Hardware Multiversioning for Fail-Operational Multithreaded Applications

Rico Amslinger
University of Augsburg
Augsburg, Germany
amslinger@es-augsburg.de

Christian Piatka
University of Augsburg
Augsburg, Germany
piatka@es-augsburg.de

Florian Haas
University of Augsburg
Augsburg, Germany
haas@es-augsburg.de

Sebastian Weis
TTTech Auto Germany GmbH
Unterschleißheim, Germany
sebastian.weis@tttech-auto.com

Theo Ungerer
University of Augsburg
Augsburg, Germany
ungerer@informatik.uni-augsburg.de

Sebastian Altmeyer
University of Augsburg
Augsburg, Germany
altmeyer@es-augsburg.de

***Abstract*—Modern safety-critical embedded applications like autonomous driving need to be fail-operational. At the same time, high performance and low power consumption are demanded. A common way to achieve this is the use of heterogeneous multi-cores. When applied to such systems, prevalent fault tolerance mechanisms suffer from some disadvantages: Some (e.g. triple modular redundancy) require a substantial amount of duplication, resulting in high hardware costs and power consumption. Others (e.g. lockstep) require supplementary checkpointing mechanisms to recover from errors. Further approaches (e.g. software-based process-level redundancy) cannot handle the indeterminism introduced by multithreaded execution.**

This paper presents a novel approach for fail-operational systems using hardware transactional memory, which can also be used for embedded systems running heterogeneous multi-cores. Each thread is automatically split into transactions, which then execute redundantly. The hardware transactional memory is extended to support multiple versions, which allows the reproduction of atomic operations and recovery in case of an error. In our FPGA-based evaluation, we executed the PARSEC benchmark suite with fault tolerance on 12 cores.

***Index Terms*—fault tolerance, redundancy, hardware transactional memory, multiversioning, multi-core**

I. Introduction

Embedded applications have a multitude of requirements for their execution environment. Safety-critical applications like fully autonomous cars or fly-by-wire electronic flight controls are required to be fail-operational, as a failure could directly endanger human lives. If an error occurs, detection and recovery need to be quick, as deadlines still need to be met. At the same time, autonomous cars or advanced terrain awareness and warning systems require high performance for purposes like image recognition. In addition, embedded systems often run on batteries, which makes a low power consumption essential. Therefore, heterogeneous multi-cores, which consist of fast cores and energy efficient cores executing the same instruction set, are considered the best option.

It is hard to implement such a high performance, energy efficient and fail-operational execution with state of the art fault tolerance mechanisms. Dual modular lockstep execution is a widespread mechanism, which can be found in many off-the-shelf CPUs like the ARM Cortex-R series [1] or some Infineon Aurix CPUs [2]. However, lockstep execution fails to properly fulfill the requirements of embedded systems, as it can only detect errors, but has to rely on alternative mechanisms like checkpointing to recover from them [3]. These recovery mechanisms are often not implemented in hardware and thus result in high overheads, even for an error-free execution, and might not be fault-tolerant themselves. In addition, they are often only performed infrequently in order to limit the overhead, which results in a large loss of progress when recovering. Multi-core lockstep systems, although there are some available, are still a kind of rarity.

Triple modular redundancy is an alternative fault tolerance approach, which is often used in the aerospace industry. While it solves the error recovery problem of lockstep execution, it introduces new issues. Needing three instances of the CPU increases power consumption and production costs in significant ways. Further, the inherent indeterminism of multi-threaded applications leads to divergent states in the redundant processors, which renders such systems unsuitable for parallel applications [4].

If an appropriate hardware-based redundancy mechanism is unavailable, developers often rely on software-based fault tolerance. Apparently, such systems exhibit more vulnerable parts, since only the redundant application is within the sphere of replication, where it is protected from errors. The mechanisms for error detection and recovery are inevitably susceptible to errors, which could render the system inoperable or lead to data loss [3]. Additionally, software-based fault tolerance often suffers from a high performance overhead and only a few implementations can handle multithreaded execution. Error detection latencies can also be a problem, as many implementations trade high error detection latencies for improved performance.

We present a novel approach for fault tolerance on embedded systems based on multiversioning to mitigate those

This project received funding by Deutsche Forschungsgemeinschaft (DFG).

Cite as: R. Amslinger, C. Piatka, F. Haas, S. Weis, T. Ungerer and S. Altmeyer, "Hardware Multiversioning for Fail-Operational Multithreaded Applications," 2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Porto, Portugal, 2020, pp. 20-27, doi: 10.1109/SBAC-PAD49847.2020.00014.

disadvantages. Both homogeneous and heterogeneous multi-cores are supported. In our approach, the application is only executed twice in order to keep the overhead minimal. Hardware checksum calculation ensures that every single-bit error is caught. Error detection is fast, as transactions offer a quick validation interval. A transactional memory based rollback mechanism allows for cheap recovery after an error is detected. The system also offers conflict detection, which makes the execution of multithreaded transactional memory applications possible. Consistency between the redundant executions is ensured by keeping multiple versions of each data word. A transactional memory based pthreads implementation ensures backward compatibility for classic multithreaded applications, which rely on atomic operations and cache coherence.

