Towards fair, scalable, locking

Enrique Vallejo[1], Sutirtha Sanyal[2], Tim Harris[3], Fernando Vallejo[1], Ramón Beivide[1], Osman Unsal[2], Adrián Cristal[2] and Mateo Valero[2]

[1]University of Cantabria
Avda. Castros S/N
Santander, Spain
+34 942202039
{enrique, fernando, mon}@atc.unican.es

[2]Barcelona Supercomputing Center
C/Jordi Girona, 31
08034,Barcelona, Spain
+34 934137716
{sutirtha.sanyal, osman.unsal, adrian.cristal, mateo.valero}@bsc.es

[1]Microsoft Research
J J Thomson Avenue
Cambridge, UK
+44 1223 479000
tharris@microsoft.com

ABSTRACT

Without care, Hardware Transactional Memory presents several performance pathologies that can degrade its performance. Among them, writers of commonly read variables can suffer from starvation. Though different solutions have been proposed for HTM systems, hybrid systems can still suffer from this performance problem, given that software transactions don't interact with the mechanisms used by hardware to avoid starvation.

In this paper we introduce a new per-directory-line hardware contention management mechanism that allows fairer access between both software and hardware threads without the need to abort any transaction. Our mechanism is based on "reserving" directory lines, implementing a limited fair queue for the requests on that line. We adapt the mechanism to the LogTM conflict detection mechanism and show that the resulting proposal is deadlock free. Finally, we sketch how the idea could be applied more generally to reader-writer locks.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures.

General Terms

Design.

Keywords

Hardware Transactional Memory, reader starvation, synchronization.

1. INTRODUCTION

Our work is looking at the problem of providing reader-writer locking of data with the aim of supporting (i) fine-grained critical sections that may perform only a small number of memory accesses, (ii) longer critical sections during which threads may be descheduled, (iii) scalable critical sections, in the sense that the implementation should not introduce contention between concurrent readers, or between access to distinct critical sections, and (iv) fair access to critical sections, in the sense that writers should not be starved by a changing set of concurrent readers.

Existing approaches to reader-writer locking do not provide all four of these properties. For example, hardware transactional memory (HTM) can be used to implement fine-grained scalable critical sections by using hardware support to allow concurrent readers to access data in parallel along with low overhead entry and exit of critical sections. However, certain HTM implementations can allow a "starving writer" pathology [1] in which a set of readers continually prevents write access being granted.

Software implementations of reader-writer locking provide the flexibility to express different fairness properties, for example Mellor-Crummey and Scott's fair-MRSW queue-based locks [10] do this by delaying read access to a lock when there is a waiting writer. This policy prevents writer starvation. However, entering and leaving a queue-based lock requires atomic compare-and-swap operations on shared fields (e.g. to maintain a reader count or to construct new queue nodes), causing contention in the lock's implementation and limiting its scalability for fine-grained critical sections.

The approach we are investigating is to provide additional hardware support to try to combine the four desirable properties that we seek. In overview we wish to use LogTM-style HTM to support fine-grained critical sections and to then fall back to using explicit queue-based spin locks to support longer critical sections. We thus hope to reduce the overheads of using queue-based spin locks everywhere, while still providing the flexibility to express different policies, integration between the lock implementation and the scheduler, and so on.

In previous work we investigated this in the context of programs written using transactions rather than explicit critical sections. We examined, in simulation, a hybrid transactional memory using LogTM in hardware for executing short-running transactions, and falling back to Fraser's STM built with queue-based spin-locks for longer-running or larger transactions. The resulting system proved to obtain a significant speedup over the baseline STM by removing some of the inherent costs such as managing read-set

EPHAM'08, April 6, 2008, Boston, Massachussetts, USA.

and write-set data structures and managing the queue-based locks themselves.

In this paper we look at the other part of the problem: how to control fairness within the HTM so that (for example) writers cannot be starved. Our approach is to extend a directory-based cache to record a single "reservation" for each line. The reservation records which processor should be next to receive access to the line. For example, a processor wishing to obtain exclusive mode to the line, but finding itself starved by processors holding the line in shared mode, can establish a reservation and, after doing so, no subsequent shared mode access will be admitted.

