2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

A Hardware Approach to Detect, Expose and Tolerate High Level Data Races

Lois Orosa
Instituto de Computação
Universidade de Campinas (UNICAMP)
Campinas, Brazil
lois.orosa@ic.unicamp.br

João Lourenço
NOVA LINCS - Departamento de Informática
FCT - Universidade Nova de Lisboa
Lisbon, Portugal
joao.lourenco@fct.unl.pt

***Abstract*—Concurrent programs are more complex and error prone than their sequential peers, and are much harder to debug as well. High level data races (HLDR) are one of the concurrency bugs most difficult to debug. They are a class of concurrency errors that are not commonly addressed by the testing and debugging techniques and tools. HLDR result from the misdefinition of the scope of an atomic block, which should be unique but was wrongly split into two or more independent atomic blocks. Interleavings involving these misdefined atomic blocks may violate the program correctness invariants and cause the concurrent program to fail. In this work we propose a hardware module to detect, expose and tolerate HLDR in concurrent programs, with applications in both the software testing and debugging and the software deployment phases. In the detecting mode, our proposal detects HLDR with few false positives and without the overhead and intrusion of other dynamic software approaches. In the exposing mode, it "stimulates" the program to expose existing latent HLDR and trigger hidden HLDRs. Finally, in the tolerating mode, it may act as a software healing technique by inhibiting certain buggy interleavings. The results shows a reasonable performance overhead and few false positives in all modes.**

***Keywords*-Concurrency, Testing, Debugging, Data races, Bloom filters**

I. Introduction

New multiprocessor architectures are forcing a shift in the software technologies towards exploiting parallel programming. This shift is challenging because now the programmer has to reason about many threads accessing data concurrently in non-deterministic interleavings.

Frequently, debugging the most tricky concurrency-related errors takes months (or even years) to be solved [1]. It is therefore of the utmost importance to invest resources in both simplifying the parallel programming models (making them less prone to errors) and in new hardware and software tools to facilitate the debugging task (thus reducing the software development costs). Many software and hardware tools have been proposed towards this last goal, covering a wide range of concurrency errors, including the detection of data races [2], atomicity violations [3][4], sequential consistency violations [5], asymmetric data races [6], and also tools to help coping with the non-determinism of concurrent programs, such as deterministic replay tools [7] and behavior analysis tools [8].

Among these concurrent errors, atomicity violations are specially difficult to debug. They are not easy to define nor to detect, as there is no way to know for certain which shared data accesses should be performed atomically by a program, as these accesses may be implicitly represented in the source code and scattered by multiple methods or even program modules. Therefore, some heuristics are used for *guessing* which accesses should be done atomically. These heuristics inevitably cause both false positives and false negatives. False positives report an atomicity violation when actually there is none, while false negatives fail to report an existing violation.

One of the most common ways to characterize atomicity violations in hardware approaches is by focusing on using trace analysis to identify unserializable interleavings with low probability of occurrence (that are directly correlated with atomicity violations). Several proposals used this scheme for detecting single-variable atomicity violations, which involves multiple accesses to a single shared variable, and multivariable atomicity violations, which involves accessing a set of variables. Whereas the first works on atomicity violations cover only single-variable atomicity violations [9][10], subsequent works tackle the more challenging problem of multivariable atomicity violations [11][4].

An alternative way to characterize atomicity violations, which will be the focus of this work, are High Level Data Races (HLDR), first defined by Artho et. al. [3]. Whereas it is difficult to compare this approach with the definition of atomicity violation as covered by the serializability, the definition of HLDR is wide enough to cover both single-variable and multivariable atomicity violations. A HLDR can occur when two concurrent threads access a set V of shared variables, which should always be accessed atomically, but at least one of those threads does not access the variables in V atomically. HLDR result from the misspecification of the scope of an atomic block, by splitting it in two or more atomic blocks that when interleaved with some other atomic blocks violates the expected atomicity in the accesses to the variables in V and may cause the concurrent program to malfunction or even to fail.

Software approaches to tackle HLDR may use dynamic [3] or static [12] program analysis techniques. In this

978-1-4673-8776-7/16 $31.00 © 2016 IEEE
DOI 10.1109/PDP.2016.57

work we propose, to the best of our knowledge, the first hardware approach for detecting, exposing and tolerating High Level Data Races in concurrent programs, by way of a unified approach covering both the development and the deployment phases of the software life cycle.

Our hardware module strongly resorts to Bloom filters [13]. A Bloom filter represents a set by way of a statistical component capable of storing and indefinite number of elements in a bounded space at the cost of reporting false positives but never false negatives, i.e., if the element is in the set the Bloom filter will always report it as present, while if the element is not in the set it will usually be reported as absent but sometimes it may wrongly be reported as present. We opted for using Bloom filters because they are a common structure used in a lot of different applications, and with a few changes in the control logic, the module could be adapted for other uses as well. Examples of other applications that use Bloom filters are transactional memory (TM) [14][15][16][17][18][19], data race detection and optimization [6][20].

