Investigating Transactional Memory for High Performance Embedded Systems

Christian Piatka[1(✉)], Rico Amslinger[1], Florian Haas[1], Sebastian Weis[2], Sebastian Altmeyer[1], and Theo Ungerer[1]

[1] University of Augsburg, Universitätsstr. 2, 86159 Augsburg, Germany
{piatka,amslinger,haas,altmeyer}@es-augsburg.de,
ungerer@informatik.uni-augsburg.de
[2] TTTech Auto Germany GmbH, Emmy-Noether-Ring 16, 85716 Unterschleißheim, Germany
sebastian.weis@tttech-auto.com

Abstract. We present a Transaction Management Unit (TMU) for Hardware Transactional Memories (HTMs). Our TMU enables three different contention management strategies, which can be applied according to the workload. Additionally, the TMU enables unbounded transactions in terms of size. Our approach tackles two challenges of traditional HTMs: (1) potentially high abort rates, (2) missing support for unbounded transactions. By enhancing a simulator with a transactional memory and our TMU, we demonstrate that our TMU achieves speedups of up to 4.2 and reduces abort rates by a factor of up to 11.6 for some of the STAMP benchmarks.

Keywords: Transactional memory · Contention management · Unbounded transactions · Embedded systems

1 Introduction

To fully utilize multicores, the ability to generate efficient parallel code is essential. Because in-depth parallelization has proven to be very error prone, alternative synchronization mechanisms, such as transactional memories (TM) [12], evolved to be a subject of research.

Implementations of hardware transactional memories were integrated into commercial high performance chips from Intel and IBM. Despite their benefits, current available commercial HTMs (e.g. Intel's TSX) do not meet the high requirements of embedded systems. Commercial HTMs statically implement contention management strategies, which can lead to high abort rates. To meet the high requirements concerning power consumption, embedded systems depend on low abort rates. Another disadvantage of COTS HTMs is that they bound transactions in several ways. The size of a transaction is limited by the capacity and

This project received funding by Deutsche Forschungsgemeinschaft (DFG).

A. Brinkmann et al. (Eds.): ARCS 2020, LNCS 12155, pp. 97–108, 2020.
https://doi.org/10.1007/978-3-030-52794-5_8

associativity of the L1 cache. In addition, transactions are aborted by events like interrupts, which limits their duration. This can negatively impact performance and complicates usability, leading to more programming errors, which is unacceptable for embedded systems due to the increasing demands of computational power and the scarce resources provided.

For our work, we developed two challenges: (1) Lowering abort rates by providing effective contention management. (2) Enabling unbounded transactions in terms of size. We want to achieve these goals by implementing a Transaction Management Unit (TMU). The main contributions of this paper are: (1) The design of a flexible TMU, which enables the user to apply different contention management strategies. (2) A solution to enable unbounded transactions, considering their size.

The rest of this paper is structured as follows: After giving an overview on the state of the art of transactional memories, we will describe our TMU. In the following section, our proposal is evaluated. At the end of this paper, we discuss related work and conclude by summing up our results and describing future work.

2 State of the Art

A transaction is a sequence of instructions that is monitored by the transactional memory system. The beginning and the end of a transaction are usually marked by special instructions. To ensure a correct execution of the program, the transactional memory system has to ensure that every transaction fulfills the following three criteria: (1) Transactions have to be executed atomically, which means that they commit or abort as a whole. (2) Transactions have to run isolated, which means that they do not impact each other. (3) The transactional executions have to be serializable, which means that a sequential execution with a matching output exists.

To fulfill these criteria, the transactional memory system has to ensure that values, which are consumed in a transaction, are not modified by another transaction running in parallel. For this purpose, read and write accesses in a transaction are logged at cache line granularity in a read and write set. A conflict occurs whenever a write access of a transaction tries to manipulate a cache line, which is already added to the read or write set of a competing transaction. A conflict also occurs, when a read access tries to read a cache line already contained in the write set of another transaction. To keep track of read and write accesses inside of transactions, HTMs usually utilize the cache coherence protocol.

If a conflict is detected, it has to be resolved by the HTM, by aborting all but one of the conflicting transactions. This involves setting back all the memory modifications performed during the transactions (rollback). Additionally, the read and write sets have to be cleared and the register files have to be restored. Afterwards, the aborted transactions have to be restarted.

