Scientific Report for STSM project

Lois Orosa
loisorosa@gmail.com

January, 2014

Reference Number	ECOST-STSM-IC1001-300913-034427
COST Action	IC1001
Action Description	A Hardware Approach for Detecting, Tolerating and Exposing High Level Atomicity Violations
STSM Period	from 30-09-2013 to 30-12-2013
Participant	Lois Orosa University of Santiago de Compostela
Host	João Lourenço, Universidade Nova de Lisboa

---

1 Purpose of the STSM

Providing multiple computing cores in a single chip is a main trend in multiprocessor construction technologies, aiming at achieving the craved performance without incurring in the exponential increase in the power consumption required when increasing the clock speed.

However, these new multiprocessor architectures are also forcing a shift in the software technologies towards exploiting parallel programming. And parallel programming is challenging because the programmer has to reason about many threads accessing data concurrently in non-deterministic and unpredictable interleavings.

Locks are among the most used mechanism to synchronize the accesses to shared memory. Using coarser grain locking enforces the serialization of large code blocks and hinders performance, while using finer grain locking is a tedious and error prone process, which frequently ends up in hard-to-debug deadlocks and other concurrency related errors. An alternative to locking is transactional memory (TM), an abstraction for defining atomic blocks that may be executed speculatively by a transactional run-time, enabling the use of all the available cores and simultaneously avoiding some of the major problems associated with locking.

As a matter of fact, most of the concurrent programs currently in production systems were written using locks, and nowadays most of the new concurrent programs are still using locks as the synchronization mechanism. Transactional memory is still a very young technology and is being considered mainly in research environments, but the landscape for this technology is very promising, as currently the main compilers include software support for TM abstractions at the programming language level, and the new chips from the main microprocessor manufacturers also include support for TM by hardware [23, 8]. Despite one of the main purposes of TM being to simplify th development of parallel programs and avoid some of the most common pitfalls from locking (over- and under-locking, deadlocks), some other concurrency bugs can still occur [10].

Debugging concurrency-related errors is a very hard task, and frequently the most tricky bugs take months (or even years) to be solved [24].

It is therefore of utmost importance to invest resources at simplifying the parallel programming models, making them less prone to errors, and to invest resources in new hardware and software tools to facilitate the debugging task. Many software and hardware tools have been proposed to help in the debugging task, covering a wide range of concurrency errors, including the detection of data races [5, 17, 6], deadlocks [3, 21], atomicity violations [11, 13, 12, 1], sequential consistency violations [19], asymmetric data races [18, 20], and also tools to help coping with the non-determinism of concurrent programs, such as deterministic replay tools [14, 16, 4] and behavior analysis tools [9].

The purpose of this short term scientific mission (STSM) was to design a tool that would tolerate atomicity violations in a transactional memory system implementing weak isolation. In these TM systems, the conflict detection mechanism only detects conflicts between transactions, but not between trans-

actional and not-transactional accesses. We consider the accesses to shared data not enclosed in a transaction as bugs (asymmetric data races, or transactional atomicity violations). From this point of view, we want to build a tool to detect and tolerate these bugs in a weak isolation TM system by prioritizing transactions over the buggy threads.

After some research on this topic, we decided to change it because we didn't find potential enough for making a significant scientific contribution. Instead, we decide to explore High-Level Atomicity Violations (HLAV) [1] in concurrent programs, a topic that has not yet been addressed in a hardware approach. HLAV result from the misspecification of the scope of an atomic block, by splitting it in two or more atomic blocks which may be interleaved with other atomic blocks. In this work we address a solution for detecting, exposing and tolerating this type of bugs.

There are software approaches that address the detection of HLAV using either dynamic [1] or static [7] program analysis techniques but, to the best of our knowledge, our approach is the first addressing the automatic detection of HLAV by hardware. The main advantages of using our hardware approach over the software approaches are:

- Detecting HLAV with negligible overhead and minimal influence in the normal flow of the program;
- In the debugging process, our module can force unlikely HLAVs to manifest by forcing the thread interleavings that trigger the error.
- In production runs, our module can be used to tolerate HLAV with low or even negligible performance penalty.

