RTTM: Real-Time Transactional Memory

Martin Schoeberl, Florian Brandner
Institute of Computer Engineering, Computer Languages
Vienna University of Technology
Vienna, Austria
mschoebe@mail.tuwien.ac.at, brandner@complang.tuwien.ac.at

Jan Vitek
Computer Science Department
Purdue University
West Lafayette, IN, USA
jv@cs.purdue.edu

***Abstract*—Hardware transactional memory is a promising synchronization technology for chip-multiprocessors. It simplifies programming of concurrent applications and allows for higher concurrency than lock based synchronization. Standard transactional memory is optimized for average case throughput, but for real-time systems we are interested in worst-case execution times. We propose real-time transactional memory (RTTM) as a time-predictable synchronization solution for chip-multiprocessors in real-time systems. We define the hardware for time-predictable transactions and provide a bound for the maximum transaction retries. The proposed RTTM is evaluated with a simulation of a Java chip-multiprocessor.**

I. Introduction

Computing is about to undergo, if not another revolution, then a vigorous shaking-up. Processor manufacturers have essentially given up trying to increase clock speeds. Moore's law has not been repealed: each year, more and more transistors fit into the same space, but clock speeds cannot be increased without overheating. Instead, processor manufacturers have focused on *multicore* architectures, in which multiple processor cores reside on a single chip. As a result of this sea change, parallel machines are becoming a de facto standard. Unfortunately, programming such systems safely and efficiently using classical concurrency abstractions is a severe burden. Yet with clock speeds stagnant, programmers must increasingly use parallelism to enable more ambitious applications. This adaptation will not be easy. Programmers typically rely on locks and condition variables for synchronization, which are implemented with special processor instructions, such as compare-and-swap (CAS). Multi-processor programming with locks is far from trivial. Simple coarse grain locks limit the possible parallelism; fine grain locking can introduce errors due to data races and deadlocks. Avoiding locks and implementing lock-free or wait-free data structures and algorithms is considered art and not engineering [11]. This conventional approach is known to be error-prone. Programmability remains the key hurdle towards effectively utilizing these next-generation computing systems. This is particularly true for *embedded systems* which are often safety critical and where failures can cause loss of life. The difficulties of programming multiprocessor systems, such as data races, and deadlocks, are magnified in the context of embedded real-time programs because there is still little support for multicore in terms of theory and tools. Interestingly, shared memory is also a scalability issue. Each core contains a local, first level cache. Second and further levels of caches are usually shared between the on-chip cores. The shared data in the local caches is kept coherent and consistent by cache coherence protocols between the cache controllers. However, keeping caches coherent and consistent limits the scaling of CMP systems.

Transactional Memory (TM) is a promising concurrency control solution for parallel and concurrent computing systems. Transactional memory has been the subject of significant interest in both academia and industry because it offers a compelling alternative to existing concurrency control abstractions, making it especially well-suited for programming applications on scalable parallel platforms. Because transactional memory implementations often support an optimistic concurrency model, they can be used to safely allow speculative access to data by a large number of processors without requiring global program analysis. TM abstractions permit logically concurrent access to shared regions of code, but ensure through some combination of hardware, compiler, and runtime support that such accesses do not violate intended serializability invariants. However, while there has been a plethora of work exploring different implementation techniques for transactional memory in both hardware and software, there is no TM system that is adequate for real-time applications running on multiprocessor systems. Although first introduced as an extension of the cache coherence protocol [8], TM can simplify caches with transactional coherence and consistency [6]. The programming model with TM is simpler than with locks; a mutual exclusive section is marked as an atomic region. The TM ensures, either in hardware or in software, that the marked section is executed atomically. The atomic sections are executed optimistically. On a conflict one of the transactions is aborted and the atomic section is automatically reexecuted.

This paper proposes *Real-Time Transaction Memory* (RTTM), a new concurrency control abstraction that is geared towards time-predictability and evaluates the proposal in the context of real-time Java programs running on a

simulation of a chip-multiprocessor (CMP). While, our work targets the Java programming language and must thus deal with some issues that are specific to that language, it is not limited to it. Choosing Java does present some significant advantages over low-level languages. Java is memory-safe, that is, the language prevents memory accesses through untyped pointers. This property enables the implementation of copying and compacting real-time garbage collection algorithms which simplify the task of writing concurrent algorithms as programmers need not worry about memory reclamation. The second benefit of Java is that compiler transformations and program analysis techniques can rely on declared properties of pointers and thus are easier to prove sound.

