A Dynamic Low-Intrusion Approach to Detect, Expose and Tolerate High Level Data Races

***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 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 dynamic low-intrusion approach 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 in binary/executable programs with few false positives and without the overhead and intrusion of other dynamic approaches. In the exposing mode, it stimulates the program to expose existing latent HLDR. Finally, in the tolerating mode, it may act as a (temporary) software healing technique by patching the code and inhibiting certain buggy interleavings.

The proposed approach use resources scarcely and could be implemented as a hardware-based tool oriented towards the class of HLDR errors.

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 and unpredictable interleavings.

Frequently, debugging the most tricky concurrency-related errors take months (or even years) to be solved [16]. 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 [5], atomicity violations [9], [1], sequential consistency violations [13], asymmetric data races [12], and also tools to help coping with the non-determinism of concurrent programs, such as deterministic replay tools [4] and behavior analysis tools [8].

In this work we address the question of the correctness of concurrent programs and propose a dynamic low-intrusion mechanism for detecting, exposing and tolerating High Level Data Races (HLDR) [1]. HLDR result from the mispecification of the scope of an atomic block, by splitting it in two or more atomic blocks which when interleaved with some other atomic blocks may cause the concurrent program to malfunction or even to fail. There are software approaches that address the detection of HLDR using either dynamic [1] or static [6] program analysis techniques but, to the best of our knowledge, our approach is the first addressing HLDR in a unified approach covering both the development and the deployment phases of the software life cycle. Furthermore, we optimize our approach for being hardware implemented, which would provide low-intrusiveness in the execution flow.

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

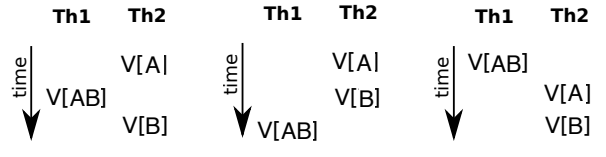
Fig. 1: Example of a high level data race.

We base our work in the concept of *view consistency* defined by Artho [1]. A *view* is the set of read and write addresses accessed in an atomic block. A view is maximal if it is not a subset of any other view in the thread. Intuitively, a maximal view represents sets of variables that should be always accessed atomically. A program is free from HLDR if all views of one thread that are a subset of a maximal views from another thread form an inclusion chain between themselves.

Figure 1 illustrates a static representation of a HLDR in a piece of source code, where the variables a and b are accessed atomically in the method *setPair*, but are accessed in two separated atomic blocks in the *areEqual* method (in the atomic methods *getA* and *getB*).

Our approach is best-effort, dynamic, and it works by temporal proximity, with the aim of introducing few performance overhead and using low resources in a hardware implementation. In its three operation modes, our approach is able to detect HLDR, expose HLDR in cases where the manifestation of the bug is very rare, and tolerate HLDR in buggy programs at execution time.

The remainder of this paper is organized as follows. We start by the description of our key ideas in Section II, we



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

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

evaluate our techniques in Section III and we present some concluding remarks in Section IV.

II. Dynamic handling of HLDR

Unlike previous works on HLDR, our approach is best-effort and it has low runtime overhead. To achieve a good compromise between precision and cost (hardware resources used) we propose to consider a bounded time-window (*thread window*) of past events and only detect HLDR in this window. Hence, we define a *thread window* as the set of the last N views collected in a thread.

At execution time, an actual HLDR may or may not lead to an incorrect execution, depending whether the bug manifests itself or not. Figures 2(a), 2(b) and 2(c) show examples of the possible interleavings of a particular case of HLDR, where there is a thread with a maximal view accessing variables A and B (V[AB]), and another thread access A and B in two separate atomic blocks. 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 the thread 2. In Figure 2(a) the atomicity between the accesses to A and B is broken, hence violating the view consistency and triggering a HLDR.

Besides the *thread window*, we also introduce the concept of *maximal window* for optimizing the access to the maximal views. The *maximal window* keeps the last *maximal views* from all threads in the same structure. Each view in this window is extended with an additional field *ownTh* to indicate the thread owning that view. This window may be larger than the regular windows for extending the visibility of HLDR and increase the precision of the system.

The main objectives of the maximal window are:

- To simplify the logic of the approach when detecting bugs: it only requires to check the intersections with views in the maximal window and not with views in all the other *thread windows*.
- To expand the scope of HLDR in tolerating and exposing modes: the maximal views associated with the potential bugs are stored in the *maximal window* with a higher priority, which allows to remain in that window a longer period of time.

Furthermore, this design simplifies the hardware implementation of the approach.

Because of the dynamic nature of programs and our limited window scope, a view may be maximal at a certain point of the execution, and stop being maximal later on (or the other way around). For this reason, our approach distinguish between several types of views in the maximal window:

- *Regular views (RV)*: are views that are in the *maximal window*, but they are not maximal. I.e, the first views of an execution will always be inserted in the maximal window as regular views (to fill the empty window).
- *Maximal views (MV)*: when a view is maximal in a thread window.
- *HLDR views (HV)*: these are maximal views that previously caused a HLDR.