The hardware module we propose in this paper allows a quick detection of HLAV at runtime, and includes and additional mode to expose unlikely HLAVs that are difficult to manifest. Furthermore, the module is also useful in the post-deployment phase, where it can be used as a program healing infrastructure for tolerating existing HLAVs in production runs. As an additional value, as the module is developed using Bloom filters [2], with few hardware additions these can be used for other purposes, such as data race detectors [15], asymmetric data race detectors [18], code analysis and optimization [22], deterministic replay [14]....

2 Description of the work carried out during the STSM

2.1 Background

High level atomicity violations (HLAV) were first defined by Artho in [1]. In this section, we will first introduce some examples of HLAV and introduce some concepts to understand this type of bug.

```
1  atomic void getA() {
2    return pair.a;
3  }
4  atomic void getB() {
5    return pair.b;
6  }
7  atomic void setPair(int a, int b){
8    pair.a = a;
9    pair.b = b;
10 }
11 boolean areEqual(){
12   int a = getA();
13   int b = getB();
14   return a == b;
15 }
```

Figure 1: Example of high level atomicity violation.

The intuitive idea behind HLAV is that if two shared data items (e.g., memory locations) were both accessed inside an atomic block, they are interrelated and probably the programmer intention is that there shall be no interleavings between these two accesses. Therefore, if (in the same program) this two addresses are accessed separately in different atomic blocks, an unfortunate interleaving may cause an atomicity violation.

Figure 1 shows a typical example of this type of atomicity violation, where the variables a and b are accessed atomically in the function *setPair*, but separately in the check operation (in the atomic functions *getA* and *getB*).

Notice that HLAVs are very tight with the semantics of the program, and therefore a HLAV is not necessarily an actual bug (i.e, a HLAV could never manifest if there are proper synchronization mechanisms). Therefore, as a HLAV is not necesarily a concurrency bug, it is the programmers task to determine that.

For defining HLAVs, we need to define a **view**, that is the set of identifiers of the read and write data accessed when a thread executes an atomic block (i.e, a set of memory addresses). A view may distinguish between read and write sets [7] (avoiding some false positives due to false read-read conflicts). A view is **maximal (MV)** if it is not a subset of any other view in that thread [1].

A HLAV involves 3 views, two of them in one thread, and the third view has to be a maximal view from a different core. The condition for these three views to possible produce a HLAV is that the intersection of the two normal views with the maximal view are not null and their intersections form an inclusion chain [1]. The detection of HLAV may be done statically [7] or dynamically [1]. Our approach is dynamic, and because it works while the program is running and not post-mortem, it is restricted to only a few views (because of hardware limitations). For simplicity we assume each thread runs in a separate core.

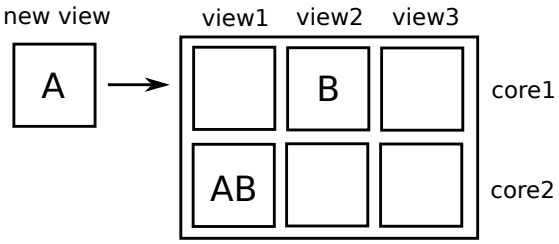

Figure 2: Example of high level atomicity violation, AV type 1 (real).

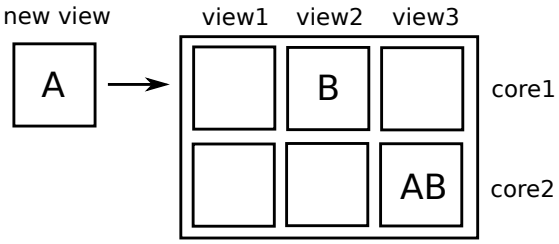

Figure 3: Example of a high level atomicity violation, AV type 1 (potential).

For implementing our module in hardware, we also need to define a **window** as the set of the last N views collected in a core. The resources in hardware are not unbounded, and therefore the detection of HLAV is restricted to this action window.

Figures 2, 3 and 4 show examples of the main types of high level atomicity violations. We classify the HLAV in two types, depending on the new view being a normal view or a maximal view. In these examples, there are 2 cores, and each core has a window of three views. The letters in the view represent data addresses. The empty views don't have any relevant information, the views are ordered chronologically from right (older) to left (newest).

Figure 2 is an example of what we call a real HLAV of type 1. The new view generated by core1 includes an access to the address A, the same core accessed previous the address B, and core2 accessed both variables AB in the same view between the accesses from core1. In this example the HLAV is manifested (the interleaving break the atomicity of the accesses to A and B).