RTTM brings the benefits of transactional memories into the real-time systems world. It simplifies the programming model with atomic regions instead of correct selection of locks. The execution time pessimism is reduced by analysis tools instead of error prone implementation of lock-free algorithms. The contributions of the paper are following: (a) the design of a time-predictable hardware transactional memory; (b) analysis of the worst-case number of retries in a periodic thread model; (c) suggestions for analysis and a programming discipline to reduce the number of possible conflicting transactions; and (d), a first evaluation of RTTM on a simulation within a Java based CMP.

II. Background: Real-Time Java and TM

This section presents some background on the use of Java for real-time processing, presents existing APIs for transactional memory in Java and overviews related work.

A. Real-Time Java

Java is increasingly being used in mission-critical systems in fields such as avionics [23] and shipboard computing for steering and control. To address the requirements of time-critical applications, the Real-Time Specification for Java (RTSJ) [4] was developed. While the first release of the RTSJ appeared in 2000, it is only recently that production implementations have become available. One of the notable advantages of the RTSJ is that it is possible to implement mixed-mode systems in which real-time and non-real-time tasks can co-exist. The integration of the two programming models, while not seamless, represents a pragmatic engineering compromise. The real-time extensions are backward-compatible with the rest of the Java programming language and require no changes to the tool chain. Thus, adopting real-time Java does not require forsaking libraries or legacy code. Instead, it is possible to implement the real-time portion of an application using the real-time extensions, and to use standard Java for the rest.

Synchronization in Real-Time Java: Real-time threads have to be scheduled carefully. Each thread may have a length of time during which it must complete a given task before it yields; this time is called the thread's *deadline*. In the RTSJ, the default scheduler is *priority preemptive*. A priority preemptive scheduler *releases* a real-time thread according to its priority. A thread can be released if and only if it has higher priority than the currently executing thread. If multiple threads have the same priority, they are scheduled in FIFO order. We say that a set of threads is *schedulable* if all threads can execute within their periods without missing deadlines. In order to ensure schedulability, it is necessary to bound both the time required to execute the thread up to the end of the current period and the thread's *blocking time*. The requirement of schedulability is complicated by a number of blocking issues. It is necessary to estimate the longest time a thread may block. Thus, bounds need to be provided for the length of any given critical section. This is a standard assumption in real-time systems. If a high priority thread blocks waiting for a low priority thread to release a lock, and the low priority thread is preempted by a medium priority thread, then the medium priority thread may execute instead of the high priority thread. This situation is called *priority inversion*, and can result in unbounded blocking times, potentially causing the high-priority thread to miss deadlines. Priority inversion has a history as a particularly troublesome issue: in the Mars Pathfinder mission, for instance, a priority inversion problem caused frequent system resets. Real-time Java distinguishes hard real-time threads from (softer) real-time threads: the former are not allowed to read references to heap objects. This restriction is meant to ensure that a hard real-time thread will never block in order to wait for the garbage collector.

One motivation for our work is to simplify the task of reasoning about critical sections by providing a concurrency control abstraction that minimizes these problems and attempts to avoid undue blocking delays and catastrophic interference between the real-time and the non-real-time parts of a RTSJ environment.

B. Example

This section introduces transactions and contrasts them with lock-based concurrency control mechanisms. Figure 1 is a simplified extract from a queue-based thread pool implementation. The method leaderExec() in the class ThreadPoolLane places an incoming Request onto the queue requestBuffer (a.4). If a processor is free, it will dequeue (and execute) the Request when it is next scheduled. The code is taken from the Zen real-time ORB [13].

