Hardware Signatures in Transactional Memory: A Review

Felipe L. Madruga, Philippe O. A. Navaux
Institute of Informatics
Federal University of Rio Grande do Sul - UFRGS
Caixa Postal 15.064 - Porto Alegre - RS - Brazil
{flmadruga,navaux}@inf.ufrgs.br

Abstract

One of the key aspects of a Transaction Memory system is to detect conflicts by tracking reads and writes inside transactions, which is intuitively done by the hardware. There are several ways to provide conflict detection via hardware mechanisms. A promising mechanism is to track reads and writes and summarize the addresses in a special register (also called signature) that represents the set using few hardware resources.

In this paper, we survey hardware signatures for conflict detection in transactional memory, provide insights about design space exploration, and show how they can be used in a transactional memory system. Hence, this document presents a review and motivation on the subject.

1. Introduction

The limits on instruction-level parallelism extraction and power dissipation [1] of high performance processors led chip manufacturers to put more processing elements on a physical processor. However, parallel programming is much more complex than sequential programming. A famous illustration of this is that to implement a queue data structure is a simple task of any programming course and making the same algorithm parallel and thread-safe is a publishable result.

In order to facilitate the task of coordination to concurrent accesses to shared data structures, a mechanism called Transactional Memory inspired in database transactions was proposed by Herlihy [7]. The majority of researchers on the subject believe that the programmer must only label sections of the code as **atomic** and the underlying system must deal with it to make the execution of this code behave atomically, isolated from other transactions and maintaining consistency of the globally viewed state of the shared memory.

Several proposals were made since the creation of the technique. Some of them require only software and compiler modifications [4][5], other provide the functionality totally with hardware mechanisms [6][11]. Due to disadvantages of both approaches, hybrid systems [10] are more likely to be implemented in real, general purpose machines.

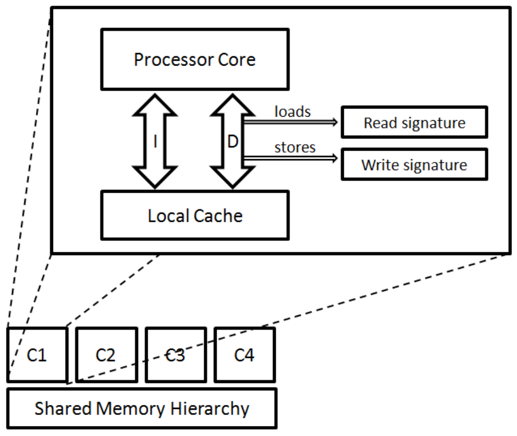

Figure 1. Block diagram of a system using hardware signatures.

One of the key problems in Software Transactional Memory (STM) is that all reads and writes are expanded to several instructions and to verify if there are conflicts between transactions is difficult. Sometimes, it is needed to cover big-sized data structures in order to do so. There are several ways to provide conflict detection via hardware mechanisms. One promising mechanism is to track reads and writes and summarize the addresses in a register (also called signature) that represents the set using few hardware resources. A rough block diagram representing the scheme is illustrated in Figure 1. There, all data accesses are sent to a hardware signature to be used for detecting conflicts. In some implementations the signature system tracks the coherence requests instead of memory accesses, which provides early conflict detection.

In this paper, we survey hardware signatures for conflict detection in transactional memory, provide insights about design space exploration, and show how they can be used in a transactional memory system. Hence, this document presents a review and motivation on the subject.

The rest of this paper is organized as follows. Section 2 describes TM systems and what are the basic parameters of a typical implementation. Section 3 shows the dimensions of a hardware signature for TM and explores them. Section 4 shows one example of a TM using hardware signatures. Finally, section 5 concludes the paper and introduces future work.

2. Transactional Memory

Basically, TM is a synchronization mechanism where the programmer defines sections of code to be treated as transactions. One of the greater motivations for this mechanisms is that most of the synchronization primitives currently used, notably the locks, are very error prone and difficult to use.

Locks have high probability of programming errors, such as lock inversion and forgetting to access the set of locks necessary to ensure correctness. Other problem is that it is very hard to use it on complex situations and it is not composable, which is an important characteristic for software reuse.

Compared to commonly used pessimistic primitives, TM is much simpler to use correctly although it is not free from programming errors. Besides, it can yield better performance by trying to execute potentially conflicting sections of code in parallel.

