints on CU's.

1) Within an atomic region, a write of a shared location v must not be followed by a read of v. In other words, a CU must not contain any $e \in E_s$.
2) An atomic region does not perform multiple "independent" computations. In other words, vertices of each CU must be *weakly connected*. (A directed graph is weakly connected if and only if its underlying undirected graph is connected.)

Unfortunately, the two constraints do not produce a unique partition of a td-PDG, even under the natural goal that each CU is maximal. This is because there is, in general, a choice as to where to cut a weakly connected graph to satisfy the first constraint. To make CU's unique, we define a heuristic that selects unique cuts. These cuts will be determined in execution order, so that vertices close in the program are grouped into the same CU.

Our plan is to define a set of *crossing arcs* whose removal will partition a weakly connected component such that a shared arc will be removed from it. After removing all crossing arcs and all shared arcs, the largest weakly connected components of td-PDG will yield a unique set of CU's.

Definition 1: Let $(y, x) \in E_c \cup E_l$. We say that (y, x) is a *crossing arc* of (b, a) iff there exists $(b, a) \in E_s$ such that (I) $y \rightarrow b$; and (II) x and a are weakly connected along arcs of $E_c \cup E_l$.

Figure 4 shows an example crossing arc. The control dependence arc (b, a) is a crossing arc, because there is a shared dependence (y, x) such that the vertices a and x are weakly connected along arcs of $E_l \cup E_c$, and y precedes b in execution order. After the crossing arc is removed, (y, x) is no longer in the same weakly connected component (along local and control arcs).

The following definition operationally defines a unique set of crossing arcs to remove, thus defining the partitioning of a thread trace into CU's.

Definition 2: A *reduced dependence graph* of a td-PDG t is obtained by removing from t arcs as follows.
1) Find an earliest arc[2] $e \in E_s$. An arc (b, a) is defined to be *earlier* than arc (y, x) if $b \rightarrow y$.
2) Remove all crossing arcs of e.
3) Remove arc e.
4) Repeat step 1, 2, 3 until E_s is empty.

Informally, as illustrated in Figure 4, we cut a td-PDG just before the execution reaches the source vertex of each shared arc.

Definition 3: Given a td-PDG t, the *computational unit* of a vertex x is the set of vertices that are weakly connected with x in the reduced dependence graph of t.

CU's defined by Definition 3 can overlap in a thread trace. In other words, vertices of a CU are not always adjacent in the thread trace. In the following, however, we assume *non-overlapping* CU's so that we can derive a heuristic to detect serializability violations by drawing results from database serializability theory as if CU's are database transactions.

3.3. Serializability and Strict 2PL

In this subsection, we define CU serializability and we derive a heuristic to detect serializability violations.

In shared-memory programs, CU's from different threads execute concurrently. However, correct and incorrect program executions differ in whether CU's are *serializable*.

Definition 4: CU's of a program trace are *serializable* iff there exists an *equivalent* program trace where all statements of each CU are adjacent to each other. We say two program traces are equivalent if they have identical d-PDGs.

If a thread trace contains only non-overlapping CU's, then at any given time the thread is executing at most one CU. All other CU's of the thread are either finished or not started. This model is identical to the serializability model of databases if CU's are replaced by database transactions [25]. In this case, CU serializability problem is equivalent to the transaction serializability problem in databases.

In databases, a popular method to guarantee transaction serializability is the *2-Phase Locking (2PL)* protocol [14]. *Strict 2PL* protocol is an important variation of 2PL protocol. In Strict 2PL, a transaction must gain exclusive access to the shared data between its initial access to the data and the end of the transaction. Exclusive access means (I) no other transaction can write a datum if the transaction is exclusively reading the datum, (II) no other transaction can read or write a datum if the transaction is exclusively writing the datum. Not violating strict 2PL is sufficient yet not necessary for serializability.

Here, we draw the results from strict 2PL to detect CU serializability violations. We check program traces for strict 2PL violations. In particular, we check whether any statement of a CU k has conflicted with a statement from a different thread before k finishes. A conflict before k finishes is equivalent to failing to have exclusive access to a shared datum. We report a serializability violation if strict 2PL is violated. The detection based on strict 2PL violation is *conservative*, because not violating strict 2PL is sufficient yet not necessary for serializability. We choose this heuristic, because detecting strict 2PL violations does not require exchanging exces-

---

2. An earliest arc is unique if each vertex read at most one shared variable (possible if dynamic statements are fine-grain enough).

```
Data Structures
   // S_T includes memory values (variables) that are being
      loaded and stored by a dynamic statement. The set s.dep-
      Pred includes statements that s is directly true-dependent
      or control-dependent upon. The reference cu points to the
      CU that contains this statement.
   S_T ::= structure (VAL leftValue,
           SET<VAL> rightValues,
           SET<S_T> depPred, CU_T cu)
   // CU_T includes a set of S_T, a set of values and a boolean
      flag. Variables in shVars are shared and have been writ-
      ten by this CU.
   CU_T ::= structure (SET<S_T> stmts,
           SET<VAL> shVars, bool active)
   // T_T is a set of S_T comprising a thread trace
   T_T ::= SET<S_T>

Algorithm
1   forall t in all threads of the program trace {
        S_T s
        while ((s := next_dyn_stmt_in_T_exec(t,s)) != NIL) {
            forall v in s.rightValues {
5               forall st in s.depPred {
                    if (st.cu.active==TRUE &&
                        v in st.cu.shVars)// shared dependence
                        st.cu.active := FALSE
            }   }
10          s.cu.stmts :=
                       ⋃                 st · cu · stmts
             st ∈ s · depPred ∧ st · cu · active ≡ TRUE
            forall st in s.cu.stmts { st.cu := s.cu } // update cu
            s.cu.stmts := s.cu.stmts + s
            s.cu.active := TRUE
15          if (s.leftValue.shared)
                s.cu.shVars := s.cu.shVars + s.leftValue
        } // end of while
        // "close" CU's, after finished scanning a thread trace
        forall cu in t {
20          if (cu.active) { cu.active := FALSE }
    }   }
```

