Towards Transactional Memory for Real-Time Systems

Schoberl, Martin; Thomsen, Bent; Thomsen, Lone Leth

Towards Transactional Memory for Real-Time Systems

Authors: Martin Schoeberl
Lone Leth Thomsen
Bent Thomsen

Technical Report 09-001
Department of Computer Science
Aalborg University

Created February 25th, 2009

Towards Transactional Memory for Real-Time Systems

Martin Schoeberl
Institute of Computer Engineering
Vienna University of Technology, Austria
mschoebe@mail.tuwien.ac.at

Bent Thomsen, Lone Leth Thomsen
Department of Computer Science
Aalborg University DK-9220 Aalborg
bt, lone@cs.aau.dk

TU Wien Technical Report 19/2009
AAU CS Technical Report 09-001

Abstract

In this paper, we explore a new synchronization paradigm for real-time systems: transactional memory for real-time systems. Transactional memory is considered as a solution for parallel programs on a shared memory chip multiprocessor. It simplifies the programming model and increases the average case throughput. However, in real-time systems we are interested in the worst-case execution time. In this paper we show that for a periodic thread model the maximum number of transaction aborts can be bounded and the system is time predictable. Furthermore, we propose a possible hardware implementation in the context of a Java processor and show first results in a multiprocessor simulation.

1 Introduction

Transactional memory (TM) is proposed in two flavors: as implementation in hardware (HTM) [10] and as implementation in software (STM) [22]. The granularity of conflict detection can be word based (STM, HTM), cache line based (HTM), or object based (STM).

For our proposed transactional memory we do not aim at a high average case throughput, but time predictability and low worst case execution time (WCET). We want to provide a simpler programming model and analyzable timing properties. Therefore, all design and architecture decisions are driven by minimizing the WCET.

Two well known issues are present in chip-multiprocessor (CMP) systems:

1. Synchronization and cache coherence/consitency is expensive
2. Multithreaded programming needed to use CMP, but the programming model is too complex

With TM we can relax the memory coherence and consistency model [7]. That results in simpler and more efficient hardware for shared memory multiprocessing. The use of generic atomic primitives relieves the programmer from the headaches to get get the synchronization correct and provide the maximum possible concurrency. For real-time systems (RTS) we shift the problem from the programmer to analysis tools to provide safe and tight WCET estimates.

2 Related Work

In [12] Knight proposes hardware support for transactions for mostly functional languages. The key elements are two fully associative caches: the *depends cache* implements the dependency list (besides acting as normal data read cache) and the *confirm cache* that acts as local cache for uncommitted writes (side effects in his terminology).

The term *transactional memory* was coined by Herlihy and Moss [10]. They realized that only a minor modification of the available cache coherence protocol is needed to implement transactional memory.

As computer architects were not convinced by the transaction idea no hardware implementation exists up to date in commercial microprocessors. To solve this chicken-egg problem researchers started to investigate solutions in software. In [22] Shavit and Touitou present *software transactional memory* (STM). The proposed STM provides static transactions. That means the data set has to be known in advance.

Saha et. al. [18] propose an ISA extension to provide architectural support for STM. The idea is based on additional mark bits for parts of a cache line (e.g., for 16 byte blocks of a 64 byte cache line).

Language support for transactions [8] in Java reinvestigates Hoare's conditional critical regions. To distinguish between normal field access and field access under a transaction a second method table holds references to a transactional version of methods. In [11] a practical implementation as a library (DSTM2) for standard Java is presented that requires only compare-and-swap (CAS) instructions.

In [7] a Transactional memory Coherence and Consistency (TCC) model is proposed. TCC combines the simpler hardware for message passing and the simpler shared memory programming model. The standard cache coherence protocol with the latency issue on each load and store instruction is substituted by the TCC hardware. The TCC hardware broadcasts all writes from each transaction in a single packet. Automatic rollback resolves any correctness violation. TCC differs from other approaches as *all* instructions are part of a transaction. The code is just split into transactions which can be done manually or automatically by the hardware. In [6] language extensions for loop and fork based parallelization for TCC are presented. The paper also contains detailed simulation results of speedup and write set size. The speedup is reported in the range of 4.5 to 7.8 for a 8 processor CMP configuration. For most applications a write buffer of 1 KB is sufficient. We assume that applications in the real-time domain will need even less on-chip memory. Also a finer granularity of the write buffer (single words instead of 64 Byte cache lines) will reduce the needed size.

