Tightening Critical Section Bounds in Mixed-Criticality Systems through Preemptible Hardware Transactional Memory

Benjamin Engel
Operating-Systems Group
Department of Computer Science
Technische Universität Dresden
Email: ⟨name⟩.⟨surname⟩@tu-dresden.de

***Abstract*—Ideally, mixed criticality systems should allow architects to consolidate separately certified tasks with differing safety requirements into a single system. Consolidated, they are able to share resources (even across criticality levels) and reduce the system's size, weight and power demand. To achieve this, higher criticality tasks are also subjected to the analysis methods suitable for lower criticality tasks and the system is prepared to relocate resources from lower to higher criticality tasks in case the latter risk missing their deadlines. However, non-preemptible shared resources defy separate certification because higher criticality tasks may become dependent not only on the functional behavior of lower criticality tasks but also on their timing behavior. For shared memory resources, hardware transactional memory (HTM) allows to discard changes made to the resource and roll back to a previous state. But instead of using HTM for conflict detection and synchronization, we use this hardware feature to abort low critical shared resource accesses in case they *overrun their time budget*.**

In this paper, we present the results from extending HTM to allow transactions to become preemptible in order to support mixed criticality real-time shared resource access protocols. We implemented a lightweight cache-based HTM implementation suitable for embedded systems in the cycle accurate model of an out-of-order CPU in the Gem5 simulation framework. The software implementation using this extension in a priority-ceiling shared resource access protocol complements our work and demonstrates how transactional memory can be used to protect higher criticality tasks from untimely lower criticality tasks despite shared resources. Our simulation with synthetically generated tasksets show a reduction in system load of up to 22 % compared to scheduling LO resource accesses with HI bounds and a schedulability improvement of up to 54 % for state-of-the art real-time locking protocols. We used a LO-to-HI ratio of 1:1.2 – 1:2 and loaded the system between 50 % – 75 %.

I. Introduction

Announced in 2007, but later cancelled, Sun's Rock processor [1] was supposed to be the first production-ready CPU to include hardware transactional memory (HTM) [2]. Four years later, IBM's 3rd generation BlueGene/Q [3] fulfilled this promise by providing HTM functionality to high-performance computing, followed in 2014 with Intel's implementation [4] for general-purpose desktop and server systems. We expect cache-based HTM implementations to soon make their way into embedded processor architectures. For example, the open-source RISC-V ISA [5] already contains a placeholder for transactional memory instructions. Ferri et al. [6] identified energy and throughput improvements for accessing contented resources in a simulated ARM multiprocessor system-on-a-chip of 30 % and 60 %, respectively.

In its simplest version, cache-based HTM implementations keep transactional data stored in the cache until the transaction is committed. The cache will continue to respond normally to coherence requests, but accesses from other CPUs (writes to cached transactional data and reads to dirty transactional state) will cause an local abort and the invalidation of all cached transactional state. The result is either that the complete transaction becomes visible (in case the cache returns to normal operation) or the core reacts as if the transaction did not happen (by invalidating all transactional state).

In this paper, we exploit this all-or-nothing effect of transactions in mixed criticality systems to protect resources that are shared across criticality levels.

Mixed criticality is about consolidating tasks with different certification requirements into a single system. In his seminal work, Vestal [7] observes that independent tasks can be integrated in such a way by ensuring that higher criticality tasks can still meet their deadlines, even if they have failed to do so when they were scheduled with more optimistic scheduling parameters of lower criticality levels. Baruah et al. [8] calls this interpretation of mixed criticality systems *certification cognisant* as it maintains the increasing pessimism that is imposed by evaluation criteria to assert correct and timely operation of more safety critical tasks. In this paper, we adopt this certification cognisant interpretation of mixed criticality systems.

Unfortunately, the independence assumption is not very realistic in practical systems because in general tasks share resources that are not as easily preemptible as the CPU. For single criticality systems, a wealth of resource access protocols have been proposed following the early works of Baker [9] and Sha et al. [10] to bound priority inversion [1] and minimise blocking times. Priority inversion occurs if a lower prioritised job prevents a higher prioritised job from running because it is holding a resource that the latter needs or because the resource is otherwise inaccessible due to the mechanics of the resource access protocol.