Figure 5: The offline algorithm scans each thread trace and computes CU's.

sive information, such as timestamps, between threads. More accurate detection of serializability violations is possible with higher detection cost. We leave exploring this direction to future work.

4. Serializability Violation Detector

This section describes *Serializability Violation Detector (SVD)*, a software-based online detector. To simplify the presentation, we first give a simple offline, *multi-pass* algorithm in Section 4.1. Section 4.2 presents an online algorithm. SVD uses several heuristics to achieve *one-pass* detection. Section 4.3 presents some pragmatic considerations when we implement SVD. Finally, Section 4.4 briefly discusses a potential hardware version of SVD.

4.1. An Offline Algorithm

In this section, we describe an offline, multi-pass algorithm. The offline algorithm detects serializability violations of a program execution by scanning the program trace multiple times. To help understand the basic principles of our detector, we keep the offline algorithm as simple as possible. Later on in Section 4.2, we give an online, one-pass algorithm.

4.1.1 Program Traces and Dependence Predecessors

The offline algorithm operates on program traces where (I) true-dependent and control-dependent predecessors of a dynamic statement s are known and stored in a data structure *s.depPred*, and (II) a boolean flag *v.shared* indicates whether a variable v is shared. It is important to note that these two types of information are required only by the offline algorithm. The online algorithm in Section 4.2 does not require them. Instead, the online algorithm employs heuristics to compute them on-the-fly.

We assume two selection functions that let us scan either the whole program trace or each individual thread trace. Function *next_dyn_stmt_in_P_exec(r)* returns the next statement s that follows r regardless of whether s is executed by the same thread that executed r. Function *next_dyn_stmt_in_T_exec(r,t)* returns the next statement s that follows r, with the restriction that s and r must be executed by the same thread t.

4.1.2 Three Passes of the Offline Algorithm

The offline algorithm operates in three passes. The first pass scans each thread trace and computes CU's. The second pass assigns a unique sequence number to each dynamic statement in the program trace, which defines a total order. The second pass also records where a CU finishes its execution. Finally, the third pass scans the program trace and checks for strict 2PL violations. By doing so, serializability violations are detected and reported.

Figure 5 shows how the offline algorithm computes CU's from thread traces. The key for the algorithm is to find and remove crossing arcs from a td-PDG as a thread is executing. Using a boolean variable, *active*, the offline algorithm distinguishes those CU's that are still "connecting" to future statements and those CU's that are "cut" by shared dependence arcs. The offline algorithm is able to compute CU's in one pass, which is an important feature for the online algorithm later.

In Figure 5, the offline algorithm starts by iterating through all statements for each thread trace. For each variable v that a statement s reads, we check whether any statement in *s.depPred* wrote v and v is shared (line 4-7). If so, then the CU that contains the predecessor is marked inactive (line 8). Therefore, all crossing arcs that connect to the CU from a later dynamic statement are "cut". Next, all active CU's that contain any of s's dependence predecessors are merged (line 10), i.e. we build a larger CU that is weakly connected. Finally, s is added to the resulting CU (line 13) and the dependences in the CU are allowed to propagate further by leaving the CU active (line 14). Once CU's are computed, the information about which statements belong to which CU is stored to be used by future passes of the algorithm.

Figure 6 shows how the offline algorithm detects and reports potential serializability violations by first assigning a total order to all statements in a program trace. After that, it checks for strict 2PL violations. The second pass iterates through all dynamic statements of a program trace and assigns a unique sequence ID to each of them (line 4). Because we have already computed CU's in the first pass, we can record the sequence ID of the last statement of a CU (line 5), i.e. when a CU finishes its execution. In the third pass, the offline algorithm checks for conflicts due to a statement s_0 from a thread t_0 and a statement s_1 in a CU of a thread other than t_0 (line 13-15). If so, line 16 checks for strict 2PL violations. Finally, line 17 reports serializability violations.

```
Data Structures
  // S_T' records information of a statement
  S_T' ::= structure (int seqId, T_T' thread,
          CU_T' cu, VAL leftValue,
          SET<VAL> rightValues)
  // CU_T' records information about a CU
  CU_T' ::= structure (SET<S_T'> stmts,
          int maxSeqId)
  // T_T' is a set of CU's of a thread trace
  T_T' ::= SET<CU_T'>

Algorithm
1   int gSeqId := 0
    S_T' s
    while ((s := next_dyn_stmt_in_P_exec (s)) != NIL) {
        s.seqId := gSeqId++ // generate a total order
5       s.cu.maxSeqId := (s.cu.maxSeqId > s.seqId)?
                          s.cu.maxSeqId : s.seqId
    }
    while ((s := next_dyn_stmt_in_P_exec (s)) != NIL) {
        forall t in all threads in a program trace {
10          if (t != s.thread) {
                forall cu in t {
                    forall st in cu.stmts {
                        if ((s.leftValue == st.leftValue ||
                        s.leftValue in st.rightValues ||
15                      st.leftValue in st.rightValues) &&
                        (cu.maxSeqId > s.seqId > st.seqId)) {
                            report_violation (s, st)
    } } } } } }
```

Figure 6: The offline algorithm scans the total order of a program trace and records where a CU finishes its execution. It scans again the program trace and checks for strict 2PL violations to detect serializability violations.