This example makes extensive use of synchronization. The method leaderExec() is synchronized (a.1) to ensure that multiple threads cannot concurrently access the method of the ThreadPoolLane on which it will be invoked. The second use of locks is around lines a.4 and a.5; it ensures that the length of the queue is consistent with numBuffered. This cannot be accomplished with the lock on the ThreadPool-

```
    class ThreadPoolLane {
1.      synchronized leaderExec(Request task){
2.          if (borrowThreadAndExec(task))
3.              synchronized (requestBuffer) {
4.                  requestBuffer.enqueue(task);
5.                  numBuffered++;

    class Queue {
7.      final Object sObject = new Object();
8.      void enqueue(Object data) {
9.          QueueNode node=getNode();
10.         node.value=data;
11.         synchronized (sObject) {
12.             // enqueue the object
```

(a) With Monitors.

```
    class ThreadPoolLane {
1.      @atomic leaderExec(Request task){
2.          if (borrowThreadAndExec(task))
3.              requestBuffer.enqueue(task);
4.              numBuffered++;

    class Queue {
5.      @atomic void enqueue(Object data) {
6.          QueueNode node=getNode();
7.          node.value=data;
8.          // enqueue the object
```

(b) With Transactional Memory.

Figure 1. Example from the Zen ORB.

Lane because there may be other methods (not pictured) that are not synchronized on the ThreadPoolLane object, but that access the requestBuffer queue and numBuffered. The final use of locking in this example occurs inside of the implementation of the Queue class: the enqueue() method relies on a private object (a.7) to protect the updates to the queue (a.12).

We contrast this with an implementation that uses transactional memory. Transactions are denoted with an @atomic boundary as in [15]. A bytecode rewriting procedure transforms annotated methods to add explicit calls to the lower-level RTTM API. As mentioned above, an @atomic method executes atomically. While it is executing, it records the original contents of locations to which it writes; these values are then restored if a conflict is detected the transaction ends as if the code had not executed at all. Aborted atomic regions are silently reexecuted until they successfully commit. The programming model is intentionally simple; in most cases, monitors can be exchanged for atomic regions with minimal changes to the program. In Figure 1.b, we use two atomic sections: one for the leaderExec() method (b.1) and another for the enqueue() method (b.5). The first atomic is sufficient to prevent all data races within leaderExec(); it is therefore unnecessary to obtain a lock on the queue. If enqueue() were only called from leaderExec(), it would not need to be declared atomic; however, as mentioned above, it is declared atomic to allow use in a non-atomic calling context. The solution that uses atomic regions is simpler and easier to prove correct, as it does not rely on multiple locking granularities. A single atomic will protect all objects accessed within the dynamic extent of the annotated method. Contrast this with the lock-based solution, where all potentially exposed objects must be locked. Furthermore, the order of lock acquisition is critical to prevent deadlocks. On the other hand, atomics cannot deadlock: they do not block waiting for each other to finish.

C. Transactional Memory

The term *transactional memory* was coined by Herlihy and Moss [8]. They realized that only a minor modification of the available cache coherence protocol is needed to implement transactional memory. Knight proposed hardware support for transactions for mostly functional languages [12]. 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.

As computer architects where 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. Shavit and Touitou present *software transactional memory* (STM) [25]. The proposed STM provides static transactions. That means the data set has to be known in advance. A number of later papers investigated the concept of software transactional memory [10], [24], and provided implementations with support for undoing operations. Herlihy et al. describe a software transactional memory abstraction [9] for Java that allows transactional objects to be created dynamically. Harris and Fraser [7] described a lightweight transactional model for Java. Their model is more general than ours, but incurs overheads that are much higher, and does not provide real-time guarantees. Anderson et al. [3] described a language independent notion of lock free objects in real-time systems. In contrast, our work leverages its integration with the language and compiler to achieve greater simplicity and efficiency. Saha et. al. propose an ISA extension to provide architectural support for STM [20]. 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).

The Transactional memory Coherence and Consistency (TCC) model is proposed in [6]. 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. Language extensions for loop and fork based parallelization for TCC are presented in [5]. The paper also contains detailed simulation results of speedup and write set sizes. 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.

Preemptible atomic regions (PAR) [15], was the first proposal of TM for real-time systems. A PAR is aborted when a higher priority task becomes ready and preempts the lower priority task – independent whether the high priority task executes an atomic region or not. The effects of a PAR are undone at the interrupt. The concept of PAR is only valid on uniprocessor systems. A first concept of TM for real-time CMP systems is presented in [**?**].