[1]For ease of presentation, we use a priority based formulation for all preemption conditions and leave it as future work to adjust this formulation to preemption levels for EDF-based locking protocols.

In mixed criticality systems arises a second problem, which has led to a debate whether resources should actually be shared across criticality levels: *the trustworthiness of the resource after a lower criticality access*. For example, in [11], Burns takes the view that with the exception of some cryptographic protocols, resources should not be shared across criticality levels. He introduces MC-PCP to prevent unbounded priority inversion among jobs of the same criticality level. Brandenburg [12] on the other hand takes a much more radical approach and requires all resource accesses to be executed in a server, which assumes the criticality level of the highest criticality resource accessing task.

We take the view that resource sharing across criticality levels should be possible without having to subject resource accesses to a timing analysis at this highest criticality level. Instead we use available hardware features, namely transactional memory, to enforce timely bounds on shared resource accessed from low criticality tasks. Unfortunately, IBM BlueGene, although successful in high-performance computing is typically not widely used in real-time and mixed criticality systems. We therefore extend a simple x86 cache-based HTM implementation with support for a single preempted transaction and report in Section III about the implementation of this HTM variant in the cycle accurate model of an out-of-order CPU in the Gem5 hardware simulator. In Section IV, we evaluate the performance of our approach before we draw conclusions in Section V.

We are confident that it is much easier to establish partial correctness (i.e., that if the resource access terminates, the resource will be in a good state) than establishing the timeliness of such accesses. In particular, establishing partial correctness with sufficient confidence is still possible if the code is incompatible with sophisticated timing analysis tools. Our main contribution of this paper is to provide a means to ensure the timeliness of lower criticality accesses by executing them transactionally. We use the hardware feature of transactional memory not for synchronizing access to shared resources (the usual locks are still in place), but to quickly abort low critical shared resource accesses that violate their time bounds.

II. Background and Related Work

In this section we describe the foundations our research builds upon, namely hardware transactional memory (HTM) as a feature of modern processors and real-time locking protocols like immediate-ceiling or inheritance based protocols for controlling the access to a shared resource. We combine both in a mixed criticality system, where low critical tasks can be aborted if they overstep their temporal bounds or if higher critical tasks overstep their optimistic scheduling parameters and actually need to be scheduled with more pessimistic ones.

A. Hardware Transactional Memory

As of today, IBM Blue Gene/Q [3] has the most elaborate HTM implementation. By versioning data in the shared L2 cache, Blue Gene/Q is able to maintain multiple transactional states in parallel, which allows them to roll back later transactions if they conflict with earlier ones. Both, IBM's and Intel's HTM, have dedicated instructions to start and end a transaction. Within a transactional region, updates to memory are kept local to the CPU and are not visible to other processors. When the transaction finishes, it tries to *commit* all changes atomically and thereby makes them visible to other CPUs. If this commit fails, no changes are written back at all, the transaction is said to be *aborted* and the CPU state is rolled back to the state before the transaction was started to do proper error handling. We use this all-or-nothing approach when accessing shared resources within temporal bounds.

Cain et al. [13] give a very detailed description of the transactional memory system, its hardware implementation and suggested OS, and application programming models for the IBM Power architecture. Interestingly, this paper also explains in detail how and why they allow suspending and resuming transactions. Rather than aborting transactions, interrupts preempt transactions. In addition, transaction preemption and resuming is made available to developers through explicit instructions: `tsuspend` and `tresume`. The authors thoroughly evaluate the costs and benefits and show that transaction suspension is a valuable feature when building robust and reliable systems.

In this work, we propose a more lightweight implementation of transaction suspension for x86 that advances Intel's Transactional Synchronization Extensions (TSX). Although most implementation details of TSX [4] remain confidential, some parts may be inferred from released information in the Intel developer and optimisation manuals, which indicate a L1D cache-based implementation.