Another source of transactional aborts are physical limits of the hardware, or interrupts. Physical restrictions are usually based on the size or associativity

of the L1 caches, which are used to store the transactional read and write sets. Most HTM systems do not implement any mechanisms to allow the read or write set to overflow the size or associativity of the cache. This limits the transactions in terms of consumed and modified cache lines. Interrupts limit a transaction concerning its duration. Frequent transaction aborts because of physical limits or interrupts can be critical for the performance, since a significant amount of work has to be discarded.

Due to these physical restrictions, a programmer usually has to provide an alternative path of execution utilizing different synchronization mechanisms, which are not affected by physical hardware boundaries. The fallback path is a weak spot for transactional memory usage. It uses alternative synchronization, which can be error prone and reduce performance, depending on the depth of parallelization. Additionally, it takes away one of the main advantages, which is the easy usability, because the fallback path increases the complexity of the parallel code. Providing an efficient alternative would render transactional memory superfluous.

3 Transaction Management Unit

In this section, we first describe the implementation and the hardware setup of our system. Afterwards, we give a short overview of the selection of contention management strategies we implemented. At the end of this section, we explain how our solution is able to support unbounded transactions concerning their size.

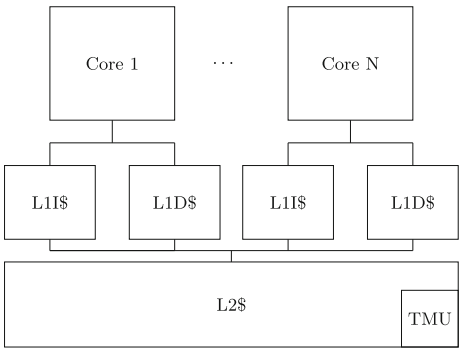

Fig. 1. The multicore system we consider consists of up to 16 cores. Each core has a private L1 instruction cache (L1I$) as well as a private L1 data cache (L1D$). The TMU is integrated into the L2 cache and is able to monitor the messages relevant for the transactional execution.

3.1 Hardware Integration

As depicted in Fig. 1, our TMU is integrated into the shared L2 cache. We consider a multicore system with N cores (we assume $N \leq 16$). The TMU monitors the execution and collects as well as provides data concerning the transactions. To be able to favor a transaction over others, our TMU is able to store priorities. Four ($\log_2 16$) bits are needed at most to be able to save a specific priority for every core. The priority of a transaction can be specified by the programmer at transaction start. The default priorities are the core IDs. In order to store priorities, timestamps, or performance counters, our TMU provides a 64 bit value. The timestamps can be set at different times (e.g. transaction start, transaction commit, etc.) depending on the conditions of the contention management strategy. Performance counters provide information concerning the transactional execution, e.g. the number of committed transactions. To mark whether a transaction runs unbounded, the TMU provides an additional bit per core. The TMU consists of a memory, which stores information, relevant for the transactional execution of each core. The information stored depends on the applied strategy. To resolve a conflict, the TMU takes as inputs (1) the core ID of the core running the transaction, which detected the conflict, and (2) the core IDs of the cores running the conflicting transactions. After comparing the corresponding data, the TMU signals to abort the transaction that detected the conflict, or the conflicting transactions. We optimistically estimate an upper bound for the hardware costs, when considering a 16 core multicore, by:

$$\text{memory} = 2 \times 16 \times 65 \text{ bit} = 260 \text{ byte} \quad (1)$$

$$\text{comparators} = 2 \times 16 \times 65 \text{ bit} = 260 \text{ byte} \quad (2)$$

$$= 520 \text{ byte}$$

Because we rely on a dual ported memory, we doubled the assumed memory capacity in our estimation (Eq. 1). A comparator works similar to an adder, which is the reason why the hardware costs are approximately double the amount of the bits compared (Eq. (2)). Even if we consider the double amount for additional hardware cost, our approach consumes less than 0.05% of the space provided by a 2 MB L2 cache.