Figure 3 is an example of a potential HLAV of type1. The difference to the previous example is that this bug is not actually manifested, but it could be if the interleaving changes (like in the example of Figure 2).

Figure 4 is an example of a potential HLAV of type 2 (it was not manifested because the adequate interleaving was not produced). The difference with the preious HLAV examples is that the HLAV is detected with the insertion of a maximal view. Notice that this type of HLAV can only be potential, as the real HLAV would be of type 1.

In the rest of the paper, we refer to the normal views of the cores as **core**

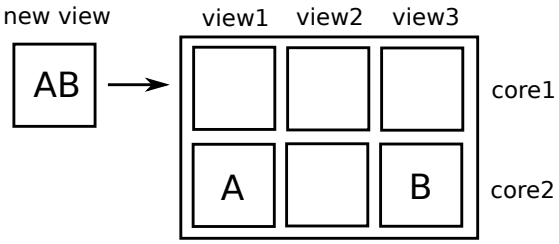

Figure 4: Example of high level atomicity violation, AV type 2 (potential).

views (or CV), and to the views that are maximal in a core as maximal views (or MV). We call **core window (or CW)** to the regular windows of the cores. Furthermore, we call **maximal window (or MW)** to a special window that maintains a set with the last maximal views of each core.

2.2 Overview of the Idea

The aim of this work is to detect, tolerate and expose HLAV in multicore processors with a hardware module placed in the system. This module is in charge of monitoring the views of all the cores, and to detect buggy interleavings, as well as potential bugs. Furthermore, in the exposing mode the module is able to stall a core to force a wrong interleaving, and in the tolerating mode, it protects potential HLAV by enclosing suspicious regions of code within transactions.

For better explain the algorithm, we describe the module in a system with a shared system bus, with the module directly connected to this bus. However, the module could be also used with some adaptations to another multicore architectures that not rely on a shared bus for communication. Furthermore, we will explain our system for detecting HLAV in lock-based programs, but it could be easily adapted for transactional-based programs.

Figure 5 shows a high level scheme of the hardware module placed in a multicore processor. The main modification in the cores is the inclusion of a local Bloom filter. Each core of the system has a Bloom filter where it inserts the addresses when it is in a critical section. We will explain the mechanism with only one Bloom filter (for both read and write sets) for simplicity, but with separate Bloom filters some false positives could be avoided [7] (because of the read-read conflicts). Also to simplify the general explanation, we assume that each core is single threaded.

With each lock acquire, the core clears its local Bloom filter, and starts to insert the read and write addresses in it. When the core finishes its critical section, the content of the Bloom filter is sent to the HLAV module through the system bus.

Each row in Figure 5 represents the window of views of one core. The new views are located in the place pointed by $last_x$, that corresponds to the older view of that window (and that will be replaced). Furthermore, there is another window, called maximal window (MW), that maintains a maximum of

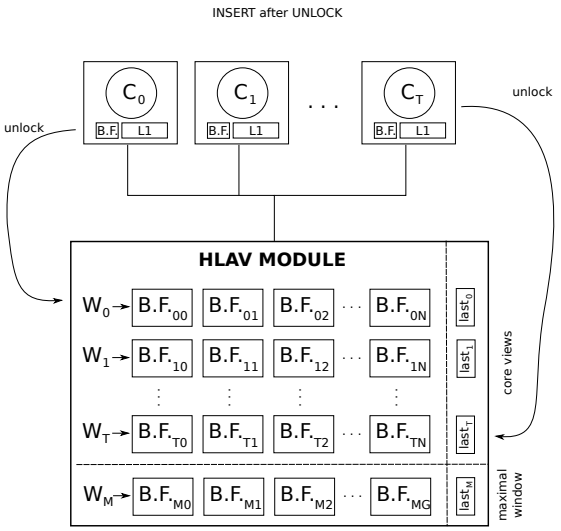

Figure 5: General high level scheme of the module.

G maximals views in the whole system.

The module has three operational modes:

- Detecting Mode: The module detects HLAVs.
- Exposing Mode: The module intelligently stalls cores to force buggy interleavings.
- Tolerating Mode: The module avoids suspicious interleavings between two atomic sections.

The basic functions of the controller are to receive the views from the cores, and to take the adequate actions depending on the mode.