B. Real-Time Locking Protocols

In this paper, we consider both single and mixed criticality resource protocols, which we classify by the mechanism used to guarantee bounded priority inversion:

- **immediate-ceiling based protocols**, such as the stack resource [9] (or ceiling priority [14]) protocol (SRP), immediately raise the priority of resource acquiring threads to a resource dependent ceiling priority. By preventing released threads at a lower priority from executing, they seek to ensure that all resources are readily available once the thread starts executing.
- **inheritance based protocols**, such as the priority inheritance protocol (PI) and the original priority ceiling protocol (OPCP) by Sha et al. [10], allow preemptions of resource holders by higher prioritised threads but *help out* the resource holder in case a thread requests a resource by raising its priority to the priority of the higher prioritised, blocked thread. We distinguish between *local helping* (i.e., helping out a resource holder on the same CPU) and *global helping* (i.e., pulling the resource access from a remote CPU to the local CPU) and restrict ourselves to local helping protocols only. The rationale is that global helping would require transferring transactional state from one CPU to another, a complexity we are not willing to take into account when extending our cache-based HTM implementation.

Single criticality protocols of the first class are the multiprocessor variants MRSP by Gai et al. [15] and FMLP by Brandenburg et al. [16]. Both execute global resource accesses (i.e., resources accessed from threads on multiple cores) non-preemptively, which corresponds to raising the priority of

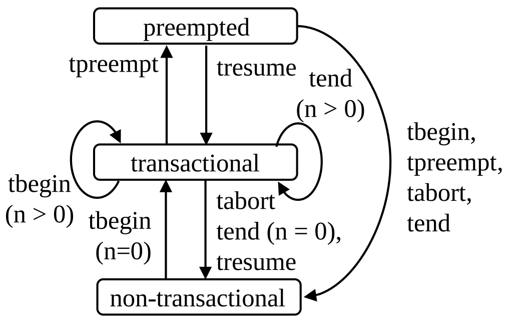

Fig. 1: States of cache controller for preemptible transactions.

the resource accessing thread to the maximum priority of threads on its core. Zhao et al. [17] extend the stack resource protocol to work with EDF schemes in which threads have more than one deadline to accommodate mode changes. As a member of the second class, Burns [11] extends the analysis of OPCP to consider criticality dependent blocking terms. Avoiding resource sharing across criticality levels, Burns allows local helping only between tasks of the same criticality. Single criticality protocols with local helping include the partitioned multiprocessor priority inheritance protocol [18] and similar variants for EDF [19]. The clustered O(m) locking protocol [20] and Brandenburg's inter-process communication scheme [12] apply global helping and are therefore not considered in this work in their original form. However, it is possible to modify the former to apply local helping (i.e., inheritance) only and we address this variant. Lakshmanan et al. [21] integrate ceiling (PCCP) and inheritance (PCIP) in their slack based scheduling approach to allow resource sharing across criticality levels. In addition to inheriting priority, they propose to also inherit criticality to prevent tasks from being suspended by low criticality tasks. Both PCIP and PCCP are single processor variants with local helping and ceiling, respectively.

III. Preemptible Transactions in the Gem5 Out-of-Order Model

Gem5 is a modular simulation framework with various CPU, memory, device and cache models. At the time of writing, there was already an HTM implementation [22] in Gem5, which is based on LogTM [23]. However, it was not built for the cycle accurate Out-of-Order CPU model (O3CPU) but for a simpler, less timing precise model. Moreover, its implementation was based on an undo log (like PARs [24]) whereas we focus on cache-based implementations, since available hardware (IBM, Intel) most likely implements transactions in the cache. We therefore started a new implementation in the O3CPU model, which we will introduce shortly in the following before we return to our implementation in Section III-B. Like most modern simulators, Gem5 decouples the internal architecture from the instruction set architecture (ISA) exposed to the user. In this way, Gem5 unifies different CPU models, like AtomicSimple, TimingSimple, and the 5-stage Out-of-Order model we use. Internally, Gem5's O3CPU makes use of a RISC like ISA, called M5, whereas user ISAs can be x86, ARM and others.

A. *Out-of-Order CPU Model*

Currently the most advanced CPU model in Gem5 is the 5 stage pipelined Out-of-Order CPU model, which loosely resembles an Alpha 21264. It implements the following usual pipeline stages: fetch, decode, rename, issue + execute + writeback, and commit. Issue forwards instructions to specific queues where they are processed by the execution units and the memory subsystem in the order in which their parameters become ready. Relevant for this work is the load/store queue and the ordering enforced by the memory barrier instruction.