3.2 Contention Management Strategies

Whenever a conflict between two running transactions occurs, the responsibility for resolving the conflict is handed over to the TMU. Depending on the strategy, the priority, a timestamp, or the number of commits are stored in the TMU. After comparing the relevant data, the TMU determines which of the conflicting transactions are aborted. We implemented three strategies:

priority: The transaction with the higher priority is allowed to continue. This strategy allows to enforce an ordered commit of the transaction and a prioritization of a transaction over others. We would like to utilize this strategy in the future to enable various real-time strategies (e.g. mixed criticality).

timestamp [16]: The transaction, that started first can carry on. Taking the timestamp of a transaction into account reduces the indeterminism concerning the aborts and guarantees progress.

commit: The transaction on the core, which committed fewer transactions is able to continue. This strategy leads to a more balanced execution, because cores, which were not able to commit transactions, are favored when conflicts occur.

Unbounded transactions always overrule the contention management strategy.

3.3 Unbounded Transactions

Transactions have to abort whenever a transaction's read or write set exceeds the size or associativity of the L1 cache. Therefore, most HTMs have to provide a fallback mechanism consisting of an alternative execution path. This does not only make it harder for the programmer to write efficient and correct code, it can also be crucial for performance because of the loss of already computed work. The TMU monitors the transactional execution and sets a bit whenever a transaction is forced to run in unbounded mode. Whenever the bit is set, conflicts are resolved favoring the unbounded transaction. Therefore, the transaction will never be aborted, which means it does not have to be rolled back. Since it is guaranteed that the transaction will succeed, the backup version of the cache line is not needed, which allows the transaction to use the entire cache hierarchy. The TMU can only support one unbounded transaction at the time. If another transaction or thread tries to perform a conflicting access, the TMU takes actions to suppress them, e.g. by stalling the core.

4 Evaluation

For the implementation of our approach, we utilized the gem5 simulator [4]. We selected the STAMP benchmark suite [6] to evaluate our approach. In this section we will describe in detail our evaluation methodology followed by the presentation of our results.

4.1 Simulation Methodology

The gem5 [4] is a cycle accurate processor simulator. It offers the possibility to choose an instruction set architecture out of a selection such as ARMv7, x86, etc. Furthermore, the periphery can be configured freely. The configuration of our system is described in Table 1. We chose this configuration, as it models a contemporary embedded multicore. High-performance embedded systems as smartphones exhibit similar specifications.

Due to the long run times entailed by the large input set of the STAMP benchmark suite [6] and the authors' recommendation to use the smaller input configuration for simulators, we chose to do our evaluation with the small input configuration depicted in Table 2.

Table 1. System configuration

Num CPUs	{1,2,4,8,16}
Microarchitecture	ARM Cortex-A15
L1 data cache	32 KB
L1 data cache assoc.	8
L2 cache	2 MB
L2 assoc	16
Cache coherence	Directory-based

Table 2. Benchmark configuration

Benchmark	Parameters
bayes	-v32 -r1024 -n2 -p20 -s0 -i2 -e2
genome	-g256 -s16 -n16384
intruder	-a10 -l4 -n2038 -s1
kmeans	-m40 -n40 -t0.05 -i inputs/random2048-d16-c16.txt
labyrinth	-i inputs/random-x32-y32-z3-n96.txt
ssca2	-s13 -i1.0 -u1.0 -l3 -p3
vacation	-n2 -q90 -u98 -r16384 -t4096
yada	-a20 -i inputs/633.2

4.2 Baseline Transactional Memory System

For our baseline, we implemented a transactional memory system into the ARM-based gem5 simulator [4]. The implementation of the interface is similar to those offered by Intel TSX [13] and the newly announced ARM TME [3]. Our interface allows the programmer to explicitly start and end transactions using the corresponding commands, which are provided by our transactional memory system.

Our baseline HTM detects and resolves conflicts eagerly: Conflicts are detected instantly when the conflicting memory access occurs (in contrast to detecting them at commit time). When a conflict occurs, the transaction, that detects the conflict, aborts to resolve it.

We provide a fallback path with regular POSIX Thread synchronization in our baseline implementation. In our baseline as well as in the runs supported by our TMU, a transaction is executed in the fallback path, if the attempt to execute the transaction failed 100 times. Trying to re-execute a transaction for 100 times makes sense, because the execution of a transaction in the fallback path prohibits the other cores to execute work. Whenever the read or write of a transaction in our baseline exceeds the L1 cache, it is directly executed in the fallback path.