This is preliminary work: we are working on a simulation of the technique, and we are also working on building queue-based spin-locks directly over it (for use from programs that are written using explicit locking, rather than our current focus on programs that are written with transactions).

2. BACKGROUND: STALLS IN THE BASELINE HTM

Eager update HTM systems update memory values "in place" and maintain the previous contents in an undo-log so that they can be written back to memory in case of conflict. Directory-based HTM with eager conflict detection relies on the coherence requests to detect a conflict with such a previously modified block. To implement this correctly in an eager-eager directory based system like LogTM [12] coherence requests are extended: whenever a transactional request arrives at a given node, if the request conflicts with the ongoing transaction a NACK ("Negative Acknowledgement") reply is sent to the coherence requestor, temporarily denying access to the line. This prevents the requester from reading or writing transactionally modified lines.

Thus, this NACKing mechanism effectively provides a hardware-based lock on those lines read or modified during the transaction. It can be either read-locking when a transaction reads a line, in which case several transactions can concurrently access the line in shared state; or write-locking if the line has been modified by a transaction and is kept with exclusive coherence permissions.

This kind of multiple-reader, single-writer locking can lead to writer starvation on frequently read lines, as previously presented in [1]. This pathology is not specific to LogTM only, but to any HTM with eager conflict detection that stalls the requestor of conflicting addresses. This problem can occur if two processors are continually running transactions that hold the same cache line in their read set while a third processor is waiting to make a transactional write to that line. The putative writer will be continually NACKed while the readers continue executing transactions. A pathological example is shown in the code in Figure 1, where *a* is a shared variable initially set to 0, *N* is the thread count and *th_id*<*N* is a per-thread id. When there are enough threads running this code, execution never ends due to writer starvation; In our experiments we found that, without especial congestion management, four threads are enough to block the system.

Figure 2 shows the coherence requests involved in this situation, where processors A and B are the readers holding the line containing *a* in shared mode, while C wants to update *a*. C sends a GETX ("get exclusive") message to the directory which is forwarded to the line's holders A and B. As long as A or B holds the line and is running transactionally then it will send a NACK to C. If A or B commits while the other remains in the transactional state, and then starts a new transaction that reads *a* again before the other's commit, the situation will persist.

```
while (a < 1000){
  atomic{
    if ((a %N)==th_id) a++;
  }
}
```

Figure 1: Example code that stalls due to writer starvation

The solution presented in [1] relies in writers detecting that they are being starved and choosing to abort the readers that are obstructing them. This can be done by maintaining timestamps. While this removes the problem, it is only applicable to collisions between different HW transactions, not those between HW transactions and ordinary, non-transactional code. Also, the timestamp-based approach can cause unnecessary transactional aborts, as we will show later.

Hybrid Transactional Memory systems make use of HW transactions when possible, and otherwise run the original STM code. There are recent proposals (such as [2] and our own subsequent work [15]) designed to make use of generic HTM support. To allow for correct execution, HW transactions are typically extended to read and write parts of the STM's concurrency-control data structures so that conflicts between HW and SW transactions are detected. This can improve performance over a pure-SW system because, when running in HW mode, several aspects of the STM are unnecessary, such as read and write set validation, commit copy of new values, and read and write sets management. After one or several aborts, the transactional mode of a processor is switched to software-only (SW) to execute the conflicting transaction.

The previously presented writer starvation problem is even more important in SW transactions in a hybrid system, given that the solution presented in [1] wouldn't allow them to proceed. We have used our lock-based Hybrid TM system presented in [15] to simulate a red-black tree microbenchmark. We found that, with 32 threads and 32 processors, after starting a couple of thousand transactions, and depending on the program run, from 4 to 12 processors are stalling trying to modify some node which is frequently read because of its location close to the root.