We target a hardware implementation because software solutions are very slow, and in practice, they are only usable with small programs. The aim of a hardware solution is to make a slow overhead approach that could be used in real applications, and in real time, with negligible slowdown. The downsize is that the hardware resources are limited, and therefore the detection is not as precise as in software implementations. Furthermore, a processor incorporating our solution could reuse these Bloom filters for other purposes in case our functionality is not needed.

The remainder of this paper is organized as follows. We start by introducing background about HLDR in Section II, we describe our proposed hardware module in Section III, we evaluate our techniques in Section IV and we present some concluding remarks in Section V.

II. Background on High Level Data Races

For detecting HLDR, we base our work in the concept of *view* and *view consistency* defined by Artho [3]. A *view* is the set of variables accessed inside an atomic block. In the original proposal by Artho [3], a view included both the locations read and written. Dias et al. [12] suggested to differentiate between *read* and *write views*, which allows to avoid false positives because of read-read conflicts. In this work we use the original definition of a single view for reads and writes, which simplifies the algorithm and the hardware implementation, and because according to our experiments there are no much false positives due to read-read conflicts. However, an approach using separate views is perfectly feasible at the cost of some additional complexity.

A view is *maximal* in a thread if it is not a subset of any other view in the same thread. Intuitively, a maximal view represents a set of variables that should always be accessed atomically. Two threads are view consistent (and originate no HLDR) if, for each maximal view of one thread, the sets resulting from its intersection with the non-maximal views of the other thread form an inclusion chain between themselves.

```
atomic void getA() {
  return pair.a;
}
atomic void getB() {
  return pair.b;
}
atomic void setPair (int a, int b) {
  pair.a=a;
  pair.b=b;
}
boolean areEqual(){
  int a = getA();
  int b = getB();
  return a==b;
}
```

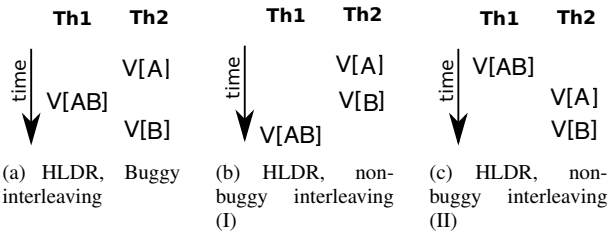
Figure 1. Example of a high level data race.

Figure 1 illustrates a static representation of a HLDR in a piece of source code, where the variables $pair.a$ and $pair.b$ are accessed atomically in the method *setPair* (they form a view $V_1 = \{pair.a, pair.b\}$), but are accessed in two separated atomic blocks in the *areEqual* method, namely in the atomic methods *getA* and *getB* (they form the views $V_2 = \{pair.a\}$ and $V_3 = \{pair.b\}$ respectively). In this example only V_1 is a maximal view, because both V_2 and V_3 are subsets of V_1. This example is not view consistent, because the intersection of V_2 and V_3 with the maximal view V_1 (in this example, $V_2 \cap V_1 = V_2$ and $V_3 \cap V_1 = V_3$) do not form an inclusion chain i.e., $V_2 \not\subseteq V_3 \wedge V_3 \not\subseteq V_2$.

We propose an approach that explores the temporal proximity and only manages atomicity violations that are close in time. This means our approach uses a best-effort policy and may cause false negatives, i.e., it may fail to flag some HLDR that although possible are not probable. However, our approach also enables simpler and better resource efficient implementations (only the last views are maintained). In its three operation modes, our approach is able to detect HLDRs, expose HLDRs in some cases where the manifestation of the bug is very rare, and tolerate HLDRs in buggy programs at execution time.

III. Dynamic handling of HLDR

In this section we explain how our approach differs from classical approach proposed by Artho [3]. Our proposal is based on a dynamic and best-effort algorithm to detect HLDR, that despite being designed for a hardware oriented implementation, it can be also implemented as a low space overhead software approach. Our approach builds on the concepts defined in the Section II, but adapts them to work with some imposed limits and restrictions, aiming at achieving a good compromise between precision and



(a) HLDR, Buggy interleaving (b) HLDR, non-buggy interleaving (I) (c) HLDR, non-buggy interleaving (II)

Figure 2. Possible view interleavings in the dynamic execution of a buggy program with a HLDR.

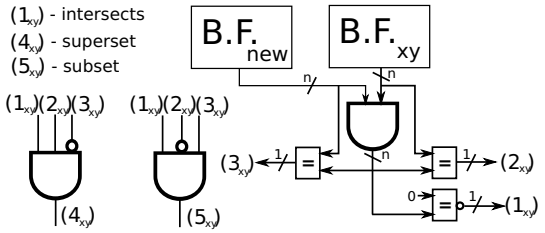


Figure 3. Logic associated with each view in the module (common to the views in the maximal window and in the regular windows).

complexity and at allowing an efficient hardware design and implementation.