The CPU model is event-driven and timing costs are attached and accumulated at each individual step. An external clock drives the CPU and creates 'ticks' for each of its stages to advance the model in a cycle-precise fashion. The number of instructions that can be fetched, decoded, issued and sent to the execution units is configurable. The delay and the bandwidth in each step, the delay of caches, the traversing of multiple ports, and the accumulating lookup-, forward- or data-copying delay are also subject to configuration. For our evaluation in Section IV, we use the default configuration for the Out-of-Order CPU, with a L1 instruction and L1 data cache of 32KB each and a 256 kB unified L2 cache. Cachelines store 64 bytes. The associativity of L1D is 4, 8 for L1I, and 16 for the L2 cache. Although modern CPUs have shared L3 cache, we did not add it, since transactional data will solely be placed in the L1 data cache. The cache one level beneath is important for the simulation, but multiple levels do not add any further detail.

B. *Preemptible transactions in O3CPU*

Based on publicly available information, we recreated part of the restricted transactional memory (RTM) implementation proposed by Intel [4][2]. More precisely, we augmented the L1 data cache with additional state —the **T** bit— to distinguish transactional from non-transactional data and extended the logic for the MOESI cache coherence protocol to react accordingly.

We chose to implement basic RTM functionality on top of MOESI although Intel CPUs implement MESIF because a MOESI protocol implementation was already present in Gem5. Common to both protocols are the cacheline states **I**nvalid for empty cachelines, **E**xclusive for data that has not yet been modified and that is present only in this cache, **M**odified for exclusive data that has been modified and **S**hared for data that may exist with the same value in multiple caches. **O**wned cachelines allow sharing of dirty data by delaying the write back to the time of eviction. The data in memory might be stale, but the cacheline is shared. **F**orward is a similar variant of **S**, which allows the forwarding cache to respond, instead of the underlying memory.

We first describe the modifications required to put the CPU and the caches in transactional and transaction preempted state before returning to the coherence protocol and how transactions change the state machine of the cache controller.

[2]Notice, while we added the full user functionality of RTM, including nested transactions, we leave the triggering of transaction aborts in all kind of exceptional cases as a future engineering task. For example, we added all instructions to begin, end and abort a transaction but do not trigger the abort mechanism when the page-table walker experiences a page fault.

Figure 1 illustrates the transaction states and the transitions assuming aborts are eager. To implement these state changes, we added three control signals to the CPU —HTM-ENABLE, HTM-COMMIT and HTM-ABORT— and interpret them in the load/store unit and in the cache controller. The outermost TBEGIN instruction transitions the CPU into transactional mode and informs the cache to start a new transaction. From now on, until the outermost TEND commits or aborts the transaction, all memory accesses will be stored transactionally in the L1 cache with the **T** bit set. Subsequent execution of TBEGIN stays in this state but increases the transaction nesting level, which TEND decreases. The outermost TEND with nesting level $n = 0$ sends a HTM-COMMIT-request to the underlying cache. The TEND instruction will retire not before the cache responds, either with commit or abort. TABORT triggers the abort directly through HTM-ABORT. In all three cases, the cache and the CPU return to non-transactional operation.

To add transaction preemption, we implemented two further instructions TPREEMPT and TRESUME and introduced one additional control signal HTM-PREEMPT to signal that the cache and the CPU are not in preempted transaction mode. TPREEMPT sets this signal, so that further memory-requests are no longer transactional and TRESUME clears it, returning to the previous transaction. Depending on the desired abort behavior, TRESUME will return an error if the transaction was aborted and immediate aborts should be supported. For lazy aborts, TRESUME returns normally but transactions will no longer commit. All other variants (including TBEGIN while a transaction is preempted) map to an abort. Aborts always affect all transactions up to the outermost one.

Special attention needs to be payed on in-flight memory operations and outstanding cache misses, since we cannot commit or abort a transaction that has pending memory requests. For this reason, all transaction instructions have to behave like a full memory barrier, which ensures that earlier memory accesses (including outstanding cache misses) are completed before a mode change is triggered and that later instructions are not started before the instruction is commited by the processor pipeline. In particular, we cannot execute these instructions speculatively because they change the behaviour of the CPU and the cache.