While the offline algorithm is simple, it cannot be used online and requires program traces annotated with extra information. Next, we approximate the offline algorithm with an online algorithm, which includes several heuristics.

4.2. An Online Algorithm

Unlike the offline algorithm, SVD computes CU's and detects serializability violations in one-pass. Therefore, SVD can be used *online* during program execution although its performance overhead may be high. Furthermore, SVD infers which variables are shared as well as true and control dependences automatically. Recall that the offline algorithm requires this extra information.

One of the goals of SVD is to detect erroneous program executions regardless of whether program source code is available. Therefore, SVD uses only information that is available from program binaries. In particular, SVD uses *dynamic instructions* instead of dynamic program statements as the unit of computation and SVD uses fixed-sized *memory blocks* instead of variables as the unit of memory accesses.

As shown in Figure 7, the online detection algorithm of SVD observes a stream of "events". An event is either a dynamic *instruction* or a remote access *message* from other threads. Because threads execute in parallel, multiple instances of this algorithm are running simultaneously. Like the offline algorithm, this algorithm needs a data structure to represent a CU (CU_T"). In addition, because dynamic instructions operate on both memory blocks and registers, we define a memory block data structure

```
Data Structures
  // FSM_STATE as defined in Figure 8
  FSM_STATE ::= enum (Idle, Loaded, Loaded_Shared,
                      Stored, Stored_Shared, True_Dep)
  // CU type
  CU_T" ::= structure (SET<BLK_T> rs,
            SET<BLK_T> ws)
  // memory block type
  BLK_T ::= structure (CU_T" cu, FSM_STATE state)
  // register type
  REG_T ::= structure (SET<CU_T"> cuSet)

Subroutines
  check_violations() // check serializability violations
  merge_and_update() // merge units in a cuSet
  deactivate_log_CU() // stop growing a CU & generate log
  ctrl_dep_from_stack() // aggregate control dependences
  push_ctrl_cu(), pop_ctrl_cu() // control dependence stack
  is_instr() // check if an event is an instruction or a message

Algorithm
1   repeat until no more events {
        currrentTarget := 0x0 // control reconvergence point
        switch (event) {
        case (LOAD) args (block, destReg):
5           if (block.state == 'Stored_Shared')
                deactivate_log_CU(block.cu)
            block.cu.rs := block.cu.rs + block
            dest_reg.cuSet := { block.cu }
            break
10      case (ALU) args (srcR1, srcR2, destR):
            destR.cuSet := srcR1.cuSet ∪ srcR2.cuSet
            break
        case (STORE) args (srcReg1, srcReg2, block):
            // reg 1, reg2 contain data and addr, respectively
15          dataCuSet := srcReg1.cuSet
            addrCuSet := srcReg2.cuSet
            ctrlCuSet := ctrl_dep_from_stack()
            check_violations(dataCuSet ∪ addrCuSet
                              ∪ ctrlCuSet)
20          block.cu := merge_and_update(dataCuSet)
            block.cu.ws := block.cu.ws + block
            break
        case (BRANCH) args (srcReg, target):
            if (target.op == 'BA') // 'BA' is Branch-Always
25              currentTarget := target.target.pc // if...else...
            else currentTarget := target.pc // if...
            push_ctrl_cu(srcReg.cuSet, currentTarget)
            break
        case (REMOTE_ACCESS) args (block):
30          if (block.state == 'True_Dep')
                deactivate_log_CU(block.cu)
            break
        } // end of switch
        if (is_instr(event) && event.pc == currentTarget)
35          currentTarget := pop_ctrl_cu()
    } // end of repeat until
```

Figure 7: SVD's online detection algorithm.

(BLK_T) and a register data structure (REG_T). In the following, we describe some key heuristics that enable this online algorithm.

Infer true dependences via CU reference propagation. SVD automatically infers true dependences online by propagating unique CU references along program data flow graph as programs execute. First, SVD keeps a CU reference for each memory block
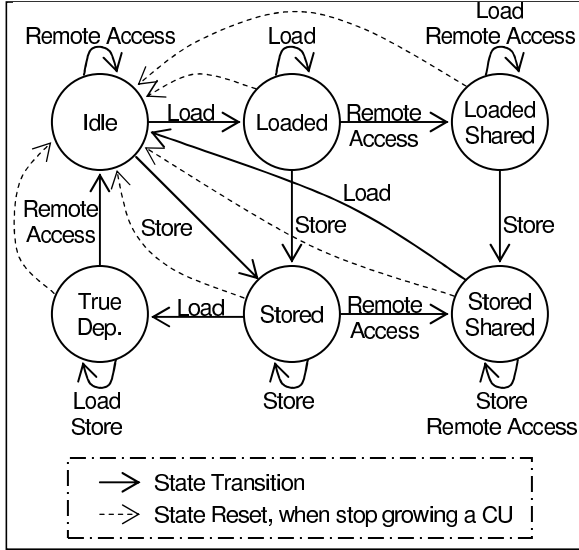
Figure 8: Memory block finite state machine (minimized).

