Transactional Memory for Dependable Embedded Systems

Christof Fetzer  
*Dresden University of Technology  
Dresden, Germany  
christof.fetzer@tu-dresden.de

Pascal Felber  
*University of Neuchâtel  
Neuchâtel, Switzerland  
pascal.felber@unine.ch

Abstract—Transactional Memory (TM) has been touted as one of the most promising approaches to concurrent programming for multi-core processors. By combining ease of use with high scalability potential, as well as checkpointing capabilities particularly useful for developing dependable software, TM has attracted considerable attention from the research community. Many of its facets have been studied over the last few years: hardware support, software TM runtimes, operating system extensions, transactional compilers, language extensions, or application workloads. On the basis of our experiences as designers and users of a complete TM stack that we developed over the last five years, we discuss in this position paper our view on the challenges one faces when extending TM to dependable embedded systems. Indeed, there is an apparent contradiction between the optimistic, best-effort operation of TM and the strict dependability requirements of embedded systems. Our position is that it is both possible and worthwhile to develop embedded transactional memory. Yet, we believe that in the context of dependable embedded systems the focus of TM should be on failure control and not concurrency control. Hence, this will require modifications of the TM language primitives, tools, algorithms, runtime systems, and hardware itself.

Keywords—Transactional memory; embedded systems; dependability; concurrency.

I. INTRODUCTION

Transactional memory (TM) [1] is a recent programming paradigm proposed as a means to support lightweight transactions in concurrent applications. Transactions execute concurrently and those that fail to commit automatically roll back and restart their execution. TM proposes an interesting alternative to lock-based synchronization, offering benefits in terms of scalability and ease of programming. Its automated checkpoint/roll-back mechanisms also provide a promising perspective for dependable computing.

TM can be implemented in hardware (HTM), software (STM), or a hybrid combination of both (HyTM) [1]. The earliest designs were hardware-based [2], [3]. The first STM proposals were operating on a predefined set of memory locations [4]. Research on STM has really taken off during the last decade when the first dynamic designs have been proposed [5], [6]. Commercial HTM support has been proposed for real CPUs only recently, in the Rock [7] and Azul [8] processors, as well as in AMD’s ASF proposal [9].

So far, the main application domain of TM is optimistic synchronization in concurrent programming. TM alleviates some of the drawbacks of locks:

- coarse-grained locking does not scale well with increasing thread counts;
- fine-grained locking is error-prone and difficult to reason about for non-trivial data structures;
- locks are subject to problematic issues such as priority inversions or deadlocks; and
- locks are not composable.

We are currently witnessing the apparition of multi-core CPUs for embedded systems. The benefits of TM in terms of scalable concurrency control make it also an attractive option for embedded systems. Moreover, TM might in some workloads be more energy-efficient than locks [10], which might represent another compelling reason for targeting embedded systems with limited power. Yet, the very principles underlying TM, best-effort optimistic synchronization, seem to be in contradiction with the strict requirements of dependable systems.

We argue in this paper that, in the domain of embedded systems, the use of TM for failure control, i.e., to improve the dependability of systems, is a more promising than its use for concurrency control. As a transaction executes in isolation and either commits, or rolls back automatically all its changes, TM can be leveraged to implement various semantics for robust software and hardware. In particular, TM provides straightforward support for failure atomicity, an important property when recovering from errors: it guarantees that an operation that cannot complete due to an unexpected event will leave the target component in its state prior to the invocation of the operation.

In this position paper, we discuss the challenges one would face when designing a TM for dependable embedded systems. We conclude that, while it is possible and worthwhile to develop embedded transactional memory, this will require some deep changes to the classic TM designs that have been developed so far.

II. DEPENDABLE EMBEDDED SYSTEMS AND TM

There exist a variety of application domains for embedded systems. As a consequence, there also exist different notions and definitions of the term “embedded system”. According to [11], an embedded system is “a microprocessor-based
A. TM for Dependability

There have been only limited research efforts on using TM for developing dependable software, and conversely for implementing dependable TM infrastructures. In the rest of this section, we briefly summarize previous work on TM and dependability in the context of large multi- and many-core systems. We will then discuss in Section III the challenges faced when trying to apply these techniques to dependable embedded systems.

Software Failures: Interestingly, a rollback mechanism very similar to TM was already described in 1975 for the recovery from software bugs [12]: a recovery block is protected by a postcondition and if the postcondition is violated, the state is rolled back using a combination of hardware and software. After the rollback, an alternative implementation of the block is executed, e.g., using an older version of block’s code or a slower but safer variant.