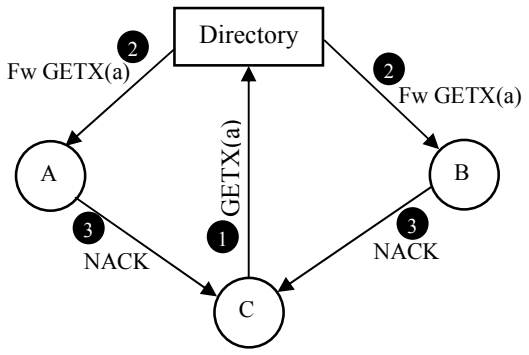

Figure 2: Writer Starvation in LogTM

This starvation does not happen in the original lock-based STM, given that locks protecting STM objects are implemented using *fair queues* [10]: Each thread wishing to acquire the lock in read or write mode joins a queue of waiters in arrival order. The queue management prevents a reader from acquiring the lock if there is a previous writer waiting. This means that a thread wanting to write-lock a lock currently in read mode will have to wait for previous readers to finish, but no new read locks will be granted. Of course, absent contention, the baseline performance of this pure STM is poor compared with that of the HTM.

The idea in this work extends this fair queuing mechanism to the access to memory lines in presence of a directory-based implementation of HTM. We introduce *Directory Reservations*, a novel mechanism that enables threads to "reserve" directory lines access once they are NACKed by others, preventing any newcomer from accessing the line before the reserver. We show how this is equivalent to a limited form of fair queue, and present an extension to provide stronger fairness guarantees. This mechanism allows HW and SW transactions to coordinate access to frequently read and modified lines, without the need to abort remote transactions.

The rest of the paper is organized as follows. Section 3 presents the general idea and details the hardware requirements. Section 4 introduces the specific details to make the directory reservations idea work with a LogTM-based Hybrid TM model. Section 5 presents some related work and we conclude with further line of work in section 6.

3. DIRECTORY RESERVATIONS: GENERAL IDEA

The general idea of Directory Reservations is that NACKed requests, such as those presented in Figure 2 (steps 1—3) will issue a reservation (RESERV) request to the directory to reserve the line. The directory is extended with new fields to support the new functionality, as presented in Figure 3. Figure 4 shows the new behavior: After receiving a reservation request (4) from processor *C*, the directory sets the *R* flag for the line (*R* for *Reserved*) and records the requestor processor id *C* in the *requestor* field. An acknowledge message is sent to the requestor (step 5). The fields *read_count* and *W* will be discussed later.

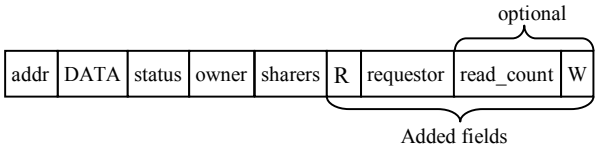

Figure 3: Directory additions

Whenever any other processor *D* issues a GETX ("get exclusive") or GETS ("get shared") request for the same line (marked as step 6 in Figure 4), the request will arrive at the directory controller where the *R* flag is already set. After checking this flag, the directory controller will compare the requestor id, *D*, with the saved requestor field containing *C*. Being different, the controller determines that the current requestor is not the one that reserved the line, and sends a NACK message (step 7) to *D* without any need to forward the request to the current sharers.

However, if the requestor of the GETX or GETS is processor *C* (i.e. the one that reserved the line), the request will be forwarded

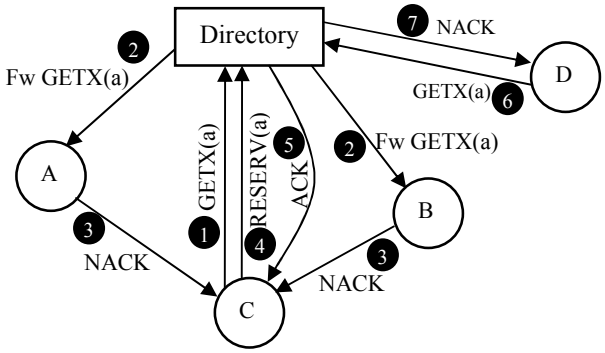

Figure 4: Reservation mechanism

to the corresponding processors (owner and sharers). If *C* receives new NACK replies, such as presented in Figure 2, it will have to repeat the request until it is successfully satisfied.