D. 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 [22].

For RTTM 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 [19].

III. Real-Time Transactional Memory

We propose a hardware implementation of the RTTM for a time-predictable CMP system. Only with hardware support of transactions the desired efficiency of atomic sections can be achieved. With TM we can solve two issues of CMP systems: (1) synchronization and cache coherence/consistency protocols are expensive; (2) exploring the power of CMP systems needs multiprogramming, but the programming model is complex. With TM we can relax the memory coherence and consistency model [6]. This results in simpler and more efficient hardware for shared memory multiprocessing. Furthermore, avoiding cache coherence protocols simplifies the timing model of the memory access for the WCET analysis. The use of generic atomic primitives relieves the programmer from the headaches to get the synchronization correct and provide the maximum possible concurrency. For real-time systems we shift the problem from the programmer to analysis tools to provide safe and tight WCET estimates.

The main design goals for the RTTM are: (a) simple programming model and (b) analyzable timing properties. Therefore, all design and architecture decisions are driven by their impact on the WCET. In contrast to other TM proposal RTTM does not aim for a high average case throughput, but for a time-predictable TM with a low WCET. RTTM is intended to support small atomic sections with a few read and write operations. Therefore, it is more an extension of the CAS instruction to simplify the implementation of non-blocking communication algorithms.

A. Transaction Buffering

Each core in the CMP is equipped with a small, fully associative buffer to cache the changed data during the transaction. All writes go only into the buffer. Reads addresses are marked in a read set – a simplification that uses only tag memories. Read data can also be cached, but caching is not essential for the correct operation of the RTTM.

The write buffer and tag memory for the read set are organized for single word access. This organization ensures that no false positive conflicts are detected. For the same reason the transaction buffer has to be a fully associative cache with a FIFO replacement strategy. Fully associative caches are expensive and therefore the size is limited. We assume that real-time systems programmers are aware of the high cost of synchronization and will use small atomic sections where a few words are read and written.

B. Transaction Commit

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 tag memory for the read set the commit token is grabbed and the transaction continues. 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.

C. Conflict Detection

Conflict detection can be performed early, when the first conflict really happens, or late on commit. From the analysis point of view both approaches lead to the same WCET. Early conflict detection is an average-case optimization.

Early conflict detection is very expensive in hardware as the buffer local write traffic has to be observed by all

other cores. That means $n-1$ cores have to listen to the other $n-1$ cores. This is the same effort that is needed for a cache coherence protocol. Therefore we propose to use late conflict detection during the commit phase. When one transaction commits its write buffer to the shared memory all other transaction units just need to listen to this write burst – leading to maximum of $n-1$ listeners to a single 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 gives also a cleaner software interface.

Transactions that are marked as aborted, but continue to run their transaction, are called zombie transactions. Zombie transactions can see a mix of old and committed data. Therefore, the invariant of the atomic section is not preserved. As those transactions will be aborted at the end of the transaction they do not change the global state. However, zombie transaction can throw unexpected exceptions (e.g., divide by zero) or run infinite loops. Thrown exceptions for a transaction that is marked aborted can be safely ignored. To avoid infinite loops the abort status has to be checked on branches. A branch can simply be redirected to the abort handler.

IV. RTTM Analysis

In order to make use of TM in real-time systems we need to show that it is possible to calculate a bound for the execution time of every thread in the given system. We will show that the number of retries for any given transaction is bounded and the WCET of a task can be calculated.

Definition 1. *For the analysis we assume a real-time system consisting of n threads $\tau_1 \dots \tau_n$ that each contain a single atomic region, which is executed only once per period. Each thread has a period T_i and a WCET (in scheduling theory often called cost) t_{c_i} that includes the execution time t_{a_i} of the atomic section.*

The preliminary WCET bounds the thread's execution time per period without TM conflicts, i.e., it does not account for aborts and retries. However, the successful execution of the atomic region is included. When the maximum number of retries r is known, the final WCET t_{wcet} is

$$t_{wcet} = t_c + r t_a \tag{1}$$

Conflicts occurring at runtime are resolved by aborting and restarting all but one of the involved transactions. Figure 2 illustrates this resolution process for mutually conflicting

Figure 2. Conflict resolution in RTTM.

threads. In the worst-case phasing all threads start their respective transaction at the same time and simultaneously try to commit the transaction's state to the shared memory. Only one of the threads is allowed to commit its state, for all others the transactions are aborted and restarted. In the next *round*, the same situation repeats with the exception that the previously winning thread does not participate in the race. The number of competing threads thus decreases with each round until all threads where able to commit.