TM can be used not only in the context of recovery blocks, but also to tolerate other types of software bugs, for example, to ensure the atomicity of exception handling [13]. A combination of TM with adequate language extensions and semantics allows various types of error recovery. Notably, one can straightforwardly implement graceful degradation, selective retries, repair and retry, and recovery blocks with TM [13].

For completeness, we should point out that TM can be used for robustness in a variety of other contexts, notably for discovering concurrency bugs [14], for verifying data invariants [15], and for software testing [16].

Hardware Failures: TM can also be used to tolerate hardware bugs. For example, Razor circuitry is a promising approach to balance energy consumption, processing speed, and error rate of a CPU [17]. However, the error rate of a commodity CPU will be tuned for its target market and will most likely be too high for dependable applications. It seems worthwhile to investigate the use of software/hardware mechanisms to decrease the error rate perceived by a dependable application executing on a (commodity) CPU. In particular, the combination with HTM or HyTM to efficiently rollback erroneous executions with a software-based error detection mechanism like SWIFT (software implemented fault-tolerance) [18] could be a practical approach. FaultTM [19] uses a hybrid hardware/software TM approach to execute vulnerable code sections in two concurrent transactions and detect errors by comparing the associated write sets before commit.

B. Dependability for TM

If one wants to use a TM infrastructure to develop dependable software, the underlying TM must itself satisfy some requirements in terms of dependability. In particular, it must provide liveness guarantees so that, if a thread is delayed or fails inside a transaction, the other threads are not blocked.

Liveness properties can be provided by a lock-free TM [20], an obstruction-free TM in combination with a robust contention manager [21], or an “almost wait-free” TM like ROBUSTM [22]. The latter is particularly interesting as it was specifically designed to tolerate crashed and non-terminating threads, and it provides performance comparable to that of non-robust state-of-the-art TMs.

Finally, to deal with the unpredictability of transaction execution times, e.g., due to aborts/retries or thread preemption, one can use (soft) real-time TMs [23], [24]. These implementations can help bound the execution time of transactions, e.g., by relying on conservative scheduling techniques as deadlines near, but we are not aware of any TM that would provide comprehensive hard real-time guarantees.

III. TM Challenges for Embedded Systems

In the context of dependable embedded systems, we foresee several challenges that would deserve further research to determine how far TM could help alleviate them. In this short paper, we focus on the following challenges:

Challenge 1: To satisfy dependability requirements, strict spatial, temporal, and failure isolation of tasks is required.

Tasks running on an embedded system can have various criticalities. One needs not only to ensure that tasks of lower criticality cannot negatively affect tasks of higher criticality, but also that faulty tasks of the same or higher criticality do not negatively affect tasks of lower criticality. Hence, one wants to enforce a strict spatial, temporal, and failure isolation of tasks. Such isolation is difficult to achieve because tasks need to communicate to achieve the dedicated functions of the embedded system.

TM can be used to protect access to shared memory, as illustrated in Figure 1. TM synchronizes access to shared memory by enforcing atomic execution of transactions. Atomicity has the potential advantage that consistency and liveness can be enforced even in situations when failures
occur. However, failures like process crashes are typically not considered in current TM implementations. A notable exception is RobustTM [22], which was developed in the context of the EU FP7 VELOX project. The advantage of TM is that, in case a process crashes while updating a shared region, all its changes can be rolled back and, in this way, consistency can be preserved. In comparison, when using a lock-based approach, a process crash within a critical section would prevent all other processes from entering that section.

RobustTM provides also some basic notion of fairness: all transactions will eventually be committed even if they are executed by a slow process. In the context of dependable embedded systems, we need to go beyond handling fail-stop failures and providing simple fairness. In particular, a stricter isolation of processes needs to be ensured. For example, spatial isolation could be improved by using a combination of isolated address spaces, TM-protected shared memory segments, and some form of address space randomization. Temporal isolation requires that the resource usage of a process does not invalidate the timeliness requirements of real-time processes.

Challenge 2: To satisfy real-time constraints, we need to extend the TM language primitives and the runtime systems.

The main advantage of TM in the context of real-time constraints is that it does not suffer from the priority inversion problem. In this sense, TM seems to be more appropriate than lock-based synchronization. However, there are other issues that need to be addressed, notably a low priority task could slow down a high priority task.

Consider the situation sketched in Figure 2. A low priority task accesses a shared region very frequently, resulting not only in cache misses on a high priority task but potentially causing aborts of the high priority task. Moreover, the high-priority task might depend on the arrival of updates that could be delayed by too high access rates.