What remains is to ensure that the cache controller reacts appropriately depending on the state it is in and, in particular, that it detects all conflicts that lead to transaction aborts. For that we augment the cache with a vector of **T**ransaction bits (one for each cacheline). To enable HTM, the snoop logic changes its behavior depending on the state of the **T** bit of the affected cacheline. While the cache executes in transactional mode, the snoop logic responds normally to external reads to exclusive (**E**), shared (**S**, **O**) or modified **M** cachelines. However, if the read origins from the local core and targets an **E**xclusive, **S**hared or **O**wned cacheline, it sets the **T** bit to mark these lines as belonging to the read set. Writes put cachelines to the write set by setting the **T** bit in the **M** state. When an external snoop request hits a cacheline that is transactional (i.e. belongs to the read or write set and thus has its **T** bit set), the transaction will be aborted if the snoop request signals an external write (rfo) or a read (busrd) of modified data. An abort unconditionally invalidates all modified cachelines and returns to non-transactional operation.

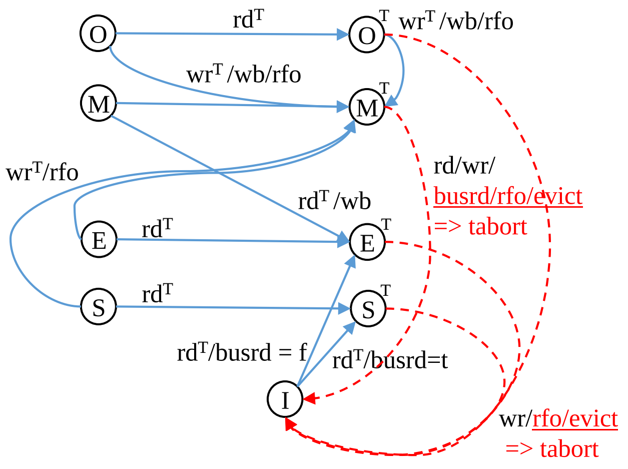

Fig. 2: Augmented coherence protocol for preemptible transactions. We omit the standard MOESI transitions and present in blue (solid) the behavior or transactional reads and writes and in red (dashed) the effect of non-transactional reads and writes on ntransactional data while the transaction is preempted. $.^T$ denotes transactional operations and state, wb indicates a required write back, evict a cache eviction and rfo and busrd are events indicating external writes and reads.

Special care must be taken for non-transactional dirty cachelines that become transactional. Because aborts will unconditionally discard cachelines, we first have to write back dirty cachelines to not lose the old data when aborting the transaction. More precisely, we have to write back **O**wned cachelines before they are written in a transaction and **M**odified lines before they are read or written.

Now, if the cache controller enters preempted transaction mode, the controller has to react to local accesses as if they were external. That is, if a local read hits the write set or if a local write hits this transactional data in the read or write set, the transaction is aborted prior to executing this request. Figure 2 shows the modified transitions of the resulting cache coherence protocol. For better readability, we omitted the transitions of the normal MOESI protocol and only show the transitions due to transactional reads and writes accessing non-transactional data and of non-transactional reads and writes hitting a preempted transaction. All other transitions among the non-transactional MOESI states and among their transactional counterparts are like in the standard MOESI protocol except that evictions of the latter trigger aborts.

To evaluate the costs of transactions, we annotate all steps in this execution with the costs we found for similar instructions (i.e., memory barriers and the signal propagation delay to the cache).

IV. EVALUATION

For the experimental evaluation we generated 1,000 random tasksets with up to 10 tasks and a given maximum utilisation (0.5, 0.75, and 1.0) using the uunifast algorithm [25]. We use a periodic task model with implicit deadlines, in which periods are the product of two randomly chosen factors from the set $[2, 3, 4, 6, 8, 9, 12]$, resulting in a maximum hyper period of 5184. Randomly choosing arbitrary periods from a given range typically results in extremely long hyper periods that can no longer be simulated in a reasonable amount of time. These tasks access shared resources and split their execution time in such a way that the first half in each period is spent outside of the critical section and the second half within. Furthermore we selected 83%, 67%, and 50% of them to be high-critical and increased their high-critical WCET by a factor of 1.2, 1.5, or 2.0 respectively. Thus we roughly have the same utilisation for the low and the high criticality mode. Fig. 3, 4, and 5 show the histogram of 1,000 tasksets. The solid three plots are almost overlapping and depict the distribution of tasks when using preemptible transactional memory, so that low critical tasks accessing their shared resource can be scheduled with their low-WCET. The transactional semantic of the cache allows us to use the more optimistic low criticality bounds when accessing the shared resource. In the case of overrunning the time budget, the timer will fire, the resource access will be aborted and the system changes into its high criticality mode, dropping all low criticality tasks. If the resource access finishes within time, the transaction will commit, the job will finish and the next job will be scheduled.