A first prototype of TCC in FPGAs [25] implements a 8 processor configuration with PowerPC cores and a custom data cache for the transaction buffer. The system consists of 4 FPGAs for the TCC (2 PowerPCs per FPGA) and one control FPGA that runs Linux. Although the PowerPC can be clocked up the 300 MHz, the TCC system is clocked with 100 MHz. The paper also reports issues with the implementation of the custom data cache in current FPGA technology. The 4-way set associative TCC cache has an access time of 13 clock cycles and clearing cache state bits at the end of a transaction takes 257 clock cycles. An on-chip block RAM is used for the register checkpoint.

Our proposed TM shares many ideas with TCC. We also perform late conflict detection at commit and grab the commit token early on a buffer overflow. However, the design of the transaction buffer in our approach is different: TCC uses standard cache organization for the transaction buffer while we optimize our design for time predictability and not for average case throughput. Furthermore, TCC uses transactions for *all* memory operations. The resulting high variability of memory access times is hard to include in the WCET analysis[1]. We use the TM only for short atomic code sections and perform non transactional loads and stores via a time-predictable memory arbiter [16]. The TCC design consumes about 8000 LCs per processor core (not including the processor). We expect that our implementation will consume about 2000 LCs.

Unbounded transactional memory (UTM) [2] uses register renaming to support the register snapshot. The transactions state (address of the abort handler, transaction counter – for nested transactions, and a save bit in register file) is visible to the OS to be saved on a context switch. Eager versioning, log in virtual (global) memory. There is still a hardware limit in the proposal: the number of physical registers determine the maximum concurrent number of transactions in one CPU. A simplified version (LTM) was simulated. LTM is different: lazy versioning, cache + hash table for overflow, does not survive a context switch. The idea is just a simple logging of versions. [3] is the Journal version of the paper.

Log-based TM (LogTM) [15] performs eager (early) versioning (write to memory) + eager conflict detection. The argument for this decision is that commits are the common case and should be fast.

To the best of our knowledge, preemptible atomic regions (PAR) [14] is the only proposal of TM for real-time systems.

2.1 Database Management Systems

Transactions have been studied widely in the context of database management systems (DBMSs) where they provide a powerful mechanism to manage concurrent access to a database.

Most DBMSs support the ACID properties: atomicity - a transaction executes completely or not at all; consistency - transactions are a correct transformation of

[1]The bounds become impractical high: for n processors the WCET of a single load or store is $n-1$ times the longest commit time.

state; isolation - even though transactions execute concurrently, for each transaction all other transactions appear to either happen before or after, but not both; durability - modifications performed by completed transactions survive failure.

Most DBMSs are designed to maximize transaction throughput rather than to meet timing constraints of individual transactions. However, there is also a well developed literature on real-time DBMS, but most literature on RT-DBMS is really on fast-time DBMS or minimizing the number of transactions overrunning their deadlines [1, 17, 21].

A lot of DBMS research has gone into protocols for conflict detection and resolution. Lock-based protocols are usually termed pessimistic concurrency protocols. Lock-based protocols, with two-phase locking, are (probably) in widest use in RT-DBMS. When a DBMS does conflict detection at the end of a transaction (during commit) it is called optimistic concurrency [13]. Optimistic protocols, where transactions are run to completion and then checked for conflicts, are considered wasteful.

There are studies of main memory real-time DBMS, which are used in telecommunication applications, e.g. [4] where an object-oriented design is presented.

2.2 Time-predictable CMP

For the schedulability analysis of (hard) real-time systems the worst-case execution time (WCET) of all tasks and critical sections needs to be known. WCET analysis of complex architectures is far from trivial. Architectural enhancements that dynamically extract instruction level parallelism are practically not analyzable. A multi-core chip consisting of simpler pipelines is a possible solution for high-performance, time-predictable systems [20].

For our HTM we assume a CMP system with a time-division multiple access (TDMA) scheduled memory access. The TDMA arbitration policy isolates the cores of the CMP in the temporal domain and is therefore time predictable. The WCET of memory accessing instructions can be calculated when the TDMA schedule is known [16].