These three types of views have a pre-defined priority within the maximal window. The RVs have the lowest priority, MVs have a medium priority, and HVs have the highest priority. New views entering the maximal window can only replace views with the same or lower priority. With this scheme, we prioritize the views which have more probabilities of cause a HLDR.

From now on, we will suppose that the atomic blocks are created using locks, but it could be easily adapted for transactional-based programs. For the explanation of the idea, we also assume that unique views include both read and write accesses. Furthermore, to improve the performance and reduce resource usage we use Bloom filters [3] for encoding the views data sets.

We propose three different modes of operation leveraging this basic structure of windows.

A. Detecting Mode

The detecting mode dynamically detects HLDR at runtime. For each new view, if its intersection with a maximal view of another thread does not form a chain with the intersection of any other view of the thread window, and both are subsets of this maximal view, them a HLDR is detected. After the HLDR detection, the new view is inserted in the corresponding window for that thread, replacing the oldest view (FIFO order). Furthermore, if this new view is maximal in the thread window, it is also inserted in the maximal window.

For reducing the amount of computations, we save the results of some computations in per view metadata. We introduce two fields for each view 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 $NONE$, $POTENTIAL$ or $REAL$. The $NONE$ value indicates that the view is not a subset of any maximal view of 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 (it is the case of the example of Figure 2(c), when processing V[A], its bit *pAv* would be 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 (in the example of Figure 2(a), when processing V[AB], the bit *pAv* of the V[A] is set to $REAL$).

Each new view updates its *pAv* and *ptMax* which will be used in the future for not redoing the subset calculations.

Complementary, 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 (RV, 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 $MType = MV$). Just at the beginning of the execution, when the maximal window is not full, not-maximal views can be introduced in the maximal window (with $MType = RV$). In case the views with $MType = RV$ finally result in maximal views, the value is changed for MV. In other case, the view is eventually replaced. 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 $ptMax$ do 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 examples of Figure 2(a) and Figure 2(c) respectively).
2) When 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 2(b)).

When our approach detects a conflict, it launches an exception that will be handled by software. The content of the Bloom filters are software handler 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 this maximal view was involved in a HLDR before.

The exposing mode requires the detection of HLDR according to the description in Section II-A. When a HLDR is detected the $exposing$ field of the maximal view implied in the HLDR is set to true, which gives this maximal view priority over the other views in the maximal window (which will be considered by the view replacement algorithm). For each subsequent views, if it is a subset of an existing "exposable" maximal view, then the corresponding thread is stalled. The aim of stalling the thread is to force the interleaving of a maximal view from another thread. 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 of the stalled threads, but since this is a debugging mode, the performance is not a critical issue.

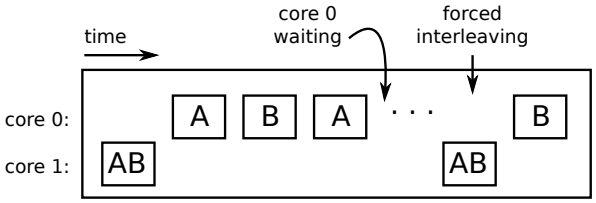


Fig. 3: Example of the exposing mode.

Figure 3 shows an example where, after detecting a HLDR with a non buggy interleaving (thread 1 executes V[AB], and them thread 2 executes access A and B in two separated atomics blocks), after the second access to V[A] (which is a subset of the "exposable" maximal view V[AB]) the thread stalls hopping a buggy interleaving (that is finally produced).

To avoid deadlocks due to stalls, we also need two extra fields per window. A $waiting$ bit indicates if the thread is stalled, and the $waiting\_num\_inserts$ field indicates the number of views inserted since the thread is stalled (when the limit is reached the thread is resumed). Furthermore, each thread will also resume their execution after a configurable waiting time.

C. Tolerating Mode

Tolerating mode also needs to detect HLDR as described in Section II-A. Furthermore, we use transactions to enclose the atomic regions that are suspicious of causing HLDR. These transactions are not the classical transactions defined by the programmer, but are rather generated automatically and transparently, without breaking the original semantics of the program. Our key insight in using transactional memory as tolerating mechanism is that it already has hardware support in some new multicore processors [14], [7] and it is expectable that others will include this support as well.

To support this tolerating mode, we need to include some more additional fields to our system. The first addition is a new $tolerating$ field in the views of the maximal window, which indicates if the maximal view was previously involved in a HLDR (we may reuse the $exposing$ field of the exposing mode). The second addition comes from the necessity of mapping the views to the corresponding lock IDs. To ensure that a single transaction provides atomicity

for a set of adjoining lock-based atomic sections, the transaction must start before the beginning of the first atomic block. Therefore, when a potential HLDR is produced, the lock addresses of the two views involved in the HLDR (excluding the maximal view) are maintained into a list of locks associated to the thread. Since that moment, before acquire any lock that is in this list of locks, the 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 (despite the lock is in the list of locks that launches transactions). In this case the transaction is useless.
ii) After a view causing a HLDR that it is not manifested.
iii) After reaching the limit of the number of views protected.
iv) After reaching the limit of the size of the data set protected (determined by the maximum capacity of the TM system).