in BLK_T.cu. SVD also maintains a set of CU references for each machine register in REG_T.cuSet. When a memory block is loaded into a register, the CU reference of the memory block is stored in the set of CU references of the register (line 8). This tags the register to be true-dependent on the memory block that is represented by the unique CU reference. When an arithmetic instruction is executed, SVD obtains the union of the two sets of CU references of the two source registers and stores the resulting set in the CU reference set of the destination register (line 11). Similarly, this tags the destination register to be true-dependent on the set of memory blocks that affect the source registers. Finally, when it comes to store instructions, SVD needs to store the dependences that are represented by the set of CU references of the source register into a single CU reference of the destination memory block. Therefore, SVD consolidates the dependences by merging all CU's pointed to by the set of CU references of the source register (line 20). Function *merge_and_update()* first merges all CU's, then updates old CU references stored in memory blocks and registers to a reference that is pointing to the new CU. Note that SVD never explicitly stores the true dependences between td-PDG vertices. Instead, SVD records enough information so that we know which CU will contain td-PDG vertices that are weakly connected via true dependences.

Infer partial control dependences via Skipper heuristic. SVD infers partial control dependences using a simple heuristic proposed in Skipper [7]. SVD infers only the control dependences of the *if-then-else* type of control flows. SVD does not infer control dependences of the loop type control flows. In particular, SVD keeps a stack of branch instructions and their control flow *recon-vergence points* [11]. When a branch instruction is executed, SVD first probes the branch target of the instruction and determines the control flow reconvergence point of the branch instruction (line 24-26). SVD then pushes the set of CU references that affect the branch instruction's outcome and the reconvergence point onto the stack. Later on, when the control reconvergence point is reached, SVD pops the top of the stack and updates the variable current-Target for next reconvergence point (line 34-35).

On line 17, SVD queries control dependences when store instructions are executed. Function *contrl_dep_from_stack()* aggregates all sets of CU references currently stored in the control dependence stack. The resulting set of CU references are used in checking serializability violations, which will be discussed shortly.

Infer shared memory blocks. In multithreaded programs, memory blocks are allocated, freed, and reallocated. Therefore, a memory block can change between being thread-local or shared in the life time of the program. SVD keeps a state (BLK_T.state) for each memory block to infer if a block is in thread-local or shared state. Note that although memory blocks are shared by all threads, SVD's data structures are privately maintained for each individual thread, i.e. different threads have separate BLK_T.state for the same memory block.

SVD maintains BLK_T.state using a finite state machine as shown in Figure 8. The state changes according to the sequence of load, store, and remote access events that happen to a block. Among the six states, two states (Loaded_Shared and Stored_Shared) represent that a shared block. When SVD detects that a CU is finished, SVD resets the block state of all blocks belonging to the CU to Idle — one of the states that represent a thread-local block.

Detect shared dependence. The purpose of maintaining BLK_T.state is that SVD can detect when a CU finishes its execution as shared dependences happen. SVD detects shared dependences on a memory block through two state transitions in Figure 8: (I) a load happens on a block in Stored_Shared state (also shown by line 5-6 in Figure 7) or (II) a remote access happens on a block in True_Dep state (line 30-31 in Figure 7). In both cases, the block state is changed to Idle as SVD detects the end of a CU. Function *deactivate_log_CU()* not only removes references of a CU from the SVD data structures (BLK_T, REG_T, and control dependence stack), but also changes the state of all blocks of a CU to Idle and generates proper entries in the CU log for the *a posteriori* examination.

Check for strict 2PL violations. SVD checks for strict 2PL violations whenever a store instruction is executed (line 18). From the three sets of CU references (Section 4.3 explains addrCuSet), SVD builds a list of memory blocks that the store instruction is control-, true-, or address-dependent upon. For each of these memory blocks, SVD checks if any conflict has happened *after* the CU has accessed the block. For brevity, we do not show how SVD keeps track of conflicts. However, the basic idea is straightforward: SVD keeps a BLK_T.conflict flag for each memory block, and sets it as load, store and remote access events are observed, and finally resets it when a CU ends.

4.3. Pragmatic Considerations

This section presents several pragmatic considerations to make SVD easier to implement and achieve a better trade-off between false negatives and false positives.

Represent CU with memory blocks, not dynamic instructions.
Instead of using a set of dynamic instructions to represent a CU (which is similar to the offline algorithm that uses a set of dynamic program statements), SVD uses two sets of memory blocks: a read set and a write set (CU_T".rs and CU_T".ws) to represent a CU. This is easier to implement, because SVD does not need to remember an arbitrary number of dynamic instructions. In our implementation, because each dynamic instruction accesses at most one

memory block, representing CU with memory blocks is strictly cheaper than representing CU with dynamic instructions. This approximation, however, introduces aliases when multiple dynamic instructions access the same memory block. Aliasing makes SVD infer CU's more conservatively, because more dynamic instructions may be included in a CU. We leave studying the impact of this approximation to future work.

CU's are weakly connected via only true dependences. By definition, vertices of a CU should be connected by either true or control dependence arcs. However, due to implementation constraints, we have only implemented connecting memory blocks of a CU through true dependence arcs. As shown on line 20, function *merge_and_update()* merges only CU references from data-CuSet. SVD stores the resulting new CU for further dependence propagation. On the other hand, SVD does check control dependences for serializability violations. We leave weakly connecting CU vertices via control dependence arcs to future work.