Eventually, in an idealized case, the blocking processors (*A* and *B* in Figure 2) will commit their transaction. When this happens, no new NACK will be issued to coherence requestor, so *C* will receive the valid data with the valid permissions. In this point, the final message from *C* to the directory clears both the *R* flag and the current *requestor*, and finishes the reservation.

However, in practice we must be careful because the processors executing *A* or *B* may themselves incur a conflict with a transaction executing in *D* (for example, this may be due to the transactions accessing different variables or objects that map to the same cache lines, different from *a*). This would cause a deadlock, e.g.: *A* waiting *D*, *D* waiting for *C*, and *C* waiting for *A*. This can be addressed by extending the existing deadlock avoidance mechanism used in the HTM. In Section 4 we will specify how to combine our mechanism with the original deadlock avoidance mechanism in LogTM.

3.1 Limited fair queuing

The proposal as previously described enables any writer to proceed execution after the current holders of the line commit. However, it does not implement any queue for the remaining readers or writers. Once the reservation is cancelled, the rest of the requests will race for the line. Now we describe an optional alternative implementation providing a result equivalent to a limited fair queue.

We make use of the optional *read_count* field and *W* flag in the directory. Once the reservation has been set, any GETS request for the same line will be NACKed as explained previously, and the *read_count* will be incremented. To prevent counting the same read request twice, every request message includes a *nack_count* field[1], which is increased by the requestor cache on every retry. Only requests with *nack_count* = 0 increase the *read_count* in the directory. On the first GETX request not coming from the original requestor *C*, the *W* flag is set, and *read_count* is no longer incremented.

This ensures that if the block is requested in exclusive mode during a reservation, the *read_count* field will contain the count

[1] In fact, a single bit is enough for this purpose, but we consider a counter for future thread de-scheduling detection mechanisms.

of previous read requests for the block, which will have to be served before acknowledging any exclusive request. Once the original reservation is served, the directory will continue to keep the *W* flag set, and decrease *read_count* on every GETS request served. When *read_count* reaches 0 and the *W* bit is set, only a single GETX request will success (and, in case of a new conflict, generate a new reservation). Meanwhile, exclusive requests are NACKed by the directory.

This design does not implement a real queue, given that the directory is not aware of the identity of read and write requestors. Once the original reservation is served, only the amount of read requests before any write request will be preserved. If new readers try to access the line, nothing prevents them from doing so before the next writer succeeds. If a new writer comes and wins the request race, its request will be satisfied. However, this is enough to make sure that the *proportion of sharers and writers* in the queue is satisfied. We consider that this mechanism is fair in that, on average, the waiting times for sharers and writers is the same as it would be with a real queue.

3.2 Thread de-scheduling and migrating

Given that LogTM transactions block accesses that conflict with its read or write set, thread descheduling is an important issue for HW transactions. Signature-based solutions for this have been proposed in [16].

However, in this section we present how to prevent starvation when a thread waiting for a reserved line is de-scheduled. If the thread holding the reservation is de-scheduled by the OS, when the resource becomes free, there will be no request for the line. This will prevent other threads from accessing the line.

This case does not generate a deadlock, but a temporal starvation; in the same manner of thread de-scheduling for a thread which is waiting in a queue. To cover this last case, in [8] a new mechanism is proposed to detect threads that have been descheduled. Waiting threads periodically "publish evidence" that they are still iterating, in the form of a timestamp increase. If other thread finds that this timestamp has not been increased in a long time, it can "jump ahead" the queue.

In our case, we might consider a timer in the directory (not depicted in Figure 3), which cancels the reservation when it expires. This timer is reset on every GETX request received from the processor holding the reservation. The timer duration will be set to several times (2 or 3) the delay between requests, to cover the case of network congestion delaying a request. A similar case must be considered for readers and writers if using the limited fair queuing proposed in section 3.1.