Figure 4 shows an example of the tolerating mode. After detecting a HLDR with non-buggy interleaving (AB-A-B), all the subsequent atomic regions that can possible cause an 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 an $inTx$ 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 (the transaction commits when the number of atomic blocks executed in the transaction reach a predefined limit).

Furthermore, in each view we need to add 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 (the Bloom filters are cleared).

III. Evaluation

In this section we evaluate our approach with 8 threads and a centralized 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 mode to measure the number of stalls introduced by the system, and we also obtain statistics about transactions in the tolerating mode.

A. Experimental Setup

We build a centralized module system using the PIN instrumentation tool [10]. We simulate the mechanisms for detecting HLDR, and experiment with several configurations. We also use the same platform to implement the strong transactional memory support required for the tolerating mode, and the mechanism for stalling threads in the exposing mode.

TABLE I: Baseline module configuration.

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

TABLE II: Atomic block characterization and HLDR detected in detecting mode.

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

Table I shows the baseline configuration. We make the experiments with 8 threads, the HDLR module maintains 5 views per thread, implemented with 256 bit Bloom filters, the read and write sets are maintained in only one view, and we keep a unique maximal window composed by 15 views.

We experimented with several well-tested lock-based benchmarks from the Parsec suite [2] and the splash2 benchmarks [15], as well as with a suite of atomicity violations taken from previous works in HLDR [6] [1]. The purpose of evaluating well-tested benchmarks is to measure the false HLDR reported, 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 NASAs Remote Agent space craft controller [11], in which the error was very difficult to find, and it rarely manifests (only under certain thread interleavings).

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 (#RD/AB), and the the average number of write accesses per atomic block (#WR/AB). The last column (#HRs) indicates the number of HLDR detected in the detecting mode (see next section).

B. Detecting Mode Evaluation

For evaluating the detecting mode, we measure the number of HLDR detected and we analyze what causes them. As we can see in Table II (in column #HRs), the

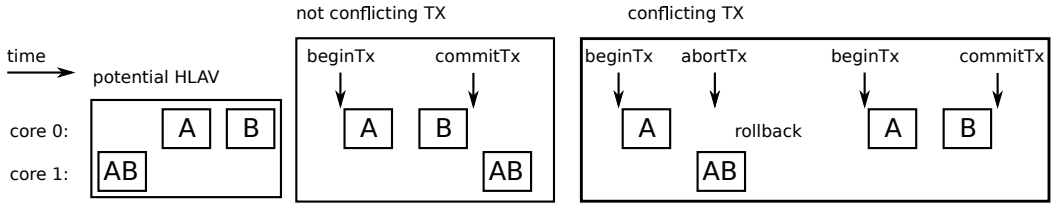


Fig. 4: Example of the tolerating mode.

TABLE III: Different types of HLDR detected in the benchmarks.

Type	Description	Bench. Ids
(1)	F.P. Ext. libraries	7
(2)	F.P. User level	1,2,3,4,5,8,9,10,11,12,13,14
(3)	Real HLDR	15

module detects one HLDR in the most of the benchmarks, in the barnes benchmark the module detected two HLDR, and in radiosity it detected three. All these benchmarks are free of bugs, excepting the nasa benchmark. Therefore, the HLDR detected by the module in parsec and splash2 benchmarks are all false positives, whereas in the Nasa benchmark the module detects the real HLDR.

We make some experiments to study the impact of the window and Bloom filters sizes. We find out that very big windows lead to more false HLDR, and that a size of 256 bits are enough to maintain a low false positive rate in the Bloom filters and limit the number of false HLDR.

Table III shows the three different types of HLDR detected by the module. Type (1) are HLDR caused by external libraries, which can not be verified because the source code is not available; type (2) are false HLDR resulting from executing application code; and type (3) are real HLDR.

C. Exposing Mode Evaluation

In the exposing mode, the system temporarily stalls the threads and introduce small delays in the threads accessing views that were previously involved in a non-buggy HLDR interleaving (as described in Section II-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 IV shows the number of total stalls in each benchmark. For each benchmark listed in this table, the stalls 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 it was a false positive. The column *timeout* indicates the number of times that the thread was resumed because the thread reach the stall time limit. The *#inserts* column indicates the number of times that the thread was resumed because it reach 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.

TABLE IV: Causes for interrupting the stalls in the exposing mode.

Bench	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

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. Remember that the transactions begin when the lock acquired in the new atomic section belong to the list of locks that should trigger a transaction.

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 and after reach that limit, the code is reexecuted without transactional protection.

Table V shows the main characteristics of the transactions. #Tx are the number of transactions in the benchmark,

TABLE V: Transaction characteristics in the tolerating mode.

Bench	#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

#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 this 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 set 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 should be reasonable.

IV. 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 intrusion. 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.

References

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

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[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] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.