Definition 2. *The worst-case time for resolving any TM conflict in a real-time system such that every thread was able to commit its local transaction state is referred to as t_r.*

This resolution time t_r is influenced by transactions of other threads that conflict with the threads own transaction, i.e., the completion of another transaction may cause the threads running transaction to be aborted when the write set of the former transaction overlaps with the read set of the latter. To bound the conflict resolve time, two transactions of thread τ_i must be separated at least by the resolution time t_r. Therefore, one thread will influence the conflict resolution only with a single transaction.

Lemma 1. *Under the assumption that the deadline of a thread τ is not longer than its period T two transactions are separated by at least the resolve time t_r.*

Proof: To meet the deadline following criterion must be met:

$$t_c - t_a + t_r \leq T \tag{2}$$

With one transaction per period the worst case is a late start of the transaction in period k and an early start at period $k+1$. Due to (2) the latest start time t^k_{late}, relative to the period k start time t^k, is

$$t^k_{late} = t^k + T - t_r \tag{3}$$

The earliest start time t^{k+1}_{early} is at the begin of the period. The period start times t^k and t^{k+1} are separated by the period $T = t^{k+1} - t^k$. The minimum difference t_{min} of the start

times is

$$\begin{aligned} t_{min} &= t^{k+1}_{early} - t^{k}_{late} && (4)\\ &= t^{k+1} - (t^k + T - t_r) && (5)\\ &= t^{k+1} - t^k - T + t_r && (6)\\ &= T - T + t_r && (7)\\ &= t_r && (8) \end{aligned}$$

■

Theorem 1. *For n periodic threads that contain a single transaction the maximum number r a single thread has to reexecute that transaction is*

$$r = n - 1 \tag{9}$$

Proof: We assume the critical instant where n threads commit their atomic section at the same time. 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 reexecute the atomic region $n-1$ times. In that case $r = n-1$.

■

With the simplification of equal[1] transaction times $\forall i \in \{1, \ldots, n\} : t_{a_i} = t_a$ the resulting resolve time is

$$t_r = (r+1)t_a \tag{10}$$

The above analysis can also be used to calculate the retry bounds of compare-and-set (CAS) operations, which are available on current CMP systems. To include several transactions per period in the analysis, those transactions can be modeled as several threads with the same period and a single transaction. The resulting bounds will be conservative, but safe.

Considering individual transactions times t_{a_i} and the resulting individual resolve times t_{r_i} is considered future work. Tightening the bound for a thread that contains more than one transaction is considered future work.

A. Program Analysis

The use of transactional memory greatly simplifies the programming model and has the potential to reduce typical synchronization errors and deadlocks. However, it also demands for accurate program analysis that allows to (semi-) automatically infer the interdependencies between threads and transactions. The analysis is required to accurately derive the conflict sets and the size of the read and write sets for each thread. Tight conflict sets result in lower bounds on the maximum retry count.

The conflict sets can be computed efficiently using flow-insensitive, context-sensitive points-to analysis. For every instruction that is executed within an atomic section, the set of possible pointers is calculated and combined to summarize the possible memory locations referenced by the transaction. Two transactions are considered to conflict if the memory locations computed by the analysis overlap, such that one thread potentially updates a location that the other thread potentially reads from. The accuracy of the analysis directly influences the number of retries that need to be accounted for in the final WCET.

Context-sensitive points-to analysis has successfully been applied on large programs in the context of various transformations and optimizations [26], [14], [16] such as elimination of type-checks for casts or receiver-type analysis for virtual method invocations. Static detection of race conditions in multi-threaded programs [17], [18] is an important application that is highly related to the problem of determining conflict sets.