3 Hardware Transactional Memory for RTS

We propose a hardware implementation of the HTM. Each core is equipped with a small, fully associative buffer to cache the changed data during the transaction. All writes go only into the buffer and all reads are either cached in the same buffer or marked in a read set (a simplification that uses only tag memories). On a commit the buffer is written to the shared memory. During the write burst on commit all other cores listen to the write addresses and compare those with their own read set. If one of the write addresses matches a read address the transaction is marked to be aborted. The atomicity of the commit itself is enforced by a single global lock – the commit token.

The commit token can also be used on a buffer overflow. When a transaction overflows the write buffer or the memory for the read set is grabs the commit token and continues with the transaction. The atomicity is now enforced by the commit token. Grabbing the commit token before commit is intended as a backup solution on buffer overflow. It effectively serializes the atomic sections. The same mechanism can also be used to protect I/O operations that usually cannot be rolled back. On an I/O operation within a transaction the core also grabs the commit token.

Conflict detection can be performed early, when the first conflict really happens, or late on commit. Early detection is very expensive in hardware as all buffer local write traffic has to be observed by all other cores. That means $n-1$ devices have to listen to the other $n-1$ device. Furthermore, there is not much benefit in the WCET analysis from early conflict detection. Therefore we propose to use late conflict detection during the commit. When one transaction commits its write buffer to the shared memory all other transaction units just need to listen to this write burst, i.e. $n-1$ listeners to one writer.

When a conflict is detected the corresponding thread can be notified to abort the transaction early or late. Early notification can be represented by a thrown exception. Late notification just marks the transaction for an abort and the abort can be communicated at the end of the transaction. Again from a real-time perspective the worst-case behavior is the same and the implementation of the late notification is simpler in hardware. It also gives a cleaner software interface.

Two concurrent transactions conflict when the read set of one transaction intersects with the write set of another transaction. We perform the conflict detection during the atomic commit.

We have two options for the transaction abort: (1) just mark the abort and perform the abort and retry on the end of the transaction instead of commit; or (2) throw an interrupt to the aborted transaction for an earlier restart. From the worst-case perspective their behaviors are equivalent. Option (1) is slightly simpler to implement.

On a commit the commit token is arbitrated, the write set is written to main memory (we could keep it in the cache), and the read set addresses are broadcasted. All other processors compare the write set of the committing processor with their read set and the read set of the committing processor with their write set. Any overlap triggers a transaction abort.

Furthermore, we can handle I/O within a transaction: just grab the commit token before the I/O operation.

Our proposed TM system has the following properties:

1. Local buffering of stores
2. Atomic commit of the stores to the memory
3. Conflict detection between transactions
4. Rollback of the transaction on a conflict
5. Global lock for commit; also used for HW buffer overflow
6. Abort on context switch

3.1 Language Integration

The question remains if we support atomic blocks or only atomic methods. When we use code blocks we have to save all local variables that are live at the transaction start to rollback the transaction (or we would need to buffer local variable writes, which is not really an option). When using atomic methods local variables are empty. With methods we do not need a change in the Java language, we can go with an annotation as in [14].

Transactional Memory (TM) shares many commonalities with Database transactions, but there are differences as well. Durability is not importantin TM - or has to be interpreted differently - if the system crashes it is a disaster and memory is lost. In DBMS we only have transactional access, in TM we may have both transactional and non-transactional access, i.e. a variable can be read and written inside a transaction and outside a transaction - what is the meaning of that? For further discussions see [5].

In DBMS transactions can be aborted from within the transaction. Similar concepts have been studied in Haskell [9]. Do we have similar concepts, e.g. atomic, retry and orelse, in Java TM? This is an open question.

3.2 Programming Style

Atomic regions simplify concurrent programming as the hardware automatically resolves synchronization conflicts. However, for good performance and tight WCET bounds the programmer should follow a few rules:

Minimize False Positives: Without any knowledge and analysis each atomic region can conflict with each other region and has to be considered by the real-time analysis. A data flow and type analysis can detect non conflicting regions and reduce the worst-case number of aborts. However, a programmer can still help the analysis. As an example consider using lists to communicate between producer and consumer threads. If independent producer/consumer pairs use the same type of lists it is hard to analyze their independency. With different subtypes for different lists the analysis is trivial. Although this practice is contrary to common OO design wisdom it can help when sophisticated analysis tools are not available.

