A Hardware Approach for Detecting, Exposing and Tolerating High Level Atomicity Violations

Lois Orosa   João Lourenço
CITI — Departamento de Informática
Universidade Nova de Lisboa, Portugal
loisorosa@gmail.com   joao.lourenco@fct.unl.pt

1. INTRODUCTION

Multicores are the main trend in computer architecture to gain performance without exponentially increasing the power consumption. However, to take advantage of these new architectures we need parallel programs. Parallel programming is challenging mainly because the programmer has to reason about many threads accessing data concurrently, and the data access interleavings are not deterministic, hence unpredictable.

Locks are the most used mechanism to synchronize the accesses to shared memory. Using coarse-grain locks enforces the serialization of large code blocks and hinders performance, and using fine grain locks is a tedious and error prone process for the programmer that frequently ends up in deadlocks and other concurrency related errors. An alternative to locks is transactional memory, an abstraction for defining atomic blocks that may be executed speculatively, making good use of the available cores and solving some of the problems associated with locks.

In this paper we address a solution for detecting and tolerating one of the most typical concurrency bugs: atomicity violations. More specifically, we address High-Level Atomicity Violations (HLAV). High-level atomicity violations 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. Figure 1 shows an example of this type of 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.

In the context of this paper, a view is the set of identifiers of the shared data items (e.g., address of memory locations) accessed in an atomic block, and a view is maximal if it is not a subset of any other view in that thread [1].

There are software approaches that handle HLAV dynamically [1] and statically [3], but ours is the first addressing the detection of HLAV by hardware. Other related works address the detection and toleration of other types of atomicity violations, such as asymmetric data races [6], and the detection of atomicity violations involving one variable (stale values). Other proposals are more ambitious [4] because their definition of atomicity violation is wider, but it requires more complex hardware and software modifications.