Some concepts from database world were ported to transactional memory. For example, a transaction is said aborted if its execution cannot proceed and a previous state has to be somehow restored. Likewise, a commit happens when a transaction gets to its end and the rest of the system can see its modifications.

TM is a great research topic, not only because of the benefits it brings up for parallel programming but also because it is very complex to implement. The most costly things it needs that are not trivially done are: detect conflicts on transactions with unbounded size and working set, rollback the execution on aborts or update global memory on commits, and manage more than one version of data accessed inside the transaction.

2.1. Software versus Hardware

The implementation of the TM can be done completely in software, completely in hardware or in software with mechanisms in hardware.

Software implementations have much overhead because they increase the number of instructions executed in very frequent operations (read and write). Besides, is necessary to keep several data structures in memory and execute some non trivial algorithms to provide TM efficiently. The main advantage is flexibility, because TM algorithms continuously evolve.

Designs completely in hardware have better performance in some cases, but in several proposals extrapolating the hardware resources (eg, size of cache memory) severely hurts performance. Besides having total inflexibility, the overheads of STMs are so serious that use hardware is practically inevitable. Combining both hardware and software is the most promising technique, because it can benefit from hardware and use the flexibility and the unbounded nature of software mechanisms.

2.2. Implementation Taxonomy

Implementing a transactional memory system involves several properties and design dimensions. The various workloads for TM require very different behavior, with neither system achieving better performance in all cases [9]. Broadly speaking, transactional memory systems can be classified according to some characteristics [8] discussed here.

The main dimensions we discuss here are the following.

- **Conflict Detection:** It can be **early** if conflicts are made on each access or **late** if the conflicts are detected only at the end of the transaction.

- **Version Management:** A system can directly update the global memory, keeping old values in an undo log for restoring on abort, or can defer the updates to the end of the transaction, by writing in a write buffer and updating the global copy on commit.

- **Granularity:** defines the unit that a TM system uses to detect conflicts. It can be by blocks of memory (cache line), object or machine word.

Another issue discussed in the academia is to define what happens when a conflict between transactional and non transactional code is raised. If this situation is treated by a system, it has strong isolation guarantees, if not it has weak isolation guarantees.

3. Hardware Signatures for Transacional Memory

Several proposals of hardware transactional memory add bits to cache lines and use them to track conflicts in a multiprocessor with coherent caches. The problem with this approach is that when a transaction extrapolates the total cache size, a software mechanism is called, hurting performance.

Some proposals [3][11][10] use a fixed-size structure to represent the transactions read and write set and detect conflicts. By using a probabilistic structure (smaller than the original set) it has to deal with false conflicts, which may hurt performance but does not make execution incorrect. In this section, we summarize concepts and make discussions based on [12].

3.1. Using Bloom Filters

The most used scheme is to implement a Bloom filter [2] in hardware. A Bloom filter is an array of m bits and a set of k hash functions, three basic functions are provided: insert, lookup and reset. An insertion is done by hashing the address using the k hash functions and inserting the corresponding bits in the signature. Lookup consists of calculating the hash functions and verifying if the bits are set. In this way, it is possible to have false positives, which should happen more frequently as the number of addresses increases. A Bloom filter with three hash functions is illustrated in Figure 2 (a).

One way to implement bloom filters is to track coherence requests. Roughly speaking, a processing core in a cache coherent multi-processor can broadcast requests to share a line (on a read) or to have exclusive access to it (on a write). The signature system must then track this requests to analyze if the working sets of more than one transaction conflict. This makes small modifications to the underlying coherence protocol. Other way to use hardware signatures is to extend the bloom filters to provide the intersection operation.

3.2. Exploring Hardware Implementation

The intuitive implementation of a bloom filter uses the same number of write ports than the number of hash functions implemented. This increases the area of the bloom filter significantly.

One alternative which aims better use of hardware resources is to use more than one register, one for each hash function. Comparing to a single bloom filter with m bits and k hash functions, a parallel bloom filter would have k hash functions and a (m/k)-bit register for each hash function. Parallel Bloom Filters are formally proven to perform the same as Single Bloom filters when $k/m << 1$. Parallel and Single Bloom filters are showed in Figure 2.