The internal buffers that hold the local state for each transaction are limited in size. This can lead to buffer overflows in the case when the set of referenced memory locations grows too large. RTTM is able to handle such overflows, at the expense of performance, by serializing transactions. In case of an overflow the global commit token is acquired and commit of all other transactions is blocked.

Identifying the size of the read and write set of a transaction cannot be done using points-to analysis. Instead, a symbolic analysis is required that accurately models all possible states of the internal buffers for every program point. We consider abstract interpretation to be the best approach to perform the desired analysis. Previous work on cache analysis using abstract interpretation showed promising results [2]. In fact, the internal organization of the transaction buffers is similar to fully associative caches. It is thus likely that results on cache analysis can be applied to identify transaction overflows.

We have used the WALA analysis library [1] for some preliminary tests to get insights into the analysis problems for RTTM. The results show that the required analysis is feasible for real-world applications. Details are given in the next section.

V. Evaluation

For a first evaluation of RTTM we have implemented RTTM within a simulation of the Java processor JOP [21]. The simulation is an interpreting JVM that can execute the linked binaries for JOP. It contains emulations of JOPs I/O devices, the memory system, and the caches. Furthermore, the simulation has the same restrictions as JOP as is primarily intended for debugging.

The simulation was extended to simulate a chip-multiprocessor version of JOP. The interpreter loop executes bytecodes of several JVMs and the switch between the JVMs is at bytecode level. Therefore, we can simulate the fine grain interaction of a real CMP system. The execution speed of