2.3 Maximal Window

The module has a maximal window for keeping the maximal views that may cause HLAVs (there is only one maximal window for all the maximal views from all the cores). Because of the dynamic nature of programs, a view may be maximal at certain point of the execution, and stop being maximal later on (or the other way around). For this reason, we distinguish between several types of views in the maximal window:

- **Speculative Atomicity Violation (SAV) views**: are views that are in the MW, but they were not a superset of any view. We call them speculative because the module doesn't have any clue that this view will cause a HLAV. I.e, the first views of an execution will always be inserted in the MW as SAV view.

- **Possible Atomicity Violation (PAV) views**: these views are MVs in a thread, and they are a superset or intersects (but they are not equal neither a subset) with one view of another core. They are possible AV because at this point it matches the necessary conditions to cause a HLAV, but only involves 2 views (3 views are necessary to detect a real or potential HLAV).

- **Buggy Atomicity Violation (BAV) views**: these are MVs that are (or were) a superset of two different views of another thread, or that just intersects with both. Furthermore, there is a HLAV if the intersection of this two views with the BAV view should not form a chain. Unlike PAV views, this views are (or were) involved in a real or potential HLAV.

These three types of views also have a priority associated within the maximal window. The SAV views have the lower priority, PAV views have high priority, and BAV views also have high priority in the maximal window. A new view inserted in the MW can only replace a view with the same or lower priority (otherwise, the action is denied).

The low priority views have sense at the beginning of the execution of the program, when the maximal window is empty. If a low priority view in the maximal window finally results in a PAV or BAV view, the type of view is changed.

The main objectives of the maximal window are:

- To simplify the hardware logic of the module when it is detecting bugs. All cores have to check the intersections only with views in the maximal window, and not with all the windows of the system.

- To support more advanced features of the module (tolerate and expose bugs). The maximal views associated with the potential bugs are saved here, and the detection and tolerate modes are triggered based on the information associated with this views.

- To expand the coverage against bugs beyond the core window. When a view is eliminated from a core window, it is not deleted from the maximal window. This allows a longer coverage when the HLAV are not very frequent (e.g. in productions runs).

With this scheme, of the three views involved in HLAV, two of them are in the same core window, and the maximal view is in the maximal window (and belongs to a different core).

2.4 Detecting Mode

The module can detect the HLAVs of type 1 and type 2 described in Section 2.1. For implementing this mode we need some extra elements in the core views:

- **pAv**: indicates if there is (or not) a possible atomicity violation.

- **ptMax**: saves the position of the view in the maximal window involved in the possible atomicity violation.
- **Max?**: indicates if this view is maximal. We need to maintain these views in the core windows because it may stop being maximal later on.

The possible values of pAV are $NONE$, $POTENTIAL$ or $REAL$. The $NONE$ value indicates that the view has no relation (in the sense of a possible HLAV) with the maximal views in the maximal window. The $POTENTIAL$ value indicates that the view intersects (but it is not the same or a superset) with the maximal view pointed by $ptMax_{xy}$, and that can causes a potential HLAV (because the maximal view arrives sooner than the normal view, as in the example of Figure 3). The $REAL$ value also indicates a HLAV, but a real one, because the maximal view arrives later than the normal view (as in the example of Figure 2).

The extra elements of the views in the maximal window are:

- **coreIDs**: indicates the core(s) that produce that maximal view in some point. Even if the view is replaced in the normal views, the view remains in the MW (whereas is not replaced by another new maximal view).
- **Max?**: indicates the type of the maximal view (SAV, PAV or BAV), as described in Section 2.3.

For each new view inserted in the module, the module takes these actions:

Check for possible AV If the intersection of the new view with a view in the MW is a subset of the view in the MW, it sets the new view as possible causative of a potential HLAV (sets pAV to $POTENTIAL$, and it sets $ptMax$ to the index of the view in the MW). If the new view is maximal, and its intersection with a CV is a subset of itself, sets this CV and the new maximal view as possible causative of a real HLAV (sets pAV to $REAL$, and it sets $ptMax$ to the index of the new view in the maximal window).

Detect potential/real AV If the new view do not intersect with a view in the same window (or intersects but it does not form a chain) with pAV set to $POTENTIAL$ or $REAL$, and the intersections of both views with the MV pointed by $ptMax$ are not null and do not form a chain, a real/potential HLAV is detected.