Handle vector, pointer data types (address dependences). SVD extends the offline algorithm to support vector and pointer data types by checking conflicts on address-dependent memory blocks when store instructions are executed. One of the source registers of a store instruction contains true dependences that affect the address computation of the store instruction (line 16). SVD computes the address dependences (addrCuSet) and reports serializability violations if any conflict happens to any of the memory blocks that affect the address computation of the store instruction. We do not propagate address dependences after variables are written to memory, because we find doing so causes more false positives.

Check only input blocks of a CU. Function *check_violations()* checks only the input blocks of a CU (CU_T".rs) for conflicts (line 18). This heuristic is not sufficient to detect all serializability violations. However, because SVD is conservative in detecting serializability violations, it is likely SVD reports false positives. In practice, we found employing this heuristic is more likely to find erroneous executions that are not serializable, hence, reduces SVD's false positives.

Approximate threads with processors. Finally, in our evaluation infrastructure (Section 6), threads may migrate from one processor to another. SVD does not have the ability to detect thread migration. Therefore, SVD approximates threads with processors, i.e. the algorithm shown in Figure 7 has a running copy for each processor in a simulated multiprocessor system.

4.4. Potential Hardware SVD

As more transistors become available on-chip, we believe that the overhead of the software version SVD can be dramatically reduced if some parts of it are implemented in hardware. First, hardware can help SVD infer true and control dependences if we piggyback CU references propagation to existing hardware data paths. Second, multiprocessor caches can help store CU's. Finally, cache coherence protocols can help detect serializability violations. We leave the detailed design and evaluation of hardware SVD to future work.

5. Qualitative Analysis

SVD relies on many heuristics to detect erroneous program executions. When these heuristics fail, SVD either fails to report erroneous executions (*false negatives*) or mis-reports correct executions
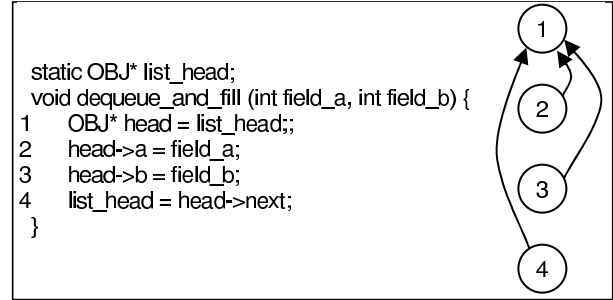

Figure 9: An example of an atomic region that contains independent computations. On the right, address dependences between statements are shown.

(*false positives*). This section analyzes the most important causes of false negatives and false positives.

5.1. False Negatives

Atomic regions contain shared dependences. As shown in Figure 3, a read to a shared variable (5.14) that follows a write to the same variable (5.03) may exist in an atomic region. In other words, atomic regions of a program may contain shared dependences. When such behavior exists in a program, SVD can cut weakly connected components of td-PDG to infer CU's that are smaller than the atomic regions. These small CU's can cause false negatives, because small CU's may be serializable while the atomic regions are not serializable. Although we mitigate this problem by allowing *a posteriori* examination to CU's inferred by SVD, these false negatives are still harmful, because SVD can not use BER to avoid them online. A better solution is to detect when a shared variable is not meant to be shared by the programmer. We leave exploring this direction to future work.

Atomic regions contain "independent" computations. Not all statements in an atomic region are weakly connected. As shown in Figure 9, an atomic region contains code to dequeue and fill a shared data queue. If both field_a and field_b are read from program inputs, i.e. not dependent, then the statements in this atomic region are not weakly connected. Again, small CU's inferred by SVD can cause false negatives. SVD mitigates the problem by checking address dependences (on variable head) before a variable is written to memory. In our experiments with shared-memory server programs, we have not observed false negatives (by comparing with another race detector) caused by atomic regions that contain "independent" computations.

5.2. False Positives

CU's that are too large. CU's that are larger than the atomic regions can lead to false positives. CU's can be too large, because SVD cuts CU's when shared dependences are observed, which is often after atomic regions have finished.

Strict 2PL violation. SVD can report spurious serializability violations when an execution violates strict 2PL but is still serializable.

Finally, our SVD *implementation* employs other heuristics, such as representing CU's with memory blocks, using fixed-size memory blocks, and checking only input blocks for a CU, etc. These heuristics also can cause false negatives and false positives. We leave detailed studies of their impacts to future work.

Table 1: Test Programs

Name	Description	The Erroneous Execution
Apache	Apache is a multithreaded open source web server. We use SURGE [5] to generate web requests that fetches a total of 500MB static web pages. We use apache 2.0.48 and enable a feature that buffers the access log in memory. Apache has estimated 285,000 lines of code.	Apache silently corrupts its access log with this setup. The bug was reported and a patch was available before we applied SVD.
MySQL	MySQL is a multithreaded open source database management system (DBMS). We use MySQL 4.1.1-alpha and an in-house query generator to continuously issue SELECT queries using the new prepared query interface of MySQL. MySQL has estimated 728,000 lines of code.	MySQL crashes non-deterministically with this setup. The crash was reported to MySQL developers. However, the root cause and a patch was *not* known before we applied SVD.
PgSQL	PostgreSQL (PgSQL) is a multiprocessed open source DBMS. PgSQL significantly differs from MySQL in database architecture and coding styles. We use PgSQL 8.0.0 beta3 and OSDL's DBT-2 benchmark to emulate a medium sized On-Line Transaction Processing (OLTP) workload that has total 1.4GB database and 20 warehouses. Each warehouse has one database connection and 15 terminals. PgSQL has estimated 659,000 lines of code.	There are *no* known errors with this setup. Our purpose is to study how well SVD performs on error-free execution such as these of PgSQL.