To address these issues, we believe that the TM primitives need to be extended to include real-time and access guarantees. Furthermore, the runtime system needs to be extended to support real-time guarantees. It seems one possible approach could be to design and implement a TM following a time-triggered approach [25]: access to shared memory is only permitted during predetermined time slots and the TM needs to not only prevent accesses outside a thread’s time slot, but also to rollback partial updates in case a thread fails to commit its changes within its time slot (see Figure 3). Note that, in this context, the speculation of a transaction would not be used for concurrency control but to ensure failure atomicity: the effects of a software or hardware failure could be rolled back. The next challenge is how to detect such failures.

Challenge 3: Ensuring consistency in the face of software and hardware faults.

TM could be used to ensure the consistency of updates in the face of failures. One possible approach to detect execution failures of the CPU (i.e., to detect the incorrect execution of the program) is the aforementioned combination of SWIFT [18] and TM. Alternatively, executable constraints could be used to detect both software and hardware failures. Software failures could be detected and tolerated in a similar manner as the recovery block approach [12]. Hardware failures can be detected with a high likelihood using appropriate executable constraints [26]. To do so, a TM could be checking executable constraints before committing a transaction. While this concept is conceptually simple, none of the standard TMs currently support this feature.

Note that if a TM is not used for concurrency control anymore (e.g., when combined with a time triggered approach), transaction demarcation can be changed. Instead of adding atomic blocks to programs, we would add executable constraints. A new transaction is started automatically after committing a previous transaction or returning from a system call (see Figure 4). A transaction is automatically committed when control flow reaches an executable assertion or a system call. In this way, one can eliminate the two main disadvantages of TM: (1) its inability to cope with external actions, and (2) its inefficiency at executing large transactions. Regarding (1), a transaction must be committed before a system call not only to avoid rolling back external
actions but also to validate the correctness of the arguments of the system call. Regarding (2), by automatically [27] or manually deriving and adding invariants to a program, one can limit the size of transactions.

Figure 4. When focusing on failure control, transactional demarcation can be set (implicitly) by executable constraints and system calls. Constraints can be added manually or automatically.

Challenge 4: Integration with replication mechanisms.

To detect wrong executions and/or to ensure availability in the face of failures, dependable embedded systems often rely on some form of replication. In many instances, active replication is used, i.e., each request is executed by a set of processes. In this way, instant fail-over can be ensured. Temporal replication is also used, for example, to detect execution failures.

Active replication of multi-threaded applications is difficult since the scheduling of threads introduces non-determinism. This non-determinism causes replicas to diverge. A divergence needs to be detected and the replicas stopped to ensure safety. To support active replication, a TM would need to integrate (or enforce itself) a form of deterministic execution scheduling. Deterministic execution typically introduces some non-negligible overheads. However, the use of a time-triggered TM would already ensure determinism (and not introduce any additional overhead).

An interesting use of TM is to roll back replicas in case a divergence is detected. The challenge that we foresee is that replicas would need to be synchronized frequently to limit the size of the rollbacks. Note that such rollback would be applicable both for active and temporal replication. We expect that some original TM-based replication mechanisms could be developed in this context.

IV. Concluding Remarks

We believe there are interesting uses of TM for dependable embedded systems. TM mechanisms might however be used less for concurrency control than for failure control. In particular, we do not expect the widespread use of speculation for concurrency control because of the need for hard real-time guarantees. In this context, we anticipate that one will stick with more conservative approaches like a time-triggered approach and use the TM to deal with software and hardware failures.

Considering that transactions were originally introduced to provide clean failure semantics, the use of transactions for failure control might not come as a surprise. However, we believe there are some interesting research questions left to be explored. In particular, the TM usage patterns, language integration, and runtime system will have to be dramatically modified. Most notably, these changes will likely eliminate some of the main problems of TM: coping with external actions and long transactions. Moreover, by eliminating concurrency control issues, the overhead of TM could be reduced. This would need to be investigated more closely.

We strongly believe that it would be a very worthwhile effort to evaluate the potential advantages, but also the disadvantages, of TM for dependable embedded systems. As argued in this short paper, we expect this will require modifications of the TM language primitives, tools, algorithms, runtime systems, and hardware itself. This would represent exciting new research directions in the domain of dependable embedded systems.

ACKNOWLEDGMENT

The research leading to these results has received funding from the European Community’s Seventh Framework Programme (FP7/2007-2013) under grant agreement 216852 (VELOX).

REFERENCES