At execution time, an actual HLDR may or may not lead to an incorrect execution, depending on the threads' interleaving and on whether the bug does or does not manifests itself. Figures 2(a), 2(b) and 2(c) depict three cases of the possible interleavings of a particular situation of HLDR, where there is a thread with a maximal view accessing variables A and B (V[AB]), and another thread accessing A and B in two separate atomic blocks (V[A] and V[B]). Although there is a potential HLDR, in the cases of Figures 2(b) and 2(c) the bug does not manifest itself as thread 1 does not break the atomicity between the variables A and B in thread 2. In Figure 2(a) the required atomicity between the accesses to A and B is broken.

We propose to consider a per thread bounded time-window of past events and only detect HLDR that involve views in this window. Hence, we define a *thread window* as the set of the last N views collected in a thread. We call $\{TW_0, TW_1, \ldots, TW_{M-1}\}$ to all the thread windows of the system, being M the maximum number of threads supported (in case of our hardware implementation, the maximum number of hardware threads). We define $\{V_x^0, V_x^1, V_x^2, \ldots, V_x^{N-1}\}$ as the set of views contained in the *thread window* x, being V_x^0 the most recent view, and V_x^{N-1} the oldest view. The *thread window* may easily be implemented in hardware as a set of Bloom filters (one per view), and the views in each *thread window* are replaced in FIFO order. Due to the bounded number of views maintained in each *thread window* our approach may introduce new false negatives. Nevertheless, this problem is of limited impact as experiments show that most of the HLDR are caused by views which are accessed close in time.

However, the use of Bloom filters to keep the views can cause false negatives in the detection of HLDR. For example, in Figure 2(a), if V[A] and V[B] are mapped to the same value of the Bloom filter, the race would be not detected. This situation should be rare if the Bloom filters are well dimensioned; we do not detect any in our evaluation.

Besides the *thread window*, we also introduce the concept of *maximal window* for optimizing the access to the maximal views. If a view is maximal in a *thread window* it is also replicated in the *maximal window* and extended with and additional field *ownTh* to identify the owner of that view. Therefore, the *maximal window* keeps all the maximal views from all the thread windows, and they are replaced in FIFO order. This maximal window may be larger than the regular thread windows for extending the visibility of HLDR and increase the precision of the system. In the maximal window, the views are represented as $\{MV_0, MV_1, ...MV_{L-1}\}$, being L the number of views that the maximal window can host.

The main objectives of the maximal window are:

- To simplify the logic and hardware implementation of the approach: when detecting HLDR is only required to check the intersections of regular views against the maximal views in the maximal window, and not with all the views in all the thread windows.
- To expand the scope of HLDR to the tolerating and exposing modes: the maximal views that took part in a detected HLDR are stored in the *maximal window* with a higher priority; and views with high priority stay in the maximal window for a longer time.

Because programs are naturally dynamic and our windows have a limited scope, a view may be maximal at a certain point of the execution and stop being maximal later on (or the other way around). Our approach distinguishes between two types of views in the *maximal window*:

- *Maximal views (MV)*: when a view is maximal in a thread window.
- *HLDR views (HV)*: are the maximal views that were previously involved in a HLDR.

These views have a pre-defined priority within the *maximal window*. The MVs have low priority, and HVs have high priority. New views entering the *maximal window* can only replace views with the same or lower priority. With this scheme, we prioritize the maximal views which have more probabilities of producing bugs.

The replacement algorithm in the maximal window is the following:

- If there are views with lower priority, the oldest low priority view is replaced.
- If there are no views of lower priority, the oldest view of high priority is replaced.

The main implementation decision was to associate each view of the regular window to a set of logical gates to

perform the intersection with the new view, as well as a comparator to check if the intersection relation with the regular view (intersect, subset or superset). Figure 3 shows how this relations are calculated in the hardware implementation. Signal 1_{xy} indicates if the new view and the regular view intersects (the signal is one) or not (zero), signal 4_{xy} indicates that the new view is a superset of the regular view and signal 5_{xy} indicates that the new view is a subset of the regular view.

Our hardware design is placed in the processor bus, and therefore it can snoop all the cache coherence messages. When a lock is detected, the module starts to collect the view (the memory accesses being accessed) until critical section ends (unlock). This design resembles to PACMAM [6].

We propose three different modes of operation, described herein, leveraging this basic structure of windows: the detecting mode, the exposing mode and the tolerating mode.

A. Detecting Mode

The detecting mode dynamically detects HLDR at runtime, in cases of both buggy (Figure 2(a)) and non buggy (Figures 2(b) and 2(c)) interleavings. The detection algorithm is an adaptation of the algorithm proposed by Artho [3] and is triggered each time a new view NV_x is added to a particular *thread window* TW_x of thread x: if the intersection of the new view NV_x with a maximal view MV_y $(y \neq x)$ is not empty and does not form an inclusion chain with all the other non-empty intersections of views in TW_x with the same MV_y, them a HLDR is detected and flagged. After the HLDR detection, the NV_x is inserted in the corresponding V_x^0 of the window for the x thread, replacing the oldest view (FIFO order). Furthermore, NV_x is maximal in TW_x, it is also inserted in the maximal window MV_x.