The dashed three plots show the same taskset when no transactional memory is used to bound the low critical WCET. Hence, we have to use their high critical counterpart for low critical jobs, resulting in a higher overall system load. The low critical WCET to high critical WCET ratios are 1.2, 1.5, and 2.0, i.e. a ratio of $1 : 1.2$ means high-critical WCETs are 20% higher then their low criticality counterpart, reflecting the higher trust and associated higher costs.

In the first experiment we chose an utilisation target of 50%, so that the load of all low-critical execution times sums up to about 50%. Since low-critical tasks share resources with high-critical ones, their WCET to access the resource needs the highest confidence of all sharing tasks. Although a task is low-critical, the resource access has to use its high-critical WCET, which leads to a higher load on the CPU. In this setup we observed up to 72% load, compared to the 50% when not sharing resources between low and high tasks. Even at an assumed very moderate LO to HI ratio of 1.2, already 1% of the tasksets were no longer schedulable, due to missed deadlines. With higher low critical to high critical ratios (1.5 and 2.0 respectively) the deadline misses increased to 6% and 21% of all tasksets. With our proposed hardware extension, we are able to use transactions for low-critical tasks in their critical section and therefore use the lower but less trustworthy low criticality bounds and abort jobs if they overrun their budget. Fig. 4 and Fig 5 show the results when increasing the initial load in the system to 75% and 100%. At a system load of 75%, already 4% of all tasksets (at ratio 1.2), 14% (at ratio 1.5), and 54% (at ratio 2.0) cause deadline misses. The very extreme is at a maximum utilisation of 100%. Due to pessimistic WCET for low critical tasks only 55% of all tasksets were schedulable when assuming a LO to HI ratio of 1.2. At 1.5 or 2.0 virtually 100% were no longer schedulable. This is not surprising, since adding even minor additional load to a very loaded system very likely causes deadlines to be missed. Therefore, we did not plot the actual load, but rather the theoretical load this system would have to handle, if we ignore all occurring deadline misses.

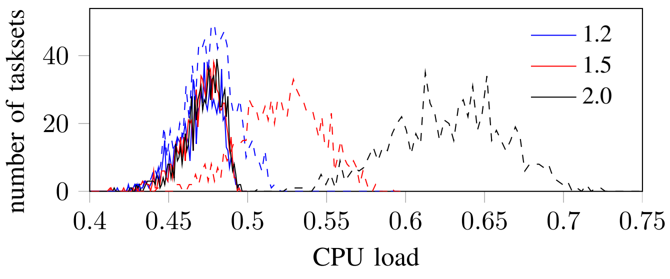

Fig. 3: Taskset with a maximum CPU load of 0.5 and a ratio of 1:1.2, 1:1.5, and 1:2.0 for low-critical to high-critical WCET. Although all tasksets are schedulable with EDF on a uniprocessor, it is clear that the additional pessimism for low critical tasks sharing a resource with a high critical task significantly increases their WCET and thus leads to a higher utilisation, i.e. higher resource demands (or less slack).

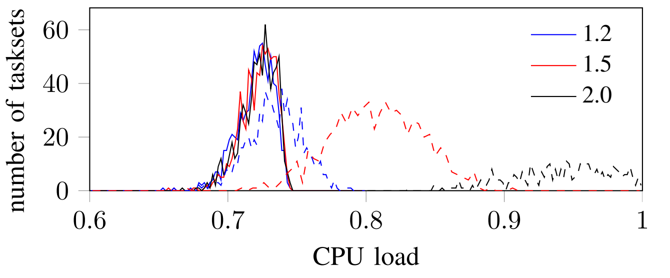

Fig. 4: Taskset with a maximum CPU load of 0.75 and a ratio of 1:1.2, 1:1.5, and 1:2.0 for low-critical to high-critical WCET. At a ratio of 2.0, 54% of the tasksets are no longer schedulable.