4.3 Analysis

We evaluated the STAMP benchmark suite [6]. Figure 2 depicts the evaluation of the eight STAMP benchmarks *bayes*, *genome*, *intruder*, *kmeans*, *labyrinth*, *vacation*, *ssca2* and *yada*. Each graph depicts three lines. All lines show the absolute speedup compared to the reference execution (one core, no synchronization). We focused on the region of interest, which are the parts executed by transactions, because they can be quite small compared to the entire benchmark, making it difficult to show the effects of our work. We calculated the speedup as shown by Eq. 3.

$$\text{speedup} = \frac{\text{reference execution time}}{\text{examined execution time}} \tag{3}$$

The labeling of the lines in Fig. 2 indicates whether the line represents the baseline execution or a contention management strategy combined with unbounded transactions.

In the following, we describe in detail the behavior of the evaluated benchmarks:

bayes: For the benchmark *bayes*, we were able to beat the baseline execution for most executions. Up to 62 unbounded transactions are executed, which shows that it is beneficial to implement unbounded transactions concerning their size. We are able to achieve the best speedup for the execution with four cores and the timestamp strategy. Considering the entire run time of the benchmark, the part in which transactions were executed, is extremely short.

genome: The benchmark *genome* scales quite well. Our system produces the same results as the baseline. The features of the TMU become relevant for the executions with eight and sixteen cores. For these runs, we are able to significantly lower the number of aborted transactions. In these executions, transactions are started, which do not fit in the L1 cache and therefore have to be executed in the fallback path. Because of our TMU, we are prepared for these cases and are able to continue without having to abort them.

intruder: For the benchmark *intruder*, we made similar observations as with the benchmark *genome*. The main difference is that the positive effects of our extensions only take effect for the execution with sixteen cores. For this execution, the baseline contention management strategy performs so poorly, that we are able to lower the abort rate by a factor of 11.6. Within the execution of the benchmark, no transaction faces a capacity problem, which means we achieve the speedup only through better contention management.

kmeans: For the benchmark *kmeans*, we were not able to outperform the baseline. The reason for this is that no transaction, for any execution, faces a capacity problem, which means no unbounded transaction has to be executed. Additionally, the benchmark has hardly any conflicts, which eliminates the grounds of what we can improve.

labyrinth: The baseline execution for the benchmark *labyrinth* is below one, because the benchmark launches fairly big transactions, which do not fit in the L1 cache and cause the transaction to abort and execute in the fallback path. The already achieved computational progress is discarded, which is