Altogether, our approach has the following advantages:

- The system is fail-operational, as it can recover from errors.
- Homogeneous and heterogeneous multi-cores are supported.
- Shared memory multithreaded applications can be executed redundantly.
- Transactional memory can be used for synchronization.

Our work is structured as follows. First, we present related work. In Section III, our redundancy concept is explained. In the next section, we explain the extension to multithreaded execution using multiversioning. In Section V, we present the runtime overhead and error detection latency evaluation of our approach. This section contains an explanation of the methodology and an analysis of the results. Ultimately, the paper is completed with a conclusion and an outlook on future work.

II. Related Work

Process-Level Redundancy [5] is a software-based approach to provide fault tolerance for singlethreaded applications. The approach replicates the process multiple times. The processes are synchronized at every system call and the parameter values are compared. In order to recover after an error, three instances are required. The evaluation assumes that a sufficient amount of free cores is available for the redundant processes. In addition, we expect the error detection latency to be unsuitable for embedded systems, as system calls can be far apart.

In our previous work [6], we have already presented a hardware fault tolerance mechanism for singlethreaded applications, which is the foundation of this approach. However, multiversioning, which is essential to realize multithreaded execution in the current paper, was not used in the previous paper. Previously, we used the simulator gem5, which required very long evaluation times. However, to be able to execute larger benchmarks, we shifted to an FPGA implementation for this paper. Some optimizations in the previous paper could also be advantageous for multithreaded workloads, but could not be realized, as the closed-source cores, which were used in the FPGA implementation, prevent the necessary changes. It also evaluates the power consumption of the singlethreaded approach on heterogeneous cores. We expect the multithreaded approach to behave similarly.

In another previous work [7], we have described a software-based fault tolerance approach for utilizing Intel TSX. Major parts of the approach are required to work around the limitations of Intel TSX. It is necessary to start two separate processes, as it is not possible to share the same memory for leading and trailing threads. Intel TSX does not support multiversioning, either. There are overheads due to instrumentation, splitting the execution into transactions, checksum calculation and transfer.

FaulTM-multi [8] is a hardware fault tolerance implementation utilizing transactional memory. In contrast to our approach, FaulTM-multi is tightly coupled. This results in several restrictions: Firstly, it is not possible to have unequal amounts of original and backup threads, which restricts parallelism. Additionally, the cores, on which the threads run, need to be homogenous to avoid the faster thread blocking at every transaction commit. Unsettled optimizations like those used in [6] cannot be used either, as the original thread cannot run ahead to compensate for any fluctuation.

HAFT [9] is a software fault tolerance implementation utilizing transactional memory. In contrast to our approach, HAFT uses instruction-level redundancy. This makes it well suited for modern high performance out-of-order CPUs, as they can often execute both instructions in a single cycle and correctly predict the comparison branch. However, the approach is less suited for embedded or heterogeneous systems, as those often feature simple in-order CPUs, which cannot overlap the execution of the redundant instructions.

Multiversion concurrency control [10] is a method used by databases to manage concurrent accesses. This technique is used by PostgreSQL, for example [11]. We suppose that hardware multiversioning support, like the one provided by our approach, can be used to accelerate such applications.

III. Redundancy

We assume a standard shared memory hierarchy (Fig. 1) consisting of multiple (potentially heterogeneous) cores. The

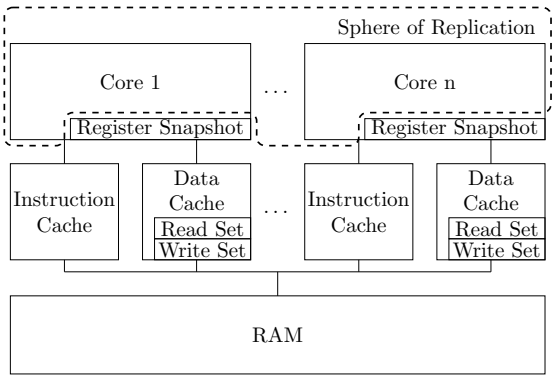

Fig. 1. This figure shows our memory hierarchy and the sphere of replication. The cores and data caches are extended to support transactional memory and multiversioning. The sphere of replication covers the pipeline in the cores. Other components are protected by ECC.