Avoid Buffer Overflow: The on-chip transaction buffer is of limited size. An overflow is correctly handled by the hardware, but stalls all other processors in the system. Keeping the read and write set of transactions within the buffer limits avoids the expensive atomic execution on an overflow.

Short Transactions: The execution time of a transaction is multiplied by $2n-1$ for the WCET analysis. For large n the execution time of the transaction can dominate the overall WCET of a thread. Keeping transactions short minimizes this cost.

4 HTM Analysis

To use transactions in a real-time system we need to find a bound on the maximum number of possible transaction aborts and the resulting retries. To find this bound we have to analyze all threads in the system. The bound per thread itself is then integrated into the WCET analysis of the individual threads. With t_{WCET} as the original WCET, t_{atomic} the maximum execution time of the atomic region, and k the number of times the atomic region is executed (1 + number of retries) the overall WCET t'_{WCET} is

$$t'_{WCET} = t_{WCET} + t_{atomic}(k-1) \tag{1}$$

Theorem 1. *For n periodic threads that contain a single atomic region the maximum number k a single thread has to execute that region per period is*

$$k = 2n - 1 \tag{2}$$

Proof. We assume the critical instant where n threads start their period at the same time and have their atomic region at the beginning. One thread will commit and $n-1$ threads will have to perform a retry. We again assume a critical instant where now $n-1$ threads execute their atomic region and $n-2$ threads have to execute their atomic region a 3rd time. The last thread that will commit was aborted $n-1$ times and had to execute the atomic region n times. In that case $k = n$.

However, if we construct a phasing where a commit of period i is back to back with a commit of period $i+1$ for each thread those two commits can abort the other thread's atomic regions two times. Therefore, the last thread has been aborted its atomic region $2(n-1)$ times and needs to execute the atomic section $k = 2n-1$ times. □

We can analyze situations where a thread j's period is shorter than the conflict resolving time. However, this is only useful when other threads are involved. Otherwise thread j can starve. Considering this case results in a recursive formulae. However, for practical systems the assumption that the period of each thread involved in a possible conflict is longer then the maximum conflict time is a reasonable one.

4.1 Region Analysis

The above bound assumes one atomic region per period with the same t_{atomic} for all threads. This is not an uncommon assumption, but it is rather restrictive.

If we assume that the execution time for atomic regions may vary for each periodic thread, we can still calculate the maximal number of retries for each atomic region in each periodic thread. We still assume that there is only one atomic region in each period.

Theorem 2. *For n periodic threads that contain a single atomic region the maximum number k_j the j'th thread has to execute its region per period is bounded by the following:*

$$k_j = \sum_{i=1}^{j-1} \left\lceil \frac{P_j}{P_i} \right\rceil + \sum_{i=j+1}^{n} \left\lceil \frac{P_j}{P_i} \right\rceil \tag{3}$$

Proof. The number of possible conflicts a thread j kan have with another thread i depends on the length of period P_j and P_i. If P_i is equal to or longer than P_j then there is at most one conflict between the two threads which implies that one of them will have completed its atomic region sucessfully and $\left\lceil \frac{P_j}{P_i} \right\rceil = 1$. If P_i is shorter than P_j then $\left\lceil \frac{P_j}{P_i} \right\rceil$ is equal to the number of times thread i can execute during the period of thread j. Each such execution of thread i could have its atomic region conflicting with the atomic region in thread j. Finally we need to sum up over all threads in the system. □

The above sum is dominated by the following number which may be easier to calculate:

Theorem 3.

$$k_j \leq (n-1) \left\lceil \frac{P_{max}}{P_{min}} \right\rceil \tag{4}$$

4.2 Analysis to reduce possible conflicts

The formulas introduced in the previous section calculate the maximal numbers of retries under the rather conservative assumption that all atomic regions could be in conflict with each other. This is clearly a safe assumption, but will often lead to very pessimistic runtime assumptions.

If it is possible to determine that a thread i does not share variables or references to objects with another thread j, it is safe to eliminate $\left\lceil \frac{P_j}{P_i} \right\rceil$ from the count of possible retries the atomic region in thread j may have to go through.