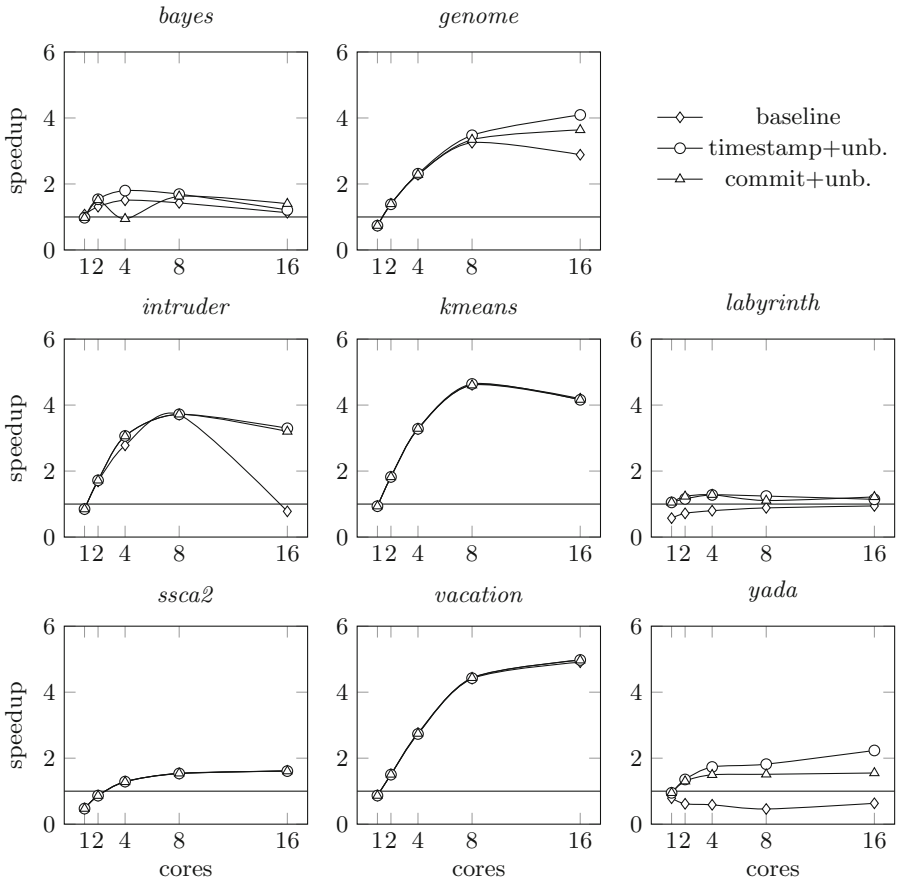

Fig. 2. Results of the execution of the STAMP benchmark suite [6]. For the benchmarks genome, intruder and yada we improved performance compared to the baseline implementation.

why the baseline execution falls below one. For this benchmark, our work is beneficial. We manage to raise the speedup above the baseline execution and one.

ssca2: The benchmark *ssca2* behaves similar to the benchmark *kmeans*. Hardly any of the almost 50000 committed transaction aborts. None of the transactions faces an issue with the capacity of the L1 cache.

vacation: The observations, which can be made for the benchmark *vacation*, are similar to those of the benchmarks *kmeans* and *ssca2*. Of the 4097 committed transactions, a maximum of only about 490 transactions aborts. Additionally, all of the transactions fit into the L1 cache, which is why no unbounded transactions are needed.

yada: The baseline execution for the benchmark *yada* suffers from a lot of conflicts, due to poor contention management. To commit around 4900 transactions, up to 33590 aborts occur. Additionally, some transactions face a problem with the capacity of the L1 cache. Therefore, the TMU handles up to 170 unbounded transactions, which is beneficial to performance and allows us to improve the baseline execution with both strategies. Additionally, we are able to achieve speedups bigger than one.

In Fig. 3, we depicted the number of aborts for every benchmark of the STAMP benchmark suite [6]. The line labeled as baseline depicts the baseline execution. The other lines represent an execution with a contention management strategy. Because we want to show the benefits of the implemented contention management strategies, we disabled the unbounded transactions for this evaluation.

For most of the benchmarks, we are able to lower the number of the aborts. Especially for executions with 8 and 16 cores. For these executions, the contention management strategy of the baseline performs poorly and we are able to reduce the number of aborts significantly for some benchmarks.

For the benchmarks, which already had a low abort rate, our strategies were not beneficial and sometimes even caused more aborts (e.g. *ssca2*). Our strategies perform best, when the contention between the transactions is high.

Our evaluation showed that our work is beneficial to a transactional memory system in terms of performance and abort rates.

5 Related Work

There are some proposals describing how to handle unbounded transactions e.g. [2,7,8,14]. In this section we describe and discuss solutions for similar problems.

The authors of [5] focused on unbounded transactions. They proposed a permissions-only cache, which allows large transactions by only tracking read and write bits without the corresponding data. Once the permissions-only cache overflows, one of two proposed implementations, to handle unbounded transactions, can be utilized. ONE-TM-Serialized only allows one overflowed transaction at a time and stalls the other cores. ONE-TM-Concurrent allows several concurrent transactions to run in parallel, although only one transaction can run in overflowed-mode. Our approach possesses the same runtime characteristics as ONE-TM-Concurrent, the difference between the approaches lies in the management of the unbounded transaction. In contrast to [5], where the authors propose to use per block meta data to ensure the correct execution of the unbounded transaction, we use our TMU to manage and protect the unbounded transaction. Due to a LogTM-style [14] baseline transactional memory system, the authors' proposal is also able to survive interrupting actions performed by the operating system. Contention management strategies were no subject to the authors.