We are proposing a simple hardware module to detect, expose and tolerate *multivariable HLAV*. This module is based on signatures, and may be easily extended to support other functionalities, such as the detection of asymmetric data races [6], or code analysis and optimization [7].

```
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 high level atomicity violation.

The advantages of using our hardware approach are:

- Negligible overhead detecting HLAV, with minimal influence in the normal flow of the program;
- Exposing mode is useful for both, identifying improbable interleavings that cause HLAV, and for providing a run-time verification of suspicious interleavings.
- In production runs, our module can be used to tolerate HLAV with low or even negligible performance penalty.

2. A HARDWARE MODULE TO DETECT HLAV

Unlike software approaches, hardware solutions have to deal with very limited resources, valuing the good trade-offs between precision and complexity.

The general scheme for our approach is depicted in Figure 2. We only keep a bounded number of views per core, and each core has one local signature[1] that keeps track of the addresses of all memory location accessed in the last atomic block (the last view) or of the current accesses (if the core is currently executing an atomic block). When an atomic block finishes, the signature with the current view is sent to the HLAV module. Upon reception by the HLAV module,

[1]A *signature* may be implemented by a *bloom filter*.

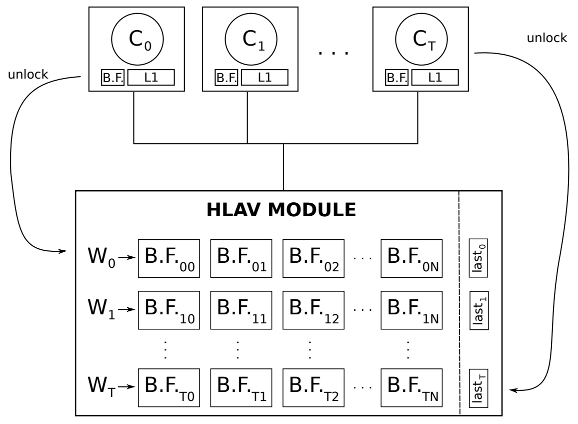

Figure 2: General picture of the module.

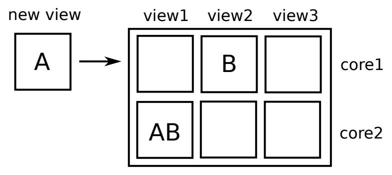

Figure 3: Example of atomicity violation detected by the module.

the view encoded in the signature is stored in the module's internal knowledge base and triggers a new checking phase for atomicity violations.

The HLAV module keeps a bounded window of the last views for each core. A larger window per core keeps more views and improves the precision of the approach, but also requires more hardware resources.

In Figure 3 we show an example of an atomicity violation detected by a module with a window of size three (keeping an history of 3 views). The module already contains two views, one with (AB) associated to core2 and another with (B) associated with core1. When the new view (A) associated with core1 enters the module, it fires the atomic violation checking process and an atomicity violation is detected due to the execution of the atomic region accessing AB in core2 in-between the execution of the accesses to (A) and (B) in core1.

To expose atomicity violations we need to collect information and estimate the potential future atomicity violations. Whenever such a potential AV is identified, we stall a core with the aim of achieving an interleaving that produces the bug.

For tolerating atomicity violations, besides collecting information about potential atomicity violations, we need to protect data identified as possibly involved in a HLDR. This protection may benefit from the transactional memory support by hardware already available in the most recent processors.

3. INITIAL EVALUATION

The HLDR detection module was implemented as an extension to Pin [5], following the algorithm as described in [3], with a window of five views per core to limit the addition hardware resources required. This experimental setup allowed to assert the effectiveness of the proposed architecture and confirmed that we were able to detect the same HLAV as the software approaches. However, this version of the module is quite complex, and other more simple approaches were required.

We have implemented a second version of the algorithm inspired in colorsafe [4], by coloring the views according with the relation of their data. Two views have the same color when they share at least one address. Abstracting the views using colors allows to simplify the hardware at the expense of missing some atomicity violations. Unlike colorsafe, the information for coloring the views is automatically collected at run-time, hence not needing annotations provided by the programmer nor by the compiler.

Currently we are implementing a third approach that includes a new window of maximal views, that keeps the last maximal views inserted in the module, and that allows to simplify the hardware and to extend the visibility of the bugs in some cases. Therefore, the results should be better that with coloring, because the algorithm is a variant of the first version [3].

We evaluated our module with Pin [5] and experimented with several well known benchmarks from the Parsec suite [2], as well as with a suite of atomicity violations from the literature [3]. We have simulated the first two versions of the module described above.

In the parsec benchmarks tested, we did not experiment any false positives with any of those two versions. However, in the examples of atomicity violations, we failed to detect some HLDR when using the coloring strategy.

4. REFERENCES

[1] C. Artho, K. Havelund, and A. Biere. High-level data races. In *Journal of Software Testing, Verification & Reliability (STVR)*, page 2003, 2003.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proc. of PACT*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[3] R. J. Dias, V. Pessanha, and J. M. Lourenço. Precise detection of atomicity violations. In A. Biere, A. Nahir, and T. Vos, editors, *Hardware and Software: Verification and Testing*, volume 7857 of *LNCS*, pages 8–23. Springer Berlin Heidelberg, 2013.

[4] B. Lucia, L. Ceze, and K. Strauss. Colorsafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In A. Seznec, U. C. Weiser, and R. Ronen, editors, *ISCA*, pages 222–233. ACM, 2010.

[5] 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 *Proc. of PLDI*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[6] S. Qi, N. Otsuki, L. Orosa, A. Muzahid, and J. Torrellas. Pacman: Tolerating asymmetric data races with unintrusive hardware. In *Proc. of HPCA*, pages 1 –12, feb. 2012.

[7] J. Tuck, W. Ahn, L. Ceze, and J. Torrellas. Softsig: Software-exposed hardware signatures for code analysis and optimization. In *Proc. of ASPLOS*, ASPLOS XIII, pages 145–156, New York, NY, USA, 2008. ACM.