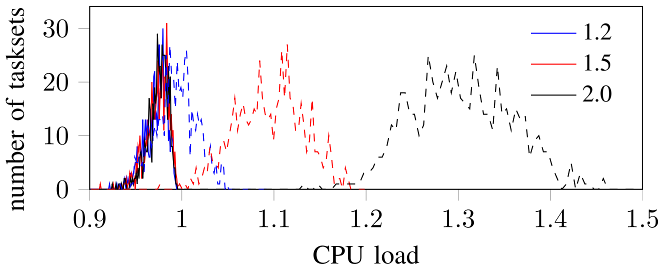

Fig. 5: Taskset with a maximum CPU load of 1.0 and a ratio of 1:1.2, 1:1.5, and 1:2.0 for low-critical to high-critical WCET. The three solid plots show the actual load when using transactions to bind low-critical resource access, whereas the three dashed plots depict the *theoretical* system load, since with a ratio of 1.5 and 2.0 virtually no tasksets were schedulable any longer. So we ignored the deadline misses and report the load the system *would* have to handle.

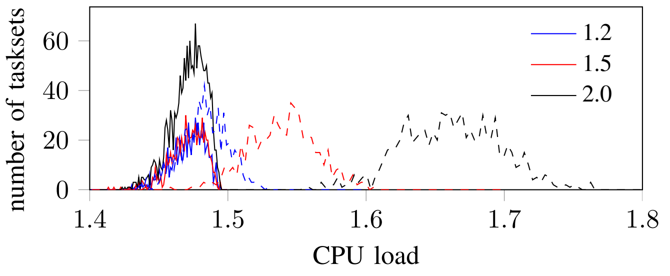

Fig. 6: Taskset with a CPU load of about 1.5, 20% of the WCET is spent in a critical section and in high criticality mode tasks require 1.2/1.5/2.0 times their low-WCET.

To substantiate the feasibility of our approach and to quantify the benefits of using transactional memory in mixed criticality systems, we evaluate a very simple multiprocessor setup. We use the same task model as in the uniprocessor case, generate the tasksets in the same fashion, and use partitioned EDF with one synchronisation processor for accessing global resources according to [26]. To generate tasksets that are still schedulable, the task's first 80% of its execution time is spent outside critical sections, the remaining 20% within. Of all tasks, about 83%, 67%, and 50% of them are classified as high critical and their high-WCET is 1.2, 1.5, and 2.0 times of their low-WCET, respectively. We removed all tasksets which caused deadline misses either in low or high critical mode, Fig. 6 shows the results. As in the uniprocessor case, reliably enforcing low-critical WCETs for shared resource accesses reduces the overall load in the system. Moreover, at a ratio of 1.2, 5% of the tasksets caused deadline misses when *not* using transactional memory to enforce timely bounds on critical sections. At 1.5 this number raises to 26% and with high-WCETs being twice as long as their low-WCETs counterparts 50% of the tasksets were no longer schedulable. This means that approximately one half is plotted, the other half was schedulable with hardware transactional memory enforcing lower WCET bounds, but could not be scheduled without it. This clearly shows the benefit of using preemptible transactional memory in combination with mixed criticality systems to improve schedulability and reduce system utilisation.

V. Conclusions

In this work, we investigated the use of hardware transactional memory (HTM) in real-time locking protocols to make low criticality resource access bounds trustworthy at higher criticality levels. We have seen that although existing HTM implementations are quite limiting or too complex to integrate in embedded systems, a lightweight implementation supporting preemptible transactions significantly broadens the applicability of our approach.

Future work includes extending our HTM implementation to an L2 victim cache to increase the amount of data that can be accessed within a resource access. Also, we did not yet exploit the optimistic locking behavior of transactions when a thread finds a resource blocked. To preserve the real-time guarantees of the legitimate lock holder, support for optimistic locking requires control over which transaction gets aborted (the optimistic) and which will be continued (the lockholder's).