For reducing the amount of computations, we save the results of some computations in per view metadata in the thread windows:

- *pAv*: indicates if the view is a subset of any maximal view of other threads, and the relative ordering with respect the implied maximal view.
- *ptMax*: pointer to the position of the maximal view in the maximal window which is a superset of the view.

The possible values of *pAv* are *NONE*, *POTENTIAL* and *REAL*. The *NONE* value indicates that the view is not a subset of any maximal view of the other threads. If *pAv* is set to *POTENTIAL*, it indicates that the view arrived after the implied maximal view, and may cause a potential HLDR (as in the case illustrated in Figure 2(c), when processing V[A], its bit *pAv* is set to *POTENTIAL*). If *pAv* is set to *REAL*, it indicates that the view arrived before the implied maximal view, and may cause a real HLDR (as in the example of Figure 2(a), when processing V[AB], the bit *pAv* of the V[A] is set to *REAL*).

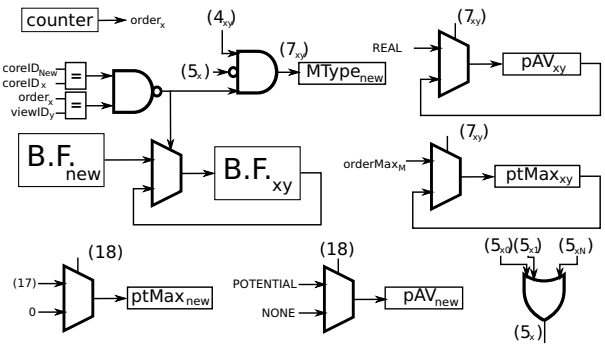


Figure 4. Logic in a regular window for inserting a new view.

Each new view updates its *pAv* and *ptMax* metadata, which will be used in the future to avoid redoing the subset calculations.

Figure 4 shows the basic logic for inserting the new view in its corresponding regular window, as well as the values of the $ptMax$ and pAV in the new view and in other views that interact with the new view. The signal 5_{xy} indicates that the view could be involved in a HLDR with a buggy interleaving. The signal 18 is generated in the checking process, and indicates that the view could be implied in a HLDR with a non-buggy interleaving.

Additionally, we also need two new fields in the *maximal views* of the *maximal window*:

- *thIDs*: indicates the owner thread(s) of the maximal view.
- *MType*: indicates the type of the maximal view (MV or HV), as described before.

If a new view is detected as a maximal view of a thread (if it is not a subset of any other view in the thread window), it is inserted in the maximal window (with $MType = MV$). A maximal view sets $MType = HV$ when a HLDR involving this maximal view is detected.

A HLDR is detected when one of the following two cases occur:

1) When a view from the same thread window than the new view has its pAv set to *REAL* or *POTENTIAL* and the intersection of this view with the maximal view pointed by its $ptMax$ does not form an inclusion chain with the intersection of the new view with the same maximal view. The case of pAv =*REAL* corresponds to a buggy interleaving, and the case of pAv =*POTENTIAL* corresponds to a non-buggy interleaving (as depicted in Figures 2(b) and 2(c) respectively).
2) When the new view is a maximal view, and there is a thread window (from a different thread) with two views that are a subset of the new maximal view and whose intersections with the maximal view do not form an inclusion chain between them (as in example of

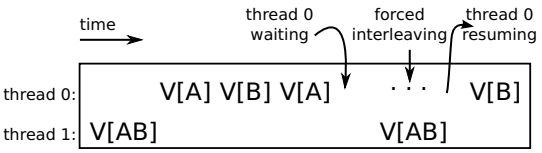


Figure 5. Example of the exposing mode.

Figure 2(a)).

When our approach detects a conflict, it launches an exception that will be handled by software. At this moment, the content of the Bloom filters are software accessible to facilitate the analysis of the HLDR. However, the actions taken by this software handler, including the analysis of the HLDR and possible notifications to the software developer, are outside the scope of this paper.

B. Exposing Mode

Frequently, concurrent bugs only manifest very rarely under particular interleavings. The *exposing mode* tries to force buggy interleavings in programs with HLDR.

To support this mode, we add a new $exposing$ field to each view of the maximal window, that indicates if that maximal view was previously involved in a HLDR.

The exposing mode requires the detection of HLDR according to the description in Section III-A. When a HLDR is detected the $exposing$ field of the maximal view involved in the HLDR is set to true, which gives this maximal view priority over the other views in the maximal window (this priority will be taken into consideration by the view replacement algorithm). For each subsequent view of any other thread, if it is a subset of an existing "exposable" maximal view, then the corresponding thread is stalled. Stalling the thread aims at enforcing a buggy interleaving. To avoid deadlocks the thread resumes after a well established period of time, or after a predetermined number of views are inserted into the system. This mode may cause some slowdown in the system because it introduces artificial delays by stalling threads, but performance shall not be a critical issue since this is a debugging mode.