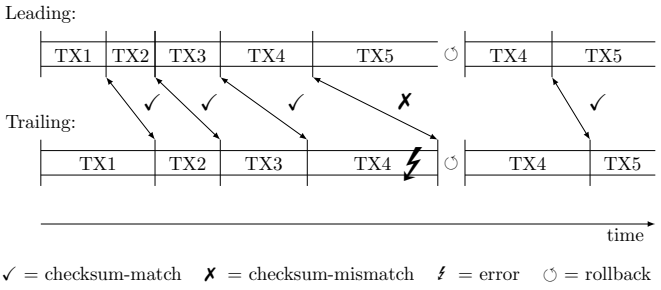

Fig. 2. This figure shows an exemplary redundant execution. The singlethreaded application is split into transactions TXi, which are executed on a leading core and a trailing core. For some time the checksums match, but after TX4 a bitflip causes a mismatch. This results in a rollback and a restart of TX4.

cores are connected to coherent private instruction and data caches. Optionally more (potentially shared) cache levels might follow. Finally, all cores can access a common main memory.

In order to implement multiversioning, we have extended the memory hierarchy in several spots. The cores themselves need to be extended with register snapshots to enable rollback. The data caches store the read and write set. They also implement the largest part of the logic necessary to handle version selection and communication. If the code is not self-modifying, the instruction caches can be left unchanged.

Our goal is to provide fault tolerance for the pipelines of the cores. We assume that the memory hierarchy is already protected by means of ECC or similar mechanisms. For reliable recovery, it is also necessary that the register snapshots are protected from faults. If the architecture completely copies the register set, ECC can be used here, too.

Our approach implements a leading/trailing execution concept similar to the one we have used in [6]. The program is first executed on (a) leading core(s). Their results are then validated by (a) trailing core(s), which execute the same code. It is possible to either use a homogeneous or heterogeneous multi-core for the redundant execution.

To realize this, the execution is split into transactions (TXi in Fig. 2). Contrary to regular transactions, those automatic transactions commit by themselves after a given time limit. The next transaction starts immediately afterwards. However, manual transactions for concurrency control, whose bounds are set by the programmer, can also be used if needed.

The most difficult aspect in implementing automatic transactions is determining the bounds. However, our transactional memory system provides multiple features, which allow for an easy implementation: If a cache line is evicted, conflict detection is still possible, as transaction meta data can be evicted to memory. Additionally, all instructions, which are necessary for regular execution, can be issued in transactions. Therefore, it can be guaranteed that every transaction, which abides to certain limits concerning runtime (in instructions) and memory operations, will eventually commit. As those limits are independent of the actual instructions and memory addresses, they can easily be monitored using simple counters. Register backup does not require a specific instruction, either, which makes it possible to start transactions at any time. Therefore, the bounds can be easily determined with low hardware costs.

An error cannot propagate to the other core, as both cores can only see their modifications to the memory. In the singlethreaded case, the first transaction can be started simultaneously for the leading and the trailing thread. If the leading thread is executed on a faster core, it will finish first. It can then already start the next transaction, as long as it has sufficient speculative resources.

While the transaction is running, a checksum of every instruction outcome is calculated. This checksum is then compared after both transactions have completed. If the checksums match, execution can continue regularly and the checkpoints at the beginning of the transactions are deleted. If the checksums do not match (after TX4 in Fig. 2), both cores need to roll back to the beginning of their transactions. This means, the leading core might have to roll back multiple transactions at once. After the rollback, both cores restart their transactions. If the fault was transient, it should not occur again and the checksums should match after both transactions have been repeated.

IV. Multiversioning

When executing multithreaded applications with fault tolerance, multiple complications occur. It is no longer sufficient to just have a single leading thread and a single trailing thread, as the transactions of all threads need to be validated. As all threads in the process use the same address space and transactions already save the register set, it is possible for a trailing thread to quickly switch between validating the transactions of different leading threads. However, it is still preferred to keep leading and trailing cores associated when possible, as this will benefit cache locality. This feature has the additional advantage that I/O operations can be easily realized. The leading core commits its transaction and leaves redundant mode to perform the I/O operation alone. After the I/O operation is finished, resynchronization between leading and trailing cores happens automatically, as the register set is transferred anyway and the multiversioning takes care of the memory content.

The required ratio of leading and trailing cores depends on the underlying system and the application. For homogeneous systems, it is usually close to one. However, for a heterogeneous system a different ratio can be chosen to accommodate slower cores. If an application has plenty of waiting time, the amount of trailing cores can be lowered, as it is useless to wait on the leading and the trailing. The same can be done if the platform has means to accelerate the trailing cores like perfect prefetching or forwarding branch outcomes [6]. The trailing cores will simply switch between validating those leading cores, which are currently not waiting.