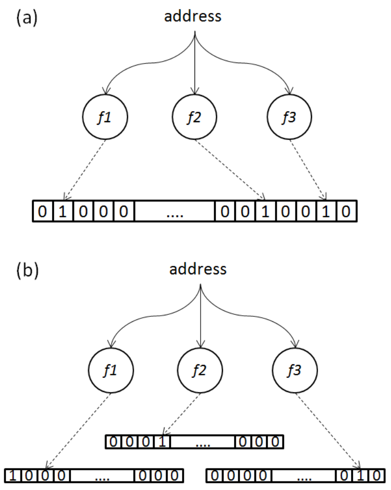

Figure 2. Two possibilities for signatures: (a)Single Bloom Filter and (b)Parallel Bloom filter

4. Using Signatures in Hybrid Transactional Memory

In this section, we overview how signatures can be used to build an hybrid transactional memory. The Algorithm 1, from [10], shows the basic operations of SigTM. It is a signature-based hybrid transactional memory derived from a well known STM ([4]).

The $SigTMtxStart$ creates a checkpoint for restoring the thread execution state. $SigTMwriteBarrier$ and $SigTMreadBarrier$ regard to writes and reads, respectively, to global data inside a transactions. $SigTMtxCommit$ is called at the end of the transaction, in order to try to commit.

At each write, the address is inserted in the signature and address with data is inserted in a software undo log. On a read, the address is checked in the reader's write signature to decide if the data will be read from the global memory or from the undo log. If an exclusive request from one transaction in another processor is present in the write signature, a conflict is detected and the transaction is aborted.

This mechanism can be used also for object-based languages. One way to do so is to insert only the address of the headers of the object in the signature for complex classes and do conflict detection in a cache line or word granularity for multi-dimensional arrays.

Algorithm 1 Pseudocode for the basic functions in SigTM.

```
procedure SIGTMTXSTART
  checkpoint()
  enableRSlookup($exclusive$)

procedure SIGTMWRITEBARRIER($addr$, $data$)
  wsSigInsert($addr$)
  $writeSet$.insert($addr$, $data$)

procedure SIGTMREADBARRIER($addr$)
  if wsSigMember($addr$) and $writeSet$.member(addr) then
    return $writeSet$.lookup($addr$)
  rsSigInsert($addr$)
  return Memory[$addr$]

procedure SIGTMTXCOMMIT
  enableWSlookup($exclusive$, $shared$)
  for every $addr$ in $writeSet$ do
    fetchEx(addr)
  enableWSnack($exclusive$, $shared$)
  rsSigReset()
  disableRSlookup()
  for every $addr$ in $writeSet$ do
    Memory[$addr$] $\leftarrow$ $writeSet$.lookup($addr$)
  wsSigReset()
  disableWSnack()
```

This is possible to do only by modifying the way software uses the signatures, if there is a clear interface to manipulate it. We believe this is an important characteristic, since object-based programming languages are much used in general purpose computing.

5. Concluding Remarks

In this paper, we discussed a mechanism for detecting conflicts on a transactional memory system called hardware signatures.

The hardware implementation was also covered, comparing Single Bloom Filters and Parallel Bloom Filters. We presented also how a hardware signature can be used by a STM with deferred update and early conflict detection. Hence, we provided an introduction to TM and hardware signatures.

As future work, we intend to analyze real address streams to evaluate some hardware signature configurations in the context of transactional memory and use the most promising alternatives to propose a hybrid TM system.

References

[1] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. *ACM SIGARCH Computer Architecture News*, 28(2):248–259, 2000.

[2] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970.

[3] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. *ACM SIGARCH Computer Architecture News*, 34(2):227–238, 2006.

[4] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proc. of the 20th Intl. Symp. on Distributed Computing*. Citeseer, 2006.

[5] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246. ACM New York, NY, USA, 2008.

[6] L. Hammond, V. Wong, M. Chen, B. Carlstrom, J. Davis, B. Hertzberg, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04)*, volume 1063, pages 20–00, 2004.

[7] M. Herlihy and J. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300. ACM New York, NY, USA, 1993.

[8] J. Larus and R. Rajwar. Transactional memory. *Synthesis Lectures On Computer Architecture*, 2007.

[9] C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46, 2008.

[10] C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA'07: Proceedings of the 34 th Annual International Symposium on Computer Architecture; San Diego, CA*, pages 69–80. Association for Computing Machinery, Inc, One Astor Plaza, 1515 Broadway, New York, NY, 10036-5701, USA,, 2007.

[11] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. LogTM: Log-based transactional memory. In *Proc. 12th Annual International Symposium on High Performance Computer Architecture*, 2006.

[12] D. Sanchez, L. Yen, M. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 123–133. IEEE Computer Society Washington, DC, USA, 2007.