3.3 Directory compacting

The directory block presented in Figure 3 includes a significant memory overhead for this mechanism: two new flags, the requestor id field and a new counter on each memory block. However, it can be simply reduced by adding an additional Reservation Table (RT) in the directory to hold reservation values. This RT, with a low amount of blocks would contain all fields except the *R* flag, and would be addressed by the block address. We consider that 8 or 16 entries will do the work most of the time, as we expect that few processors hold reservations, and only on a single line each, distributed across all of the directory

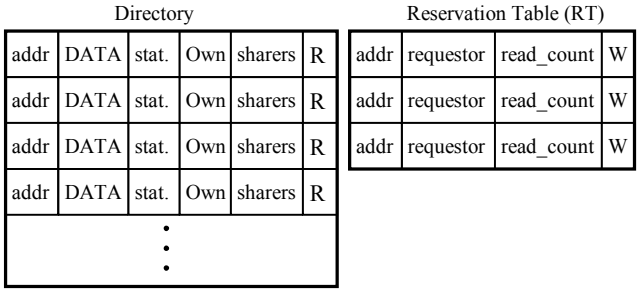

Figure 5: Compact directory implementation

controllers. Figure 5 shows an example with a RT with three entries.

With this design, when a given directory request finds the *R* flag set, the RT is searched for an entry containing the given block address, containing the remaining fields. If a new reservation comes and there is no empty line, the reservation won't be made.

The additional area overhead of the directory is minimal: one bit per line. Moreover, the addition of the *R* bit doesn't affect the directory access time, given that its check can be made in parallel with other fields, such as the status of the line. Delay is only increased in the case of the reservation being active, which is the uncommon case and not in the critical path. In that case, considering that involved processors are waiting for each other, which is a long wait by nature, the increase of a few cycles shouldn't affect performance significantly.

Finally, the amount of coherence messages in the network, and thus the bandwidth used, are not seriously increased by the mechanism. The reservation request (step 4 in Figure 4) is needed to end the NACKed memory operation in the directory side. Repeated requests that are NACKed while a line is reserved are required to provide fairness, and the use of an exponential backoff mechanism would reduce congestion in the access to the directory controller. The rest of the messages correspond with the original coherence requests.

4. DIRECTORY RESERVATIONS FOR LOGTM-BASED HYBRID SYSTEMS

To integrate this mechanism with LogTM, some changes have to be applied to the general idea in Section 3. First of all, the processors that are sharing a line and NACKing a request (*A* and *B* in Figure 4) need to invalidate their local copy of the line after commit. Otherwise, they might start a new transaction after commit that also reads the same line. To this end, we add a new *requested* flag to the L1 caches, which is set by a directory indication when the reserved block is requested. When the processor commits, all of the *requested* lines in the local cache are invalidated, or sent back to the directory if they have been modified.

As commented in Section 3, the general scheme might deadlock when used with LogTM. In the original LogTM proposal in [12] the deadlock avoidance system is based on timestamps for each transaction. The timestamp is, essentially, the clock cycle in which a transaction begins, and is held for the whole transaction. Thus, "older" transactions have lower timestamps. There is no serialization or commit arbitration based on the timestamp; it is used only to prevent deadlock. This mechanism works as follows:

if a given processor sends a NACK to other processor, and then receives a NACK from another processor, there might be a deadlock in the system. In this case, there will be at least one processor in the dependency cycle having NACKed an older processor, and having received a NACK from an older processor (lower timestamp). In this case, this processor aborts its own transaction. To detect this case, each processor has a given *possible_cycle* bit which is set when the processor sends a NACK to an older one, and in case of receiving a NACK from an older processor with the bit set the processor aborts. This solves the deadlock problem, without aborting all of the transactions in the cycle. However, there are still some "false positive", in the sense that processors can still abort in the absence of any cycle in the system, but this rate is low.

It is possible to adapt the general idea of Directory Reservations to the deadlock avoidance mechanism in LogTM. Here we outline the basic mechanism. Basically, the idea is to add a new field to the Reservation structure containing a timestamp (not depicted in Figure 5). The behaviour is handled as follows, considering the example in Figure 4:

- When processor *C* requests *a*, it receives NACKs from processors *A* and *B*. These replies contain *A* and *B*'s timestamps.

- In the reservation message, *C* sends the newest timestamp received, which will be stored in the Reservation structure as the "reservation timestamp".