We envision that Points to Analysis for Java, can be adapted to calculate an abstract set of memory locations each thread may refer to during their execution of their atomic region and thus calculating the intersection of

```
// The producer task
while (cnt < Const.CNT) {
    RTTM.start();
        if (!queue.full()) {
                ++cnt;
                queue.enq((T) obj);
        }
    RTTM.end();
}

// The consumer task
while (cnt<Const.CNT) {
    RTTM.start();
        Object obj = queue.deq();
        if (obj!=null) {
                ++cnt;
        }
    RTTM.end();
}

// The mover task
while (cnt<Const.CNT) {
    RTTM.start();
        if (!in.full()) {
                Object obj = out.deq();
                if (obj!=null) {
                        in.enq((T) obj);
                        ++cnt;
                }
        }
    RTTM.end();
}
```

Figure 1: The producer, consumer, and mover tasks

such sets for each pair of threads would be straightforward. The Soot framework already provides three implementations of points-to analysis: CHA, SPARK and Paddle [24]. [23] presents a scalable and precise context-sensitive points-to analysis for Java.

Another way of looking at such an analysis, is that if it yields a high level of possible sharings between threads in the system, it is possibly a sign of programming error or at least bad design.

5 Evaluation

For a first evaluation of HTM we have implemented HTM within a simulation of the Java processor JOP [19]. The simulation was extended to simulate a chip-multiprocessor version of JOP. Within this simulation we are able to gather some statistics on the HTM behavior that will guide the hardware implementation. We varied the size of the transaction buffer and the read set cache. As applications we used a few micro benchmarks.

			Address set	
Thread	Trans.	Retries	Write	Read
Producer	1000	0	654	679
Consumer	1001	1000	4	19

Table 1: Single vector

			Address set	
Thread	Trans.	Retries	Write	Read
Producer	1000	0	654	679
Mover	1001	501	7	30
Consumer	3005	1000	4	19

Table 2: Two vectors

TM for RTS is evaluated with a few micro-benchmarks implementing different configurations of the producer/consumer pattern. We use two different buffers for the data exchange: (1) the standard Java Vector, and (2) a bounded queue. Three types of tasks exchange information: the task Producer, the task Consumer, and the task Mover. All tasks run in a tight loop and perform their operations 1000 times. Figure 1 shows the code for the three tasks for the queue version. The Producer inserts 1000 objects into the buffer. The same object is reused to provoke maximum transaction collisions in the examples. The Consumer removes elements from the buffer.

The Mover task is the classic example that does not compose with traditional locks. An element shall be removed from queue *A* and inserted into another queue *B* with the invariant that the element has to be either in *A* or *B*. When the queues use internal locks for the synchronization, the transfer needs to be protected by an additional lock. However, other threads that operate on the queues are usually not aware of the additional transfer lock. With atomic sections this operation composes naturally.

The tables 1–6 show the transaction statistics for each worker thread for the six examples. The tables show the number of transactions committed, retried after an abort, and the size of the write and read set.

With the first experiment, the Vector based communication with 2 threads (one producer and one consumer), shown in Table 1, we see large read and write sets. The consumer does not keep up with the producer and the

Thread	Trans.	Retries	Address set Write	Address set Read
Producer 1	1000	0	654	679
Consumer 1	1001	501	4	19
Producer 2	1000	0	654	679
Consumer 2	1002	501	4	19

Table 3: Two independent vectors

Thread	Trans.	Retries	Address set Write	Address set Read
Producer	1000	999	3	18
Consumer	5359	637	2	13

Table 4: Single queue

Vector is internally resized to buffer the request. The experiment shows that this kind of data structure is not ideal for real-time systems. The Vector based communication with 3 threads (one producer, one mover, and one consumer), shown in Table 2, shows the similar issue with the resizing of the internal array in the first queue. As the code of the Mover takes longer to execute than the code of the Consumer the Vector between these threads does not grow. The last Vector example shows two independent producer/consumer pairs. As the simulation runs all cores in lock-step, the results of both pairs, shown in Table 6, is almost identical.