Figure 5 shows an example where, after detecting a HLDR with a non buggy interleaving: thread 1 executes V[AB], and them thread 0 accesses A and B in two separate atomics blocks; when thread 0 executes V[A] the second time (note that V[A] is a subset of the "exposable" maximal view V[AB]) the thread stalls for a while to increase the probability of a buggy interleaving (that occurs in the example after the execution of V[AB] and V[B] at threads 1 and 0 respectively).

To avoid deadlocks due to stalls, we also need two extra fields per window. A $waiting$ bit indicating if the thread is stalled, and the $waiting\_num\_inserts$ field indicating the number of views inserted since the thread is stalled (when the limit is reached the stalled thread is resumed).

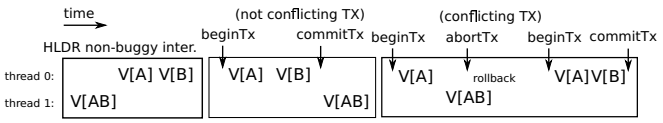


Figure 6. Example of the tolerating mode.

C. Tolerating Mode

Our key insight for this mode is to leverage on hardware transactional memory [21] (HTM) to build a HLDR tolerating mechanism. HTM is already supported in some new multicore processors [22], [23] and it is expectable that others will adopt this technology as well.

The tolerating mode also follows the approach described in Section III-A for detecting HLDR, but we propose to enclose the atomic regions that are suspicious of causing HLDR in a hardware memory transaction generated automatically and transparently by our hardware module. This mode however only works for non-critical HLDRs, and the system can only avoid repetitions of an "observed" HLDR.

To support the tolerating mode, four additional fields must be added to our system (two are presented now, the other two later in this section). The first addition is a new $tolerating$ field in each view of the maximal window. This field indicates if the maximal view has previously been involved in a HLDR (as execution modes are mutually exclusive, we may reuse the $exposing$ field of the exposing mode as the $tolerating$ field of this mode). The second addition comes from the necessity of mapping the views to the limits of the critical sections, that in our case are defined by the lock and unlock primitives. We identify each critical section by the lock variable. To ensure that a single transaction provides atomicity for a set of adjoining lock-based critical sections, the transaction must start before the beginning of the first critical section. Therefore, whenever the system detects an interleaving that may lead to a HLDR, the lock variable addresses of the two non-maximal views involved in the HLDR are stored in a per-thread list of locks. New occurrences of critical sections which lock variable is in this list, should trigger a new transaction.

Before acquiring any lock that is in its list of locks, a thread starts a transaction that protects subsequents atomic sections. A transaction will commit:

i) After the first unlock, if the view is not a subset of any maximal view with the $tolerating$ field set, in which case the transaction was useless;
ii) After detecting a HLDR with non-buggy-interleaving;
iii) Once the limit (maximum number) of views to be protected is reached; or
iv) Once the limits of the underlying HTM system are reached (e.g., the maximum size of the data set being protected).

Figure 6 shows an example of the tolerating mode. After

```
atomic void insert(num,x){
  if(ss[num] == NULL){
    ss[num] = x;
    num_elements++;
  }
}
atomic void delete(num,x){
  if(ss[num]==x){
     ss[num]=NULL;
     num_elements--;
  }
}
// global vars: num_elements; ss[];
// Initial values: ss=[r,NULL]
void thread1_exec(){
    delete(0,b); // executed at time 1
    insert(1,d); // executed at time 3
}
void thread2_exec(){
    insert(1,c); // executed at time 2
}
```

Figure 7. Example of low level false positive in *fmm*.

detecting a HLDR with non-buggy interleaving (AB-A-B), all the subsequent atomic regions that can possibly cause a HLDR are enclosed in transactions, and in case of conflict, the transaction is aborted and restarted.

Finally, to support this mode each thread window in the system requires a $inTx$ field that indicates if the thread is in a transaction or not, and a $Tx\#inserts$ field that indicates the number of views that were inserted since the transaction started (remember that the transaction also commits when the number of atomic blocks executed in the transaction reaches a predefined limit).

Furthermore, each view also needs a $spec\_view$ field that indicates if the view was generated inside a transaction, and therefore it is a speculative view. When a transaction commits, this field is set to zero in all the views of the window. In case of abort, the speculative views are discarded (i.e, the corresponding Bloom filters are cleared).

D. Hardware Issues: Low Level False Positives

Implementing this technique in hardware has some drawbacks; HLDR are defined as a high abstraction level, and all the explanations in Section II are based in variable names. However, our hardware proposal does not have such a high level visibility, and it is only able to construct the views based on the memory addresses being accessed by the processor in the load and write instructions. This can cause extra false positives in the system, which we call low level false positives.