References

[1] M. Tremblay and S. Chaudhry, "A third-generation 65nm 16-core 32-thread plus 32-scout-thread sparc processor," in *International Solid-State Circuits Conference Digest of Technical Papers (ISSCC 08)*. IEEE, 2008, pp. 82–83.

[2] 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*, May 1993, pp. 289–300.

[3] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G.-T. Chiu, P. Boyle, N. Chist, and C. Kim., "The IBM blue gene/q compute chip," *IEEE Micro*, vol. 32, no. 2, pp. 48–60, April 2012.

[4] I. Corp., "Web resources about intel transactional synchronization extension," www.intel.com/software/tsx, July 2014.

[5] A. Waterman, Y. Lee, D. Patterson, and K. Asanović, "The RISC-V instruction set manual volume i: User-level ISA - version 2.0," CS Division, EECS Department, University of California, Berkeley, Technical Report UCB/EECS-2014-54, May 2014.

[6] C. Ferri, A. Viescas, T. Moreshet, I. R. Bahar, and M. Herlihy, "Energy implications of transactional memory for embedded architectures," in *Workshop on exploiting parallelism with transactional memory and other hardwre assisted methods*, April 2008.

[7] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Real-Time Systems Symposium*, December 2007, pp. 239–243.

[8] S. Baruah, H. Li, and L. Stougie, "Towards the design of certifiable mixed-criticality systems," in *16th IEEE Real-Time and Embedded Technology and Applications Symposium*, ser. RTAS. IEEE, April 2010, pp. 13–22.

[9] T. P. Baker, "A stack-based resource allocation policy for real-time processes," in *Real-Time Systems Symposium*. IEEE, 1991.

[10] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronisation," in *IEEE Transaction on Computers, 39*, 1990.

[11] A. Burns, "The application of the original priority ceiling protocol to mixed criticality systems," in *L. George and G. Lipari, editors, Proc. ReTiMiCS, RTCSA*, 2013, pp. 7–11.

[12] B. Brandenburg, "A synchronous IPC protocol for predictable access to shared resources in mixed-criticality systems," in *35th IEEE Real-Time Systems Symposium (RTSS 2014)*, 2014, pp. 196–206.

[13] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le, "Robust architectural support for transactional memory in the power architecture," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 225–236.

[14] N. H. Cohen, "Ada as a second language, chapter real-time systems annex." McGraw-Hill, 1996.

[15] P. Gai, G. Lipari, and M. D. Natale, "Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip." in *Real-Time Systems Symposium*. IEEE, 2001, pp. 73–83.

[16] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson, "A flexible real-time locking protocol for multiprocessors," in *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2007.

[17] Q. Zhao, Z. Gu, and H. Zeng, "Integration of resource synchronization and preemption-thresholds into EDF-based mixed-criticality scheduling algorithm," in *RTCSA*, 2013.

[18] R. Rajkumar, L. Sha, and J. P. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors," in *Real-Time Systems Symposium*. IEEE, 1988, pp. 259–269.

[19] C.-M. Chen and S. K. Tripathi, "Multiprocessor priority ceiling based protocols," College Park, MD, USA, Tech. Rep., 1994.

[20] B. B. Brandenburg and J. H. Anderson, "Real-time resource-sharing under clustered scheduling: mutex, reader-writer, and k-exclusion locks," in *EMSOFT*, 2011.

[21] K. Lakshmanan, D. de Niz, and R. Rajkumar, "Mixed-criticality task synchronization in zero-slack scheduling," in *IEEE RTAS*, 2011, pp. 47–56.

[22] G. Blake and T. Mudge, "Duplicating and verifying LogTM with os support in the M5 simulator ABSTRACT."

[23] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: Log-based transactional memory," in *in HPCA*, 2006, pp. 254–265.

[24] J. Manson, J. Baker, A. Cunei, S. Jagannathan, M. Prochazka, B. Xin, and J. Vitek, "Preemptible atomic regions for real-time java," in *In 26th IEEE Real-Time Systems Symposium*, 2005.

[25] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Syst.*, vol. 30, pp. 129–154, 2005.

[26] R. Rajkumar, L. Sha, and J. P. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors," in *Real-Time Systems Symposium*. IEEE, 1988, pp. 259–269.