Inserts the view in the maximal window In case the new view is a maximal view in one CW (because is a superset), it is inserted in the MW. It is also inserted in the MW with low priority in case it is not a subset of any view, and there is enough space for low priority views in the MW.

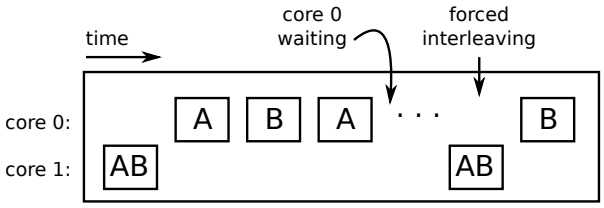

Figure 6: Example of the exposing mode.

Inserts the view in the normal window Finally, the view is always inserted in the CW, by replacing the older view.

When the module detects a conflict, it launches an exception, and the rest of the actions are managed by some software routines that will take the appropriate actions and possibly report to the user. The actions and analysis taken after the conflict detection are decisions taken by the programmer, and are out of the scope of this work. Notice that the content of the Bloom filters are software accessible through shared memory to facilitate the analysis of the HLAV.

2.5 Exposing Mode

To expose a HLAV, it is a requisite that the HLAV (real or potential) had happen previously. The module forces that future HLAV involving the same shared variables to be real (and not potential) HLAV. When a potential HLAV is detected, the associated maximal view in the MW is marked as an "exposable" MV. In subsequent inserted views, if the intersection of the new view with the maximal view (marked as "exposable") is a subset of the maximal view, the controller nacks the core (which becomes stalled) that sent the new view.

The aim of stalling the core is to force the interleaving of a MV from another core. Figure 6 shows and example where the core 0 is stalled for interleaving the view AB of core 1 between views A and B, which causes a real HLAV.

To avoid deadlocks, the core resumes after an established period of time, or after a predetermined number of views inserted in the module. This mode may cause some slowdown in the system because of the stalls, but since this is a mode for debugging, it is not critical.

To implement this functionality, the module needs two extra fields per core window. A *waiting* bit indicates if the core is stalled by the module, and the *waiting_num_inserts* field indicates the number of views inserted in the module since the core is stalled (when it reaches its limit, the core resumes). In the maximal window, each view has an extra *exposing* bit that indicates that the views related with that maximal view (the intersection of both is a subset of the maximal view) should stall the core to force buggy interleavings.

2.6 Tolerating Mode

The main aim of the tolerating mode is to avoid that real HLAVs to occur. For doing that, we use transactions to enclose the atomic regions that are suspicious of causing HLAV. This 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 main point to use transactional memory as tolerating mechanism is that it already has hardware support in new multicore processors [23, 8], not requiring extra-hardware in modern multicores for support it.

In tolerating mode, when a potential or real HLAV is produced, the MV involved is marked to be tolerated in future occurrences. For implementing the tolerating mode, besides the read set, we also need to keep the address of the lock.

When a potential HLAV is produced, the addresses of the locks of the two views involved in the HLAV (excluding the MV) are sent to the core (the last one is already in the core, stored in a special dedicated register) to a list of locks. Since that moment, before adquire a lock that is in the list of locks, the core starts a transaction.

The transaction commits in these cases:

- After one unlock, if the view does not belong to a possible HLAV (the case where the transaction only covers one atomic region).
- After two or more unlocks, when the transaction covers the two views involved in a potential HLAV.
- After reach the limit of the number of views protected.
- After reach the limit of the size of the data set.

To attach the case of nested locks, when a transaction begins, it saves the current nested level, and if at any moment during the transaction the nested level is lower than the one at the beginning, the transaction commits. Some deadlocks situations are avoided by enforcing this restriction.

Figure 7 shows an example of the tolerating mode. After detecting a potential HLAV (AB-A-B), all the subsequent atomics regions protected by the same locks than A-B are enclosed in transactions, and in case of conflict, the transaction aborts and restarts.

To support this functionality, each core window requires a $inTransaction$ bit that indicates if the core is in a transaction or not, and a $inTransaction\_num\_inserts$ field that indicates the number of views that were inserted in the module since the transaction starts (the transaction commits when the number of atomic blocks executed in the transaction reach a certain limit).