The authors of [9] adapted an HTM for embedded systems, focusing on energy consumption and complexity. In this proposal, the authors evaluate different cache structures and three different contention management schemes (eager, lazy,

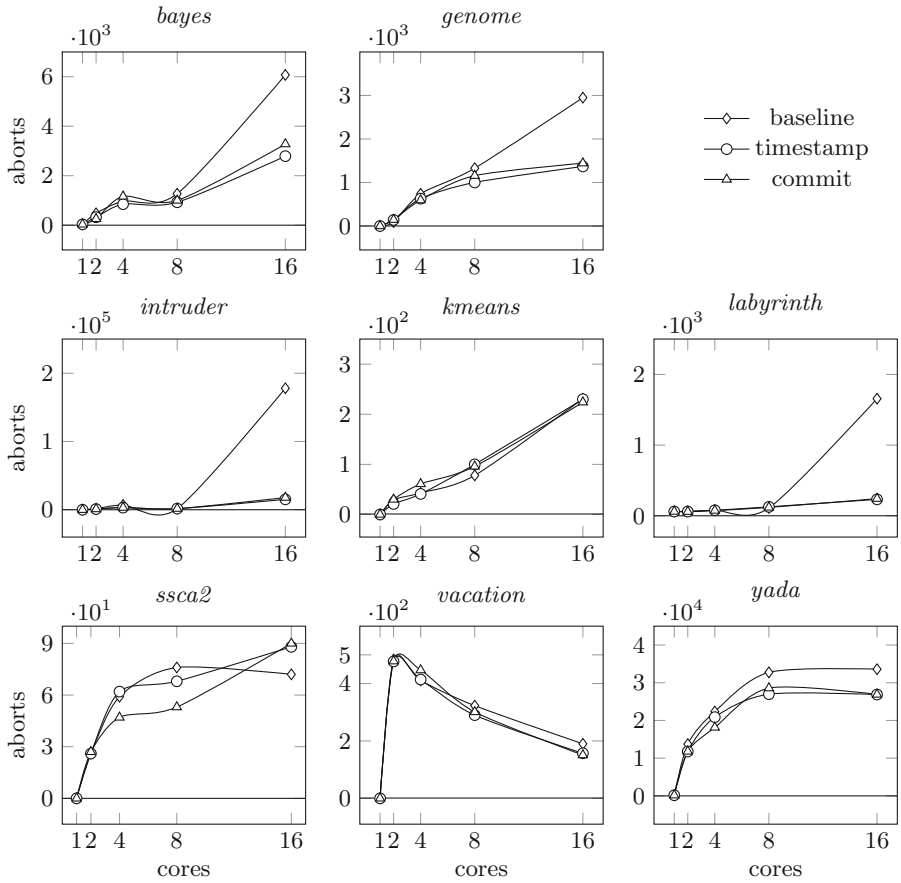

Fig. 3. We were able to reduce the number of aborts for most of the benchmarks. Especially the benchmarks, for which a lot of aborts were saved (*genome*, *intruder*, etc.), achieved significant speedups, which can be observed in Fig. 2.

forced-serial), concerning their complexity and energy consumption. The authors also provide a mechanism to support overflowing transactions (exceeding cache limits) by running them in serial mode. Because the authors are very sensitive for complexity, they only allow a simple execution mode for unbounded transactions. Therefore, running a transaction in serial mode means that all other CPUs get suspended. The overflowed transaction can now run isolated and is able to utilize the complete memory hierarchy. Our approach differs from [9], because we focus on performance and abort rates. Therefore, we provide a more complex execution mode for unbounded transactions. Furthermore, we offer several different and more complex contention-management strategies.

The work describing the most relevant contention management policies focus on software transactional memories (STM) [10,11,16,17]. Later work has applied

some of these contention management strategies to HTMs e.g. [15]. In contrast to our work, the authors focused on using HTM as an synchronization primitive for an operating system as well as managing it in the scheduler. This made it necessary to implement a new more complex HTM. The authors focused, concerning the contention management strategies, on finding a well performing policy in most of the cases. This makes sense, since the best working policy is workload dependent, as also mentioned by the authors.

6 Conclusion and Future Work

In our work, we present a TMU, which is located in the shared L2 cache and costs approximately less than 0.05% of the space of a 2 MB L2 cache. We provide three different contention resolution policies and enable unbounded transactions. In our evaluation, we did not consider the contention management strategy, which enables priorities, because in our perspective it would not produce any interesting results concerning its execution time. The priority strategy will be of more focus in our future work. By our evaluation with the gem5 simulator [4] and the STAMP benchmark suite [6], we show that the TMU is beneficial for performance and is able to reduce the number of aborted transactions.