Next, we quantitatively evaluate SVD using shared-memory server programs (Section 6 and Section 7).

6. Evaluation Methods

The major premise of SVD is that it can be integrated with BER to avoid erroneous executions transparently. Our methodology measures the success of SVD indirectly by comparing SVD with a happens-before race detector that we have developed.

By using both detectors on identical executions of three large shared-memory programs, we measure *apparent false negatives* and *dynamic false positives* of SVD. Apparent false negatives are those erroneous program executions that are found by the happens-before detector but not by SVD. SVD is as effective as the happens-before detector if no apparent false negatives are found. Dynamic false positives are those dynamic instances of false positives reported by SVD. Reporting few dynamic false positives is important for a detector that is used in BER, because the number of dynamic false positives is proportional to the performance loss due to unnecessary rollbacks.

Our methodology favors the happens-before detector, because the required *a priori* annotation is available to the happens-before detector only.

We also evaluate the *static false positives* and how well a detector *helps programmers understand bugs*. These metrics are important when SVD is used with a post-mortem debugger. Static false positives do not include multiple warnings from the same static piece of code. Reporting few static false positives keeps programmers from being distracted. Helping programmers understand bugs means they can fix the bugs more quickly. We also report performance overhead of SVD.

Next, we describe our evaluation infrastructure and the happens-before race detector that is used to compare against SVD.

6.1. Simulation-based Deterministic Replay

We use Simics [19], a full-system simulator, at the center of our evaluation infrastructure. First, *full-system simulation* provides a deterministic and flexible execution environment for shared-memory programs. Starting from the same simulation checkpoint each time a program executes in Simics, the thread/process interleaving is solely determined by an initial random seed. By specifying the same seed, we can "replay" the same execution in Simics. Second, Simics provides a flexible implementation platform for SVD, which needs to capture various execution events, such as register reads/writes. Not only does Simics enable us to apply SVD to a wide range of programs, but also it allows us to study a hardware design of SVD in future work.

To obtain realistic program execution behavior, we use Wisconsin SMP Performance Model [2,20,37] to model the timing of a simulated SMP system, which contains four cache-coherent, 1 GHz, 4-way issue, out-of-order, SPARC processors running Solaris 9. SVD is entirely hidden from the simulated programs and OS and therefore does not perturb the simulated program executions.

The disadvantage of the simulator is its slowdown compared to the native execution. We overcome this problem by fast-forwarding and sampling the simulated executions. Fast-forwarding turns off the detailed timing simulation and helps us simulate only the part of the program execution that contains the actual bug manifestation. Sampling helps us study how long-running programs may impact SVD.

Table 1 summarizes the three server programs we use to test SVD. Programs are 285,000 lines of source code or greater. Our setups are derived from actual bug reports to these server programs. *The bug causing crashes in this MySQL setup was not known prior to running SVD.* Our setup adequately tests the scalability of SVD.

6.2. The Frontier Race Detector

To compare with SVD, we implemented a happens-before race detector called the *Frontier Race Detector* (FRD). A happens-before detector detects a race if two threads access a shared memory location and the accesses are *causally unordered* in a precise sense as defined by Lamport [18]. The original happens-before detectors compute the causal relationships between memory accesses based on known synchronization. FRD detects data races in two passes. In the first pass, without knowing synchronization, FRD first computes the tightest races, i.e. those conflicting accesses that are not causally ordered by any other conflicting accesses. These tightest races are called the frontier races [9]. Then FRD asks the programmer to annotate the frontier races as either *data* or *synchronization* races. After that, the frontier race detector scans the same program trace with the known synchronization accesses and finds data races just like a standard happens-before race detector.

We chose to use the frontier race detector instead of a standard happens-before detector, because we wanted to avoid annotating the massive amount of source code of the server programs. The frontier detector reports the same set of data races as a happens-before detector. However, it requires us to annotate only the synchronization operations that actually exist in the program trace. To avoid false sharing, we use *word-size* blocks in SVD and FRD.

' denotes a buggy exec.	Segment — Million Insts Across 4 CPUs	Samples	Apparent False Negatives	Static False Positives		Dynamic False Positives Per Million Insts (Total)		SVD's Computational Units	
				SVD	FRD	SVD	FRD	*a posteriori* Examinations	Dynamic CU's Per Million Insts (Total)
Apache'	16	1	**0**	**1**	2	**0.2 (3)**	1.3 (20)	2	324 (5183)
Apache	16	4	**N/A**	**2**	3	**0.1 (7)**	0.3 (16)	48	47 (2976)
MySQL'	40	1	**0**	**44**	91	**5.8 (233)**	140 (5620)	50	77 (3080)
MySQL	40	6	**N/A**	**60**	76	**8 (1924)**	29(6841)	97	77(18399)
PgSQL	16	16	**N/A**	**46**	4	**1.8 (456)**	0.03 (7)	87	8.6 (2194)

7. Evaluation Results

Table 2 summarizes our experimental results. We sample multiple execution segments for the three server programs. We report the results from both the erroneous execution samples and bug-free execution samples of Apache and MySQL.

7.1. Apparent False Negatives

In our experiments on Apache and MySQL, we found that SVD exhibits no apparent false negatives. Both SVD and FRD found two timing-dependent bugs in the two programs.