Figure 7 shows a simplified example of a low level false positive detected in the *fmm* benchmark. There are two atomic functions, one to delete the content of a list, and another to insert content in the list. The views generated by the first thread (thread1_exec()), considering variable names, are $V_1 = \{ss, num\_elements\}$ (at time 1) and $V_2 = \{ss\}$ (at time 3), and the (maximal) view generated by the second thread (thread2_exec()) is $V_3 = \{ss, num\_elements\}$ (at time 2). These two threads are view consistent (no HLDR) according with their views: $V_1 \cap V_3 = IV_{13} = \{ss, num\_elements\}$, $V_2 \cap V_3 = IV_{23} = \{ss\}$, and IV_{13} and IV_{23} do not form an inclusion chain.

However, at a hardware level the views are formed by addresses, which imply that the views are $V_1 = \{ss[0], num\_elements\}$, $V_2 = \{ss[1]\}$ and $V_3 = \{ss[1], num\_elements\}$, and the two threads are not view consistent: $V_1 \cap V_3 = IV_{13} = \{num\_elements\}$, $V_2 \cap V_3 = IV_{23} = \{ss[1]\}$, and IV_{13} and IV_{23} form an inclusion chain ($IV_{13} \not\subseteq IV_{23} \wedge IV_{23} \not\subseteq IV_{13}$), which cause a low level false positive.

There are several ways of minimizing this problem, as for example use data coloring [4] to identify at hardware level which data corresponds to the same variable. However, as we show in Section IV, the number of low level false positives is contented, and does not justify the extra cost of new mechanism to avoid them.

Table I
BASELINE MODULE CONFIGURATION.

#threads	8
#Views per thread's window	5
Bloom Filters	256 bits
Type of views	Single view for RD/WR
#Views Maximal Window	15

IV. EVALUATION

In this section we evaluate our approach using a centralized hardware module to keep the thread windows and the maximal window. We configure the module with different parameters to compare their influence in the detected HLDR and we analyze the false positives reported. Furthermore, we experiment with the module in the exposing and tolerating modes, measuring the number of stalls introduced by the system in exposing mode, and obtaining statistics about the introduced HTM transactions in the tolerating mode.

A. Experimental Setup

We modeled our centralized hardware module system in a customized simulator build on top of the PIN instrumentation tool [24]. We simulated the mechanisms for detecting HLDR, and experimented with several configurations. We also used PIN to implement the hardware transactional memory support required for the tolerating mode, and the mechanism for stalling threads in the exposing mode.

In our experiments we used 8 threads, the HDLR module kept 5 views per thread implemented with 256 bit Bloom filters, the read and write sets were maintained in a single

```
synchronized(table) {
  table[N].value=V;
}
synchronized(table) {
  table[N].achieved=true;
}
synchronized(table) {
  if (table[N].achieved &&
    system_state[N]!=table[N].value){
    issueWarning();
  }
}
```

Figure 8. NASA HLDR.

Table II
ATOMIC BLOCK CHARACTERIZATION AND HLDR DETECTED IN DETECTING MODE.

		Characterization			HLDRs		
Id.	Benchmark	#AB	#RS	#WS	#RR	#LL	#RH
1	radix	341	8.2	3.1	1	0	0
2	fft	113	9.0	3.6	1	0	0
3	cholesky	22092	23.8	6.3	0	1	0
4	lu_cb	1013	9.2	3.7	1	0	0
5	lu_ncb	293	9.2	3.7	1	0	0
6	barnes	3549	104.3	40.0	1	1	0
7	fmm	1338	168.4	32.0	0	1	0
8	ocean_cp	15164	8.8	3.3	0	1	0
9	radiosity	108464	154.5	61.4	0	3	0
10	water_ns	41687	119.7	16.0	0	1	0
11	water_sp	453	7.8	2.9	0	1	0
12	fluidanimate	76643	6.8	1.1	1	0	0
13	streamcluster	178719	2.7	0.6	1	0	0
14	bodytrack	2998	6.1	3.2	1	0	0
15	nasa	1500	8.5	3.1	0	0	1

view, and we kept an unique maximal window composed by 15 views. Table I summarizes this baseline configuration.

We experimented with several well-tested lock-based benchmarks from the Parsec suite [25] and the SPLASH2 benchmarks [26], as well as with a suite of atomicity violations taken from previous works in HLDR [3][12]. The purpose of evaluating well-tested benchmarks is to measure the false HLDR reporting and the overhead introduced in different modes. The correctness of the module has been tested with simple kernel examples that simulate all the possible situations that generate HLDR.

As a realistic example of HLDR we took the problem that was detected in NASA's Remote Agent space craft controller [27], in which the error was very difficult to find, and it rarely manifests (only under certain thread interleavings). Figure 8 shows a simplified version of this HLDR.

Table II shows some characteristics of the benchmarks used to evaluate our system: the number of atomic blocks (#AB), the average number of read accesses per atomic block, or read set (#RS), and the the average number of write accesses per atomic block, or write set (#WS).

Table III
CAUSES FOR INTERRUPTING THE STALLS IN THE EXPOSING MODE.