The work we present in this paper is the foundation to employ several other features. To further increase performance, we also would like to enable thread-level speculation, which will utilize the TMU to ensure correct execution. Our research also concerns fault tolerance utilizing transactional memory [1], where we will also consider investigating the use of the TMU. Because safety is a major issue for embedded systems, we want to try to utilize our TMU to enable mixed criticality and real time for hardware transactional memories.

References

1. Amslinger, R., Weis, S., Piatka, C., Haas, F., Ungerer, T.: Redundant execution on heterogeneous multi-cores utilizing transactional memory. In: Berekovic, M., Buchty, R., Hamann, H., Koch, D., Pionteck, T. (eds.) ARCS 2018. LNCS, vol. 10793, pp. 155–167. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77610-1_12
2. Ananian, C.S., Asanovic, K., Kuszmaul, B.C., Leiserson, C.E., Lie, S.: Unbounded transactional memory. In: 11th International Symposium on High-Performance Computer Architecture, pp. 316–327, February 2005. https://doi.org/10.1109/HPCA.2005.41
3. ARM Ltd.: Transactional memory extension (TME) intrinsics. https://developer.arm.com/docs/101028/0009/transactional-memory-extension-tme-intrinsics. Accessed 13 Jan 2020
4. Binkert, N., et al.: The gem5 simulator. SIGARCH Comput. Archit. News **39**(2), 1–7 (2011). https://doi.org/10.1145/2024716.2024718
5. Blundell, C., Devietti, J., Lewis, E.C., Martin, M.M.K.: Making the fast case common and the uncommon case simple in unbounded transactional memory. SIGARCH Comput. Archit. News **35**(2), 24–34 (2007). https://doi.org/10.1145/1273440.1250667

6. Minh, C.C., Chung, J.W., Kozyrakis, C., Olukotun, K.: STAMP: stanford transactional applications for multi-processing. In: 2008 IEEE International Symposium on Workload Characterization, pp. 35–46, September 2008. https://doi.org/10.1109/IISWC.2008.4636089
7. Chuang, W., et al.: Unbounded page-based transactional memory. SIGARCH Comput. Archit. News **34**(5), 347–358 (2006). https://doi.org/10.1145/1168919.1168901
8. Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D.: Hybrid transactional memory. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 336–346 (2006). https://doi.org/10.1145/1168857.1168900
9. Ferri, C., Wood, S., Moreshet, T., Bahar, R.I., Herlihy, M.: Embedded-TM: energy and complexity-effective hardware transactional memory for embedded multicore systems. J. Parallel Distrib. Comput. **70**(10), 1042–1052 (2010). https://doi.org/10.1016/j.jpdc.2010.02.003
10. Guerraoui, R., Herlihy, M., Pochon, B.: Polymorphic contention management. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 303–323. Springer, Heidelberg (2005). https://doi.org/10.1007/11561927_23
11. Guerraoui, R., Herlihy, M., Pochon, B.: Toward a theory of transactional contention managers. In: Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing, pp. 258–264 (2005). https://doi.org/10.1145/1073814.1073863
12. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture, pp. 289–300 (1993). https://doi.org/10.1145/165123.165164
13. Intel Corporation: Intel Transactional Synchronization Extensions (Intel TSX) Overview. https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-intel-transactional-synchronization-extensions-intel-tsx-overview. Accessed 23 Jan 2020
14. Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., Wood, D.A.: LogTM: log-based transactional memory. In: 2006 The Twelfth International Symposium on High-Performance Computer Architecture, pp. 254–265 (2006). https://doi.org/10.1109/HPCA.2006.1598134
15. Rossbach, C.J., Hofmann, O.S., Porter, D.E., Ramadan, H.E., Aditya, B., Witchel, E.: TxLinux: using and managing hardware transactional memory in an operating system. In: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, pp. 87–102 (2007). https://doi.org/10.1145/1294261.1294271
16. Scherer, W.N., Scott, M.L.: Contention management in dynamic software transactional memory. In: PODC Workshop on Concurrency and Synchronization in Java Programs, pp. 70–79 (2004)
17. Scherer, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing, pp. 240–248 (2005). https://doi.org/10.1145/1073814.1073861