- Whenever any request for *a* is received in the directory:

    - If the request comes from a processor not in HW transactional mode, the request is NACKed by the directory and the processor has to wait.

    - If the request comes from a processor in a HW transaction, there are two cases, depending on the request timestamp and the reservation timestamp:

        a) If the reservation timestamp is older than the request, the processor is NACKed by the directory.

        b) If the request timestamp is older than the reservation one, the request is granted.

This policy solves the deadlock case presented in Section 3. If *D* was older than *A* or *B*, then *D* would not receive a NACK, it would eventually finish and let *A* and *B* continue. Otherwise, *D* would be NACKed by the directory with an older timestamp than its current transaction timestamp. Given that *D* has already NACKed *A* and *B*, its *possible_cycle* bit will be set, and *D* should abort, thus allowing *A* and *B* to continue. This mechanism also covers the case of *A* or *B* sharing the line and wanting to modify it inside the transaction. There is a possibility of some reader "ignoring" the reservation and accessing the line (case b above), but that is needed to prevent a deadlock case as presented in Section 3.

5. RELATED WORK

The LogTM HTM model was presented in [12]. The problem of writer starvation in the LogTM model was presented in [1] labelled STARVINGWRITER. In that work the authors propose an improvement by which a starved transaction trying to write a cache line shared by many others aborts the remote transactions. Our approach is different in that we don't need to abort remote transactions, and we explicitly cover the case of both HW and SW transactions using a hybrid system.

The first approach of a Hybrid Transactional Memory system that can use any generic HTM as the HW acceleration substrate was presented in [2]. This work used the LogTM model as the base HTM, same as our base model. The extension in [9] proposes different transaction execution phases for hybrid systems, such as HARDWARE, HYBRID or SOFTWARE. While we don't consider such phase division, the proposal of this paper would be applicable to the HARDWARE and HYBRID phases of execution, affecting the *orecs* instead of the locks, and the *modeIndicator* variable which is checked by all transactions and used to change the execution mode. Even more, the paper also introduces the idea of using scalable non-zero indicators instead of counters in the *modeIndicator* variable; these would also suffer from strong starvation in the general case. As a note, the authors indicate in the paper that they modified the contention manager in LogTM, changing the stalling mechanism addressed in this work. NZTM [13] is another hybrid proposal (also with a modified contention manager) that achieves zero-indirection STM, eliminating some of the performance overheads of our base STM.

Contention management for TM has been previously considered in many different works such as [2] or [13]. However, as far as we know no other work has addressed cache-line contention management for directory-based hybrid systems.

Many different HW mechanisms have been proposed to improve the performance of shared-memory synchronization and exclusion. Software reader-writer queue-based locks [10], as the ones used in our base hybrid TM, reduce contention by using a queue of waiters, at the cost of increased memory usage. QOLBY [6] was the first proposal to improve shared-memory synchronization, using hardware distributed queues. Memory-side atomic operations, first used in the NYU Ultracomputer [7], perform atomic operations in the memory controller rather than the processors' caches to prevent cache lines bouncing between processors. Recently proposed Active Memory Operations [5] extend the performance to streams of data. Active Messages [3] is a software proposal to move computation to the owner node, considering that the programmer knows where it resides. As we comment in next section, we are considering the extension of Directory Reservations to support fair access for explicit synchronization. Although possible, we don't know of any of these works specifically addressing writer starvation in shared-memory synchronization. Even more, the complexity of our directory changes is much lower than any of the previous mechanisms.

6. CONCLUSIONS AND FUTURE WORK

Directory Reservations properly used provide fair contention management between SW and HW transactions in a hybrid LogTM-based schema. We have analysed the memory cost of the system, which is low (one bit per directory line and L1, and the additional RT in the directory controller) and does not affect directory access latency. The idea can implement with a very low cost a limited fair queuing handled by the directory hardware. Finally, we have proposed a LogTM-specific version in which there is no deadlock, considering the specific mechanisms used in the original LogTM proposal.