We have run the same examples with bounded queues for the communication. The results of the simulation are shown in Tables 4–6. As the queues are bounded we see only small read and write sets. From the results in Table 5 we can derive a few observations on the three thread example: (1) The Mover task has the longest execution time and limits the throughput. The other two tasks execute their atomic sections more often finding the queue either full (Producer) or empty (Consumer). (2) As the test for full and empty does

Thread	Trans.	Retries	Address set Write	Address set Read
Producer	5317	208	3	18
Mover	1003	1006	4	28
Consumer	8420	269	2	13

Table 5: Two queues

Thread	Trans.	Retries	Address set Write	Address set Read
Producer 1	1000	999	3	18
Consumer 1	5360	636	2	13
Producer 2	1000	999	3	18
Consumer 2	5371	633	2	13

Table 6: Two independent queues

no change the state of the queue the retry count for the Producer and the Consumer is quite low. (3) The Mover task is aborted as often as it successfully commits.

In summary, we evaluated the HTM with examples that stress the transaction system to observe some real conflicts. All threads run in a tight loop executing an atomic section. Even under this load no thread starved. For real-world applications the atomic section is only a small part of the workload and conflicts are seldom. We have run some examples with periodic threads, but could not produce enough conflicts to provide interesting results.

6 Conclusion

In this paper, we explored a new synchronization paradigm for real-time systems: transactional memory. We showed various formulaes to bound the maximum number of retries for a transaction. Therefore, transactional memory can be considered in real-time systems.

Furthermore, we have implemented TM in a simulation of the Java processor JOP and evaluated the design with producer/comsumer tasks. It has been shown that bounded queues result in a small read and write set.

As future work we consider more simulation with realistic applications and an implementation of TM within an FPGA.

References

[1] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions. *SIGMOD Rec.*, 17(1):71–81, 1988.

[2] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings 11th International Conference on High-Performance Computer Architecture (HPCA 2005)*, pages 316–327,

San Francisco, CA, USA, Feb. 2005. IEEE Computer Society.

[3] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. *IEEE Micro*, 26(1):59–69, 2006.

[4] S. K. Cha, B. D. Park, S. J. Lee, S. H. Song, J. H. Park, J. S. Lee, S. Y. Park, D. Y. Hur, and G. B. Kim. Object-oriented design of main-memory dbms for real-time applications. In *RTCSA '95: Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications*, page 109, Washington, DC, USA, 1995. IEEE Computer Society.

[5] P. Felber, C. Fetzer, R. Guerraoui, and T. Harris. Transactions are back—but are they the same? *SIGACT News*, 39(1):48–58, 2008.

[6] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. K. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (TCC). In S. Mukherjee and K. S. McKinley, editors, *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2004)*, pages 1–13, Boston, MA, USA, October 2004. ACM.

[7] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 102, Washington, DC, USA, 2004. IEEE Computer Society.

[8] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the OOPSLA '03 conference*, pages 388–402, New York, NY, USA, 2003. ACM Press.

[9] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[10] M. Herlihy, J. Eliot, and B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Computer Architecture, 1993. Proceedings of the 20th Annual International Symposium on*, pages 289–300, 1993.

[11] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *Proceedings of the 2006 OOPSLA Conference*, pages 253–262, New York, NY, USA, 2006. ACM Press.

[12] T. Knight. An architecture for mostly functional languages. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 105–112, New York, NY, USA, 1986. ACM Press.

[13] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[14] J. Manson, J. Baker, A. Cunei, S. Jagannathan, M. Prochazka, B. Xin, and J. Vitek. Preemptible atomic regions for real-time java. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 62–71, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[15] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265. IEEE Computer Society, Feb. 2006.

[16] C. Pitter. Time-predictable memory arbitration for a Java chip-multiprocessor. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems (JTRES 2008)*, pages 115–122, New York, NY, USA, 2008. ACM.

[17] K. Ramamritham, S. H. Son, and L. C. Dipippo. Real-time databases and data services. *Real-Time Syst.*, 28(2-3):179–215, 2004.

[18] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[19] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[20] M. Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, in press, 2009.

[21] S. Semghouni, L. Amanton, B. Sadeg, and A. Berred. On new scheduling policy for the improvement of firm rtdbss performances. *Data Knowl. Eng.*, 63(2):414–432, 2007.

[22] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[23] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 387–400, New York, NY, USA, 2006. ACM.

[24] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, pages 18–34, London, UK, 2000. Springer-Verlag.

[25] S. Wee, J. Casper, N. Njoroge, Y. Tesylar, D. Ge, C. Kozyrakis, and K. Olukotun. A practical fpga-based framework for novel cmp research. In *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pages 116–125, New York, NY, USA, 2007. ACM.