[1] Or using the maximum value of $t_a = max(t_{a_i})$.

```
// 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 3. The producer, consumer, and mover tasks

the simulation, when running on an actual PC, is similar to the execution time on the real hardware – a JOP clocked at 100 MHz. Within this simulation we are able to gather some statistics on the RTTM behavior that will guide the hardware implementation.

A. Examples

RTTM 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 3 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.

			Address set		
Thread	Trans.	Retries	Write	Read	R & W
Producer	1000	0	654	673	1316
Consumer	1001	1000	4	15	15

Table I
SINGLE VECTOR

			Address set		
Thread	Trans.	Retries	Write	Read	R & W
Producer	1000	0	654	673	1316
Mover	1001	501	7	23	23
Consumer	3502	1000	4	15	15

Table II
TWO VECTORS

			Address set		
Thread	Trans.	Retries	Write	Read	R & W
Producer 1	1000	0	654	673	1316
Consumer 1	1001	501	4	15	15
Producer 2	1000	0	654	673	1316
Consumer 2	1002	501	4	15	15

Table III
TWO INDEPENDENT VECTORS

Tables I–VI 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 set, the read set, and the union of the read and write set.

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

We have run the same examples with bounded queues for the communication. The results of the simulation are shown in Tables IV–VI. As the queues are bounded we see only small read and write sets. From the results in Table V 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

Thread	Trans.	Retries	Address set Write	Read	R & W
Producer	1000	999	3	14	14
Consumer	5359	637	2	9	9

Table IV
SINGLE QUEUE

Thread	Trans.	Retries	Address set Write	Read	R & W
Producer	5317	208	3	14	14
Mover	1003	1006	4	21	21
Consumer	8420	269	2	9	9

Table V
TWO QUEUES

Thread	Trans.	Retries	Address set Write	Read	R & W
Producer 1	1000	999	3	14	14
Consumer 1	5360	636	2	9	9
Producer 2	1000	999	3	14	14
Consumer 2	5371	633	2	9	9

Table VI
TWO INDEPENDENT QUEUES

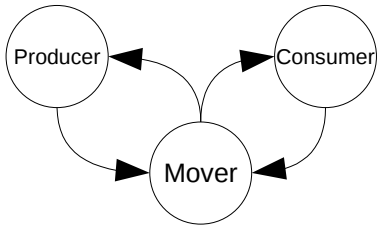

Figure 4. Conflict sets represented as an interference graph for the two queues example.

Producer	
read set	Q_0.wrPtr, Q_0.rdPtr, Q_0.buffer.length
write set	Q_0.buffer[], Q_0.wrPtr

Mover	
read set	Q_0.wrPtr, Q_0.rdPtr, Q_0.buffer.length, Q_0.buffer[]
	Q_1.wrPtr, Q_1.rdPtr, Q_1.buffer.length
write set	Q_1.buffer[], Q_1.wrPtr, Q_0.rdPtr

Consumer	
read set	Q_1.wrPtr, Q_1.rdPtr, Q_1.buffer.length, Q_1.buffer[]
write set	Q_1.rdPtr

Figure 5. Read- and write-sets computed by the points-to analysis.

sections more often finding the queue either full (Producer) or empty (Consumer). (2) As the test for full and empty does 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 RTTM 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.

B. Preliminary Analysis Results

As discussed in Section IV-A, the program analysis is a cornerstone for the successful identification of possible conflicts between threads and their corresponding transactions. To gain some initial insights we have evaluated the feasibility of the proposed program analysis using the open source analysis library WALA [1]. The existing flow-insensitive, context-sensitive points-to analysis [26] was used to analyze the sample programs. In particular, we were interested in the memory locations referenced within transactions of the worker threads. All touched memory locations were summarized using points-to sets within functions of interest, e.g., the queues internal implementation, that are intersected in order to identify possible transaction conflicts.

The analysis builds an abstract representation of the program's heap based on allocation sites, i.e., program points where new objects are allocated on the heap. Two live objects allocated at different allocation sites may never reside at the same memory location. Our results show that a context-insensitive points-to analysis is not suited for our purpose, as it is not able to distinguish allocation sites within classes that are often reused. For example, the internal buffers of queues, lists, and vectors are usually allocated within the container's constructor. Context information is needed to be able to distinguish accesses to these internal data structures.

We have found that analyzing the standard primitives of the Java class library is considerable more complex with respect to context-sensitivity. This is not surprising as the standard library is not specifically intended to be analyzable or to be used in real-time applications. Objects and buffers are more frequently allocated and reallocated at various program points within libraries, often combined with a large depth of the call tree. This leads to a considerable higher complexity for the points-to analysis and consequently vague points-to relations. In addition, data structures are more often copied resulting in large read and write sets and thus pessimistic results for the analysis of transaction overflows.

Nevertheless, for all examples presented in the last section the analysis was able to correctly identify the conflicts between transactions. The resulting conflict sets for the *two queues* example is shown as an interference graph in Figure 4. The analysis correctly identifies 2 queues Q_0 and Q_1 that are accessed within the transactions of the Producer, Mover, and Consumer threads. The internal buffers of these queues are allocated in the queue's constructor and are never reallocated or copied. It is possible to distinguish them using a minimum length of the call string of 2 in our example.

Figure 5 depicts the read and write set computed by the points-to analysis. As can be seen the Producer thread may conflict with the Mover thread via the write pointer and the internal buffer of its queue ($Q_0.wrPtr$, $Q_0.buffer[]$). The conflict for the opposite direction is caused by the corresponding read pointer ($Q_0.rdPtr$). The same applies to the conflicts between Mover and Producer, except for the interchanged roles of the two threads. Most importantly the read and write sets proof that the Producer and Consumer threads may *never* interfere.

The analysis of the other examples yields similar results. In particular, the examples with independent queues and vectors are of interest. The read and write sets determined by the points-to information show that only one producer may conflict with its corresponding consumer and vice versa. However, conflicts with the other running threads are impossible.