Both SVD and FRD found a known timing-dependent bug in Apache. SVD reported a serializability violation and FRD reported several data races related to this bug. We have examined the details of the bug in Section 2 (Figure 2). SVD helped the programmers to understand the bug by showing that the computation on the buffer index is "broken", i.e. the input to the computation is changed by other threads before the output of the computation is written.

Both SVD and FRD found a bug in MySQL whose root cause was previously unknown. SVD also *helped us understand* the bug. In Section 2, we have described the bug, which caused sever crashes (Figure 3). We have confirmed the root cause of the bug with the MySQL developers. SVD found the root cause of the bug by presenting the log of CU inputs and their last thread-local producers to the programmer. FRD also found this bug through data races on the mistakenly shared variables. However, it was not easy for us to find the race and identify it as a bug, because it was reported by FRD among many other static false positives.

7.2. Static and Dynamic False Positives

Table 2 also compares the reported false positives by both detectors. False positives include both static and dynamic false positives. The ratio of the static false positives and the actual bugs found indicates how many distractions the programmers would have to deal with when using a detector. The *frequency* of dynamic false positives (per million instructions per CPU) indicates how much performance would be lost in unnecessary rollbacks (BER).

For Apache and MySQL, SVD improves the race detection accuracy compared to FRD. With fewer static false positives, the programmer can find timing-dependent bugs more quickly. With fewer dynamic false positives (order of magnitude fewer for MySQL), SVD can reduce the unnecessary rollbacks.

For PgSQL, SVD reported more static and dynamic false positives than FRD. PgSQL has a different shared memory architecture than other multithreaded programs we have tested. We speculate that PgSQL developers may have spent more effort making it data race

free. Considering that SVD does not require *a priori* annotations and the low frequency of the dynamic false positives for PgSQL, SVD performs reasonably well for PgSQL.

7.3. Overheads

SVD has significant space and time overheads. The overheads mainly come from the following:

- Algorithm Complexity. SVD performs dependence calculations on every instruction of the execution. This incurs a significant time overhead.

- Recording CU's. SVD records CU pointer for each memory block, which means the space overheads is proportional to total memory footprint of a program.

- Debugging Support. SVD collects detailed debugging information, such as virtual PC and stack traces, which introduces both time and space overheads.

In our simulator, SVD incurs a significant slowdown, as high as a factor of 65. For some programs, such as Apache, SVD doubles the memory usage of the simulator. Despite the high overhead, we found SVD is scalable, because its performance overhead did not increase as the program size increases. The scalability of SVD is a result of focusing only on particular dynamic executions.

Finally, we sampled long executions (10 seconds in the steady state) to study if the long executions make SVD report more false positives. We found the number of static false positives grow slowly as the length of the execution increases, which means the main parameter to the number of static false positives is the exercised code size during the execution, not the length of the execution. On the other hand, dynamic false positives approximately increased linearly with the increase of the execution length.

8. Related Work

Another project that proposes online bug avoidance is ReEnact [29]. Using hardware available for thread level speculation, ReEnact strives to be both a deterministic debugger for data races and an on-the-fly race avoidance mechanism. However, ReEnact differs from SVD in that it requires an *a priori* program annotation to perform race detection and requires a predefined bug database to avoid of *known* bugs. SVD, on the other hand, does not require *a priori* annotations and can help avoid *unknown* bugs if it is integrated with a BER mechanism.

Existing bug-detectors for shared-memory programs, such as *data race detectors* and *atomicity violation detectors*, strive to detect more bugs, even when a program execution is correct. SVD com-

plements these techniques by being a dynamic detector that distinguishes *erroneous program executions* from correct executions.

Netzer and Miller [24] formalize races and point out it is NP-hard to detect data races without false negatives and false positives. Practical race detectors often sacrifice detection accuracy in allowing false positives, but not allowing false negatives.

One type of race detector uses the *lockset* algorithm. The lockset algorithm checks whether each shared variable in a program is consistently guarded by at least one lock. Eraser [33] uses the lockset algorithm during program execution to find data races. RacerX [13] uses the lockset algorithm in compile time to find data races.

Another type of race detector uses the *happens-before* algorithm. The happens-before algorithm checks whether conflicting accesses to shared variables in a program are ordered by explicit synchronization. Many false positives reported by the lockset algorithm can be avoided, because the happens-before algorithm can find synchronization that orders the unlocked accesses found by the lockset algorithm. Many dynamic race detectors implement the happens-before algorithm in software [10,17,21,32]. Hardware [22,29] and Distributed-Shared-Memory [26,31] implementations were also proposed to reduce the runtime overhead of the detectors.

It is also possible to combine these two algorithms [12]. Choi et al. have proposed hybrid detectors [8,36] that have both low overhead (lockset) and high accuracy (happens-before). SVD differs from both the lockset and the happens-before algorithms in that it does not require *a priori* annotations.

Recently, researchers have noticed that race detectors *cannot* find all timing-dependent bugs. For example, the stale-value detector [6] finds where stale values are used after critical sections have ended, because this type of program behavior may be an indicator of timing-dependent bugs.

More generally, atomicity based detectors find atomicity violations for predefined program code regions. The static atomicity detector uses a type system to allow programmers to specify atomic regions, hereby referred to as *atomicity annotations*. Therefore, potential bugs of atomicity violations can be found statically [16]. The dynamic atomicity detector tries to automatically infer atomicity annotations in Java programs and detects atomicity violations while monitoring the program executions [15].