Furthermore, in each view we need to add a $tolerate\_view$ 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

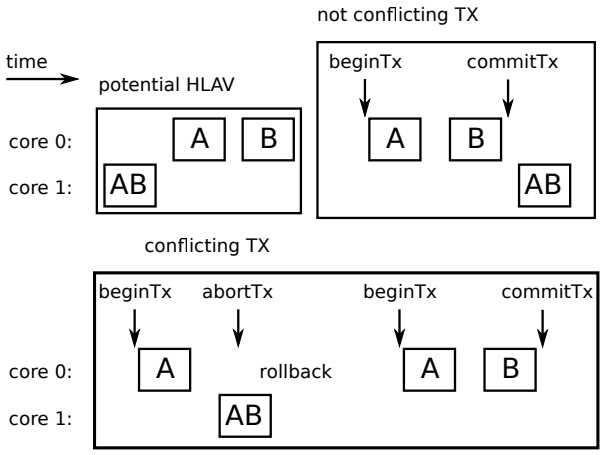

Figure 7: Example of the tolerating mode.

the window. In case of abort, the views are discarded (the Bloom filters are cleared).

Also, in the views of the maximal window there is a *tolerating* bit that indicates if the views whose intersection with the maximal view is subset of the maximal view should be protected by a transaction.

3 Future Collaboration with Host Institution

The participants intent to continue this collaboration to finish the evaluation of the module and to submit a paper to an international conference. I also applied for a PosDoc position to continue my work with João Lourenço in the Universidade Nova de Lisboa.

4 Confirmation by the Host Institution of the Successful Execution of the STSM

Prof. João Lourenço has sent the confirmation directly to the STMS coordinators.

References

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

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. Commun. ACM, 13(7):422–426, July 1970.

[3] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '02, pages 211–230, New York, NY, USA, 2002. ACM.

[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, ISCA '10, pages 187–197, New York, NY, USA, 2010. ACM.

[5] J.-D. Choi et al. Efficient and precise datarace detection for multithreaded object-oriented programs. In PLDI, 2002.

[6] 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, ISCA '12, pages 201–212, Washington, DC, USA, 2012. IEEE Computer Society.

[7] R. Dias, V. Pessanha, and J. Lourenço. Precise detection of atomicity violations. In A. Biere, A. Nahir, and T. Vos, editors, Hardware and Software: Verification and Testing, volume 7857 of Lecture Notes in Computer Science, pages 8–23. Springer Berlin Heidelberg, 2013.

[8] T. Jain and T. Agrawal. The haswell microarchitecture–4th generation processor. International Journal of Computer Science and Information Technologies, 4(3):477–480, 2013.

[9] 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, PADTAD '09, pages 3:1–3:9, New York, NY, USA, 2009. ACM.

[10] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In Proceedings of the 13th international conference on Architectural support for programming languages and operating systems, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM.

[11] 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, ASPLOS-XII, pages 37–48, New York, NY, USA, 2006. ACM.

[12] 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, ISCA '10, pages 222–233, New York, NY, USA, 2010. ACM.

[13] 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, ISCA '08, pages 277–288, Washington, DC, USA, 2008. IEEE Computer Society.

[14] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution ef?ciently. In Proceedings of the 35th Annual International Symposium on Computer Architecture, ISCA '08, pages 289–300, Washington, DC, USA, 2008. IEEE Computer Society.

[15] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas. Sigrace: Signature-based data race detection. In Proceedings of the 36th Annual International Symposium on Computer Architecture, ISCA '09, pages 337–348, New York, NY, USA, 2009. ACM.

[16] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs. In Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '10, pages 2–11, New York, NY, USA, 2010. ACM.

[17] M. Prvulovic and J. Torrellas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In International Symposium on Computer architecture, 2003.

[18] 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, pages 1 –12, feb. 2012.

[19] 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, ASPLOS '13, pages 535–548, New York, NY, USA, 2013. ACM.

[20] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman. Detecting and tolerating asymmetric races. In Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '09, pages 173–184, New York, NY, USA, 2009. ACM.

[21] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst., 15(4):391–411, Nov. 1997.

[22] 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, ASPLOS XIII, pages 145–156, New York, NY, USA, 2008. ACM.

[23] 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, PACT '12, pages 127–136, New York, NY, USA, 2012. ACM.

[24] 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, OSDI'10, pages 1–13, Berkeley, CA, USA, 2010. USENIX Association.