Benchmark	timeout	#inserts	interleaving	Total
radix	51	12	0	63
fft	0	0	0	0
cholesky	8132	0	0	8132
lu_cb	0	0	0	0
lu_ncb	0	0	0	0
barnes	1370	680	0	2050
fmm	687	0	0	687
ocean_cp	1502	0	0	1502
radiosity	58649	0	0	58649
water_ns	0	0	0	0
water_sp	11	0	0	11
fluidanimate	58	3	113	174
streamcluster	0	0	0	0
bodytrack	10	0	0	10
nasa	6	5	1182	1193

B. Detecting Mode Evaluation

For evaluating the detecting mode, we measured the number of HLDR detected and we analyzed their causes. Table II shows the HLDRs detected classified in three categories; the #RR column shows the number of false positives detected due to read-read conflicts, the #LL column shows the number of low level false positives, and the #RH column shows the number of real HLDRs. The read-read conflicts could be avoided by duplicating the bloom filters and increasing the complexity by differentiating read and write views, and the low level false positives could also be avoided with extra complexity (discussed in Section III-D). However, we advocate for maintaining the module simple at the cost of this (few) false positives.

We also evaluated the impact of the window and Bloom filters sizes in the number of reported false positives. From our experiments we concluded that much larger windows increase the number of false positives, while sizes in the order of 256 exhibit a low false positive rate in the Bloom filters and also limit the number of false HLDR.

C. Exposing Mode Evaluation

In the exposing mode, the system introduces small delays by temporarily stalling the threads accessing views that were previously involved in a non-buggy HLDR interleaving (as described in Section III-A), with the aim of stimulating the buggy interleavings. There are three ways of resuming a stalled thread: after a default timeout (in the order of several hundreds of dynamic instructions), after a default number of inserts in the module (three in this experimental setup), or after a detection of a buggy interleaving.

Table III shows the number of total stalls in each benchmark. For each benchmark listed in this table, the stalls

Table IV
TRANSACTION CHARACTERISTICS IN THE TOLERATING MODE.

Benchmark	#Tx	#AB/abort	#AB/comm.
radix	20	0.95	0.00
fft	4	0.71	0.00
cholesky	804	1.43	2.98
lu_cb	44	1.00	0.00
lu_ncb	16	1.00	0.00
barnes	28	1.31	0.00
fmm	28	1.13	3.00
ocean_cp	48	1.08	0.00
radiosity	24119	0.71	3.00
water_ns	0	0.00	0.00
water_sp	49	1.23	2.78
fluidanimate	31522	1.04	3.00
streamcluster	0	0.00	0.00
bodytrack	0	0.00	0.00
nasa	72	1.14	3.00

reported were all triggered by a single HLDR, which occurred many times during the program lifetime. In the case of the NASA benchmark it was a real HLDR, in all the other benchmarks the stalls were caused by the false positives (see Table II). The column *timeout* indicates the number of times that the thread was resumed because the thread reached the stall time limit. The *#inserts* column indicates the number of times that the thread was resumed because it reached the maximum number of executed atomic blocks (in the whole system) since the thread was stalled. The *interleaving* column shows the number of times that the thread was resumed because the module forced a buggy interleaving.

Despite the number of stalls may appear very big in some cases, they are all produced by the same HLDR (see Table II). In the case of the NASA benchmark, most of the HLDR bugs are exposed correctly. We could avoid the unnecessary stalls by introducing a list of know false positives that should be ignored in future executions.

D. Tolerating Mode Evaluation

For evaluating the tolerating mode, we analyze the transactions started by the module for protecting atomic blocks.

Transactions commit when they reach a predefined number of inserts without conflicts. In the configuration of this evaluation, the module protects three atomic blocks before commit. If there is a data conflict with any other data access in the system (transactional or not transactional data), the transaction aborts and restarts. To avoid deadlocks and livelocks, we establish a limit of three restarts per transaction. When this limit is reached, the transactional boundaries are eliminated and the code is executed without protection.

Table IV shows the main characteristics of the transactions. *#Tx* are the number of transactions in the benchmark, *#AB/abort* are the number of protected atomic blocks per aborted transaction, and *#AB/commit* is the average number of protected atomic blocks per committed transaction (it could never exceed 3 in our testing configuration). A value of 3 in the *#AB/commit* column indicates that all the commits happened because they reached the maximum number of views, and not because they found an HLDR. A value of 0 indicates that no transactions committed (because there were no transactions or because all transactions reach the maximum number of restarts).

The performance overhead of this mode is given by the number of transactions, the size of the (read and write sets of the) transactions, and the number of aborts. The number of transactions is low in general (excepting *radiosity* and *fluidanimate*), the size of the read and write sets is also moderate (with the exception of *cholesky* and *water_sp*), and the number of aborts is also small in general. Taking into consideration that hardware support for transactional memory is already mainstream for some processors of the main manufacturers, the execution overhead caused by the transactions introduced by the module is expected to be reasonable.