We are also looking at applying the Directory Reservations idea to traditional locking. Considering a system with a Reservation Table implemented, our idea is to allow the programmer to explicitly reserve and release some lines, instead of leaving that task to the coherence protocol when detecting conflicts with HTM transactions. By carefully reserving selected lines in a locking implementation, we believe that it is possible to provide fair access with reduced contention in read-write locks. However, this has to be handled carefully to ensure, for example, that reservations don't prevent current holders of the lock from releasing it, generating a deadlock. Our initial sketches make us consider that it might be possible to obtain a performance similar to using memory-side atomic operations, exploiting the RT.

We are currently implementing the design in the GEMS simulator [11], using the Hybrid Lock-based system presented in [15]. This hybrid system presents writer starvation in the lock access, which is handled by the Directory Reservations idea. However, we still don't have results to present in this workshop paper.

7. ACKNOWLEDGMENTS

This work is supported by the cooperation agreement between the Barcelona Supercomputing Center National Supercomputer Facility and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contracts TIN2004-07440-C02-01, TIN2007-60625 and TIN2007-6802-C02-01, and by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC). The authors would like to thank the valuable comments received from the anonymous reviewers.

8. REFERENCES

[1] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift and D. A. Wood, *Performance Pathologies in Hardware Transactional Memory*. International Symposium on Computer Architecture (ISCA), June 2007.

[2] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir and Daniel Nussbaum. *Hybrid transactional memory*. 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, Oct. 2006.

[3] T. von Eicken, D. Culler, S. Goldstein and K. Schauser. *Active Messages: A mechanism for integrated communication and computation*. In Proc. of the 19th ISCA, May 1992.

[4] F. Ellen, Y. Lev, V. Luchangco and M. Moir, *SNZI: Scalable Non-Zero Indicators*. In Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing. Portland, Oregon, USA, August 12 - 15, 2007.

[5] Z. Fang, L. Zhang, J. B. Carter, A. Ibrahim and M. A. Parker. *Active memory operations*. In Proceedings of the 21st Annual international Conference on Supercomputing. Seattle, Washington, June 17 - 21, 2007.

[6] J. R. Goodman, M. K. Vernon and P. J. Woest. *Efficient syncrhonization primitives for large-scale cache-coherent multiprocessors*. In Proceedings of the 3rd international conference on Architectural support for programming languages and operating systems (ASPLOS'89), Boston, MA, USA, 1989.

[7] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph and M. Snir. *The NYU multicomputer – designing a MIMD shared-memory parallel machine*. IEEE TOPLAS, 5(2):164–189, Apr. 1983.

[8] B. He, W. N. Scherer III and M. L. Scott. *Preemption Adaptivity in Time-Published Queue-Based Spin Locks*. 11th Intl. Conf. on High Performance Computing, Dec. 2005

[9] Y. Lev, M. Moir and D. Nussbaum. *PhTM: Phased Transactional Memory*. Presented at The 2nd ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 07), Portland, Oregon, August 16, 2007.

[10] J. M. Mellor-Crummey and M. L. Scott. *Scalable reader-writer synchronization for shared-memory multiprocessors.* Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Williamsburg, Virginia, 1991.

[11] M. M.K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood *Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset*. Computer Architecture News, Sept. 2005.

[12] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill and D. A. Wood, *LogTM: Log-based Transactional Memory*. International Symposium on High Performance Computer Architecture (HPCA), February 2006.

[13] W. Scherer and M. Scott. *Advanced contention management for dynamic software transactional memory*. In Proc. 24th Annual ACM Symposium on Principles of Distributed Computing, 2005.

[14] F. Tabba, C. Wang and J. R. Goodman. *NZTM: Nonblocking Zero-Indirection Transactional Memory*. Presented at The 2nd ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 07), Portland, Oregon, August 16, 2007.

[15] E. Vallejo, T. Harris, A. Cristal, O. Unsal and M. Valero. *Hybrid Transactional Memory to accelerate safe lock-based transactions*. 3rd ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 2008). Salt Lake City, Utah, USA, February 23, 2008.

[16] 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*. International Symposium on High Performance Computer Architecture (HPCA), February 2007.