SVD differs from atomicity detectors in that they use two different program safety properties — *serializability* versus *atomicity*. Atomicity requires that the program codes with atomicity annotations *always* execute in series. Atomicity detectors check how synchronization is done in programs. On the other hand, serializability is concerned with particular program executions. Atomic regions inferred by SVD may execute in series in certain executions, but not in others. SVD essentially ignores how synchronization is done in programs.

9. Conclusions and Future Work

We propose a serializability violation detector (SVD) that can detect erroneous executions of shared-memory programs without requiring *a priori* program annotations. Such a detector is potentially useful in alerting software users as software errors happen online or in triggering recovery to avoid erroneous executions with a backward error recovery (BER) mechanism. SVD reports few

dynamic false positives, which makes it particularly suitable to be used in avoiding erroneous executions caused by *unknown* bugs.

One of our most important contributions is to propose a new inference method of atomic regions that removes the *a priori* annotation requirement exists in typical detectors. We exploit region hypothesis to infer atomic regions dynamically as programs execute, without program source code. Experimental results shows that the inference method is effective on real server programs.

In the future, we plan to implement SVD in simulated hardware so that its space and time overheads are reduced. With low overhead, SVD can be integrated with a BER mechanism to avoid (on-the-fly) erroneous executions of server programs.

10. Acknowledgements

References

[1] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. In *Proceedings of the SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 246–256, June 1990.

[2] A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, M. Xu, D. J. Sorin, M. D. Hill, and D. A. Wood. Simulating a $2M Commercial Server on a $2K PC. *IEEE Computer*, 36(2):50–57, Feb. 2003.

[3] Apache HTTP Server Project. http://www.apache.org/.

[4] D. F. Bacon and S. C. Goldstein. Hardware-Assisted Replay of Multiprocessor Programs. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, pages 194–206, 1991.

[5] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of the 1998 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 151–160, June 1998.

[6] M. Burrows and K. R. M. Leino. Finding stale-value errors in concurrent programs. Technical report, Compaq Systems Research Center Technical Note (2002-04), May 2002.

[7] C.-Y. Cher and T. N. Vijaykumar. Skipper: a microarchitecture for exploiting control-flow independence. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 4–15, Dec. 2001.

[8] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 258–269, June 2002.

[9] J.-D. Choi and S.-L. Min. Race Frontier: Reproducing Data Races in Parallel-Program Debugging. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 145–154, July 1991.

[10] M. Christiaens and K. D. Bosschere. TRaDe, a topological approach to on-the-fly race detection in java programs. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM'01)*, 2001.

[11] J. D. Collins, D. M. Tullsen, and H. Wang. Control Flow Optimization Via Dynamic Reconvergence Prediction. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 129–140, Dec. 2004.

[12] A. Dinning and E. Schonberg. The Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 1–10, Mar. 1990.

[13] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proc. 19th Symposium on Operating System Principles*, pages 237–252, Oct. 2003.

[14] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[15] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 256–267, 2004.

[16] C. Flanagan and S. Qadeer. A Type and Effect System for Atomicity. In *Proceedings of the SIGPLAN 2003 Conference on Programming Language Design and Implementation*, June 2003.

[17] Y.-K. Jun and K. Koh. On-the-Fly Detection of Access Anomalies in Nested Parallel Loops. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging (PADD)*, pages 107–117, 1993.

[18] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[19] P. S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[20] C. J. Mauer, M. D. Hill, and D. A. Wood. Full System Timing-First Simulation. In *Proceedings of the 2002 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 108–116, June 2002.

[21] B. P. Miller and J.-D. Choi. A Mechanism for Efficient Debugging of Parallel Programs. In *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 135–144, June 1988.

[22] S. L. Min and J.-D. Choi. An Efficient Cache-based Access Anomaly Detection Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 235–244, Apr. 1991.

[23] MySQL AB. http://www.mysql.com/.

[24] R. H. B. Netzer and B. P. Miller. What are Race Conditions?: Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, Mar. 1992.

[25] C. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, Rockville, Maryland, 1986.

[26] D. Perkovic and P. Keleher. A Protocol-Centric Approach to On-The-Fly Race Detection. *IEEE Transactions on Parallel and Distributed Systems*, 11(10):1058–1072, Oct. 2000.

[27] PostgreSQL Global Development Group. http://www.postgresql.org/.

[28] K. Poulsen. SecurityFocus News: Tracking the blackout bug. http://www.securityfocus.com/news/8412.

[29] M. Prvulovic and J. Torrellas. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 110–121, June 2003.

[30] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 111–122, May 2002.

[31] B. Richards and J. R. Larus. Protocol-based Data-race Detection. In *SIGMETRICS symposium on Parallel and Distributed Tools*, pages 40–47, 1998.

[32] M. Ronsse and K. D. Bosschere. Non-intrusive On-the-fly Data Race Detection using Execution Replay. In *Automated and Algorithmic Debugging*, Nov. 2000.

[33] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.

[34] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 123–134, May 2002.

[35] U.S.-Canada Power System Outage Task Force. Final Report on the August 14th Blackout in the United States and Canada. Technical report, Department of Energy, 2004.

[36] C. von Praun and T. Gross. Object-Race Detection. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA)*, Oct. 2001.

[37] Wisconsin Multifacet GEMS Simulator. http://www.cs.wisc.edu/gems/.

[38] M. Xu, R. Bodík, and M. D. Hill. A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 122–133, June 2003.