V. CONCLUSIONS

In this paper we propose a dynamic best-effort method for detecting, exposing and tolerating High Level Data Races with low overhead and low hardware support. The proposed method is oriented towards an hardware implementation and thus use resources scarcely. The detecting and exposing modes help the programmer to identify and check HLDR, while the tolerating mode is useful in production runs to (temporarily) heal buggy software distributions and tolerate some HLDR in faulty codes. The results show that the implemented module can detect the HLDR with few false positives and can expose them with a reasonable performance slowdown. Furthermore, the proposed module can also tolerate HLDR with the support of a (hardware) transactional memory system with also a reasonable performance overhead.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their comments. This work was supported by FAPESP (grants 2014/03840-2 and 2013/08293-7) and STSM action cost ECOST-STSM-IC1001-300913-034427.

REFERENCES

[1] J. Wu, H. Cui, and J. Yang, "Bypassing races in live applications with execution filters," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–13.

[2] J. Devietti, B. P. Wood, K. Strauss, L. Ceze, D. Grossman, and S. Qadeer, "Radish: Always-on sound and complete ra detection in software and hardware," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 201–212.

[3] C. Artho, K. Havelund, and A. Biere, "High-level data races," in *JOURNAL ON SOFTWARE TESTING, VERIFICATION & RELIABILITY (STVR*, 2003, p. 2003.

[4] B. Lucia, L. Ceze, and K. Strauss, "Colorsafe: Architectural support for debugging and dynamically avoiding multi-variable atomicity violations," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 222–233.

[5] X. Qian, J. Torrellas, B. Sahelices, and D. Qian, "Volition: Scalable and precise sequential consistency violation detection," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 535–548.

[6] S. Qi, N. Otsuki, L. Orosa, A. Muzahid, and J. Torrellas, "Pacman: Tolerating asymmetric data races with unintrusive hardware," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, feb. 2012, pp. 1 –12.

[7] Y. Chen, W. Hu, T. Chen, and R. Wu, "Lreplay: A pending period based deterministic replay scheme," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 187–197.

[8] J. a. Lourenço, R. Dias, J. a. Luís, M. Rebelo, and V. Pessanha, "Understanding the behavior of transactional memory applications," in *Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, ser. PADTAD '09. New York, NY, USA: ACM, 2009, pp. 3:1–3:9.

[9] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "Avio: detecting atomicity violations via access interleaving invariants," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-XII. New York, NY, USA: ACM, 2006, pp. 37–48.

[10] B. Lucia, J. Devietti, K. Strauss, and L. Ceze, "Atom-aid: Detecting and surviving atomicity violations," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 277–288.

[11] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou, "Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 103–116.

[12] R. J. Dias, V. Pessanha, and J. a. M. Lourenço, "Precise detection of atomicity violations," in *Proceedings of the 8th International Conference on Hardware and Software: Verification and Testing*, ser. HVC'12. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 8–23.

[13] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[14] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "Logtm-se: Decoupling hardware transactional memory from caches," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 261–272.

[15] L. Peng, L. guo Xie, X. qiang Zhang, and X. yan Xie, "Conflict detection via adaptive signature for software transactional memory," in *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, vol. 2, april 2010, pp. V2–306 –V2–310.

[16] C. Ferri, A. Marongiu, B. Lipton, R. I. Bahar, T. Moreshet, L. Benini, and M. Herlihy, "Soc-tm: Integrated hw/sw support for transactional memory programming on embedded mpsocs," in *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '11. New York, NY, USA: ACM, 2011, pp. 39–48.

[17] W. Choi and J. Draper, "Improving utilization of hardware signatures in transactional memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 11, pp. 2230–2239, 2013.

[18] J. Casper, T. Oguntebi, S. Hong, N. G. Bronson, C. Kozyrakis, and K. Olukotun, "Hardware acceleration of transactional memory on commodity systems," *SIGPLAN Not.*, vol. 46, no. 3, pp. 27–38, Mar. 2011.

[19] L. Orosa, E. Antelo, and J. D. Bruguera, "Flexsig: Implementing flexible hardware signatures," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 30:1–30:20, Jan. 2012.

[20] J. Tuck, W. Ahn, L. Ceze, and J. Torrellas, "Softsig: Software-exposed hardware signatures for code analysis and optimization," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 145–156.

[21] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ser. ISCA '93. New York, NY, USA: ACM, 1993, pp. 289–300. [Online]. Available: http://doi.acm.org/10.1145/165123.165164

[22] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael, "Evaluation of blue gene/q hardware support for transactional memories," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 127–136.

[23] T. Jain and T. Agrawal, "The haswell microarchitecture–4th generation processor," *International Journal of Computer Science and Information Technologies*, vol. 4, no. 3, pp. 477–480, 2013.

[24] 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 *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200.

[25] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 72–81.

[26] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ser. ISCA '95. New York, NY, USA: ACM, 1995, pp. 24–36.

[27] B. Pell, E. Gat, R. Keesing, N. Muscettola, and B. Smith, "Plan execution for autonomous spacecraft," in *AAAI Fall Symposium Series: Plan Execution: Problems and Issues*, 1996, pp. 109–116.