An interesting problem arises for the benchmarks based on the standard Vector implementation. The internal buffers of theses vectors can dynamically grow over time and thus need to be reallocated and copied. This poses two problems to the analysis: (1) the size of the read and write sets is virtually unbounded and (2) the minimum context required for accurate points-to information grows. The first problem arises from the fact that the current size of the vector is not known statically – in fact it cannot be predicted for the considered benchmarks at all. When the vector needs to be expanded it is not clear how many elements need to be copied, consequently all Producer threads of the vector benchmarks potentially overflow. The second problem arises from the programming style employed for the implementation of the Java runtime library. Common operations are factored out and (partially) distributed across different classes and methods. This increases the minimal call string length required to achieve accurate points-to information and thus complicates the analysis. For large programs this may lead to imprecise data, because a full analysis is too expensive in terms of memory consumption and computation time.

We conclude that the analysis techniques available today are powerful enough to achieve accurate data on conflicts between atomic sections and potential transaction overflows. However, as can be seen for the implementation of the standard Vector, programmers of real-time applications need to take the limitations of the analysis and the underlying hardware into account.

C. Hardware Implementation

We have implemented a first prototype of the transaction buffer for the write set in a field-programmable gate array (FPGA). As expected the high associativity of the transaction buffer results in a high resource consumption and limits the maximum clock frequency. Table VII shows the results for a single buffer in a low-cost Altera Cyclone-I FPGA. The resource consumption is given in logic cells (LC) and in memory bits. The design was constraint to meet a maximum clock frequency of 100 MHz.

associativity	LC	Memory	Fmax
16-way	528	0.5 KBit	137 MHz
32-way	937	1 KBit	121 MHz
64-way	1768	2 KBit	113 MHz
128-way	3425	4 KBit	103 MHz
256-way	6743	8 KBit	94 MHz

Table VII
IMPLEMENTATION RESULTS FOR A TRANSACTION BUFFER

To set the numbers in relation to a processor core: the current version of JOP consumes 3590 LCs and the maximum frequency in the Cyclone-I device is 93.6 MHz. Therefor, a buffer for up to 256 words would be feasible without restricting the maximum clock frequency. However, the resource consumption for such a *large* buffer is prohibitive. The table shows the resource consumption for the write set buffer; the read set buffer will consume about the same amount of hardware.

Due to the high hardware cost, the transaction buffer should be reused as normal, high-associative data cache outside of a transaction. Another option is to use a shared tag memory for the read and write set. The results from the simulation suggest a common tag memory as most write addresses are also in the read set. In that case, the tag memory is extended with a read and a write bit. On a commit only the entries where the write bit is set are written back to the memory.

VI. CONCLUSION

This work represents the first steps towards a new synchronization paradigm for hard real-time systems. We have introduced real-time transactional memory and explored design issues on chip-multiprocessors. We showed that the maximum number of retries of this optimistic concurrency protocol can be bounded for periodic threads. Our simulation results and first analysis results are encouraging and show that a simplified programming model with bounds on the number of transaction aborts and time predictability is achievable.

ACKNOWLEDGEMENT

This research has received partial funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOPARD).

REFERENCES

[1] WALA - T.J. Watson Libraries for Analysis. http://wala.sourceforge.net/, 2008. Version 1.2.1.

[2] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In *SAS '96: Proceedings of the Third International Symposium on Static Analysis*, pages 52–66. Springer-Verlag, 1996.

[3] J. Anderson, S. Ramamurthy, M. Moir, and K. Jeffay. Lock-free transactions for real-time systems. In *Real-Time Database Systems: Issues and Applications*. Kluwer Academic Publishers, Norwell, Massachusetts, 1997.

[4] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[5] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. K. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2004)*, pages 1–13, Boston, MA, USA, October 2004.

[6] 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*, 2004.

[7] T. Harris and K. Fraser. Language support for lightweight transactions. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, pages 388–402, Seattle, Washington, Nov. 2003.

[8] 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.

[9] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic-sized data structures. In *ACM Conference on Principles of Distributed Computing*, pages 92–101, 2003.

[10] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300. ACM Press, 1993.

[11] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufman, 2008.

[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] A. S. Krishna, D. C. Schmidt, and R. Klefstad. Enhancing Real-Time CORBA via Real-Time Java Features. In *24th International Conference on Distributed Computing Systems (ICDCS 2004), Hachioji, Tokyo, Japan*, pages 66–73, Mar. 2004.

[14] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: Is it worth it. In *CC '06: 15th International Conference on Compiler Construction*, volume 3923, pages 47–64. Springer, 2006.

[15] 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.

[16] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, 2005.

[17] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 327–338. ACM, 2007.

[18] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 308–319. ACM, 2006.

[19] 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.

[20] 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, 2006.

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

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

[23] D. Sharp. Real-time distributed object computing: Ready for mission-critical embedded system applications. In *Proceeding of the Third International Symposium on Distribtued-Objects and Applications (DOA'01)*, 2001.

[24] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*, pages 204–213, Aug. 1995.

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

[26] 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. ACM, 2006.