When regular transactions are used, the order of the transactions must be preserved between the leading and trailing ex-

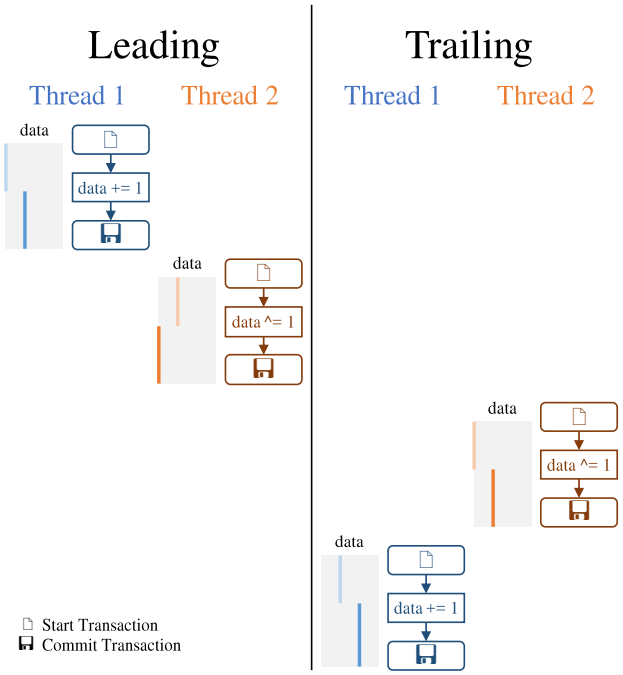

Fig. 3. This figure shows an incorrect execution, which can occur if the order of the transactions is not preserved between leading and trailing. Thread 1 wants to execute a transaction which increments data by 1. Thread 2 wants to execute a transaction which xors data by 1. If thread 1 is executed first (leading case in this figure), the final result is 0. On the other hand if thread 2 is executed first (trailing case in this figure), the final result is 2. This causes a mismatch in checksums and thus a rollback.

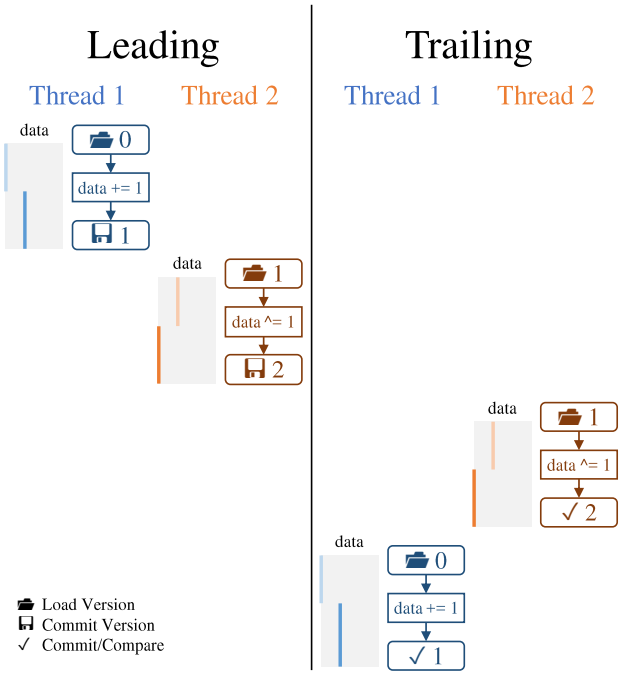

Fig. 4. This figure shows the same application as in Fig. 3, but this time executed with multiversioning. From the perspective of the software, the execution on the leading core behaves the same. The result after the first transaction is stored in version 1 and the result after the second transaction in version 2. Even if the thread 2 is executed first in the trailing execution, it still loads version 1 as base. This is possible, as the version is still retained from the leading execution. Thus the final result is also 0. Version 1 is validated later and the speculative resources are dropped, as version 2 is already available.

ecution to avoid unnecessary and potentially infinite rollbacks (see Fig. 3). If this situation emerges often, which is to be expected on high core count out-of-order CPUs, performance will be poor, as rollbacks are expensive due to the work that needs to be executed again. It is also necessary to have consistent rollback checkpoints between all threads. It is not sufficient to roll back only the transaction in which the error was detected, as another transaction might have already read data written by that transaction.

In our approach, the fault-tolerant execution occurs on a system supporting multiple versions of the same memory word. This solves the indeterminism problem, as can be seen in Fig. 4. The transaction, in which the load is executed, defines which version of a word is read. Every word has a safe version, which is used for rollback if an error occurs and is only updated if the trailing cores have validated the new value and all previous transactions. Additional versions are used by the leading cores to store speculative values. As soon as the transaction has committed, those will be made visible to the other leading cores. The trailing cores generate versions, too. These are only used by the same core to satisfy reads after writes in the same transaction. Other cores will never see those values and they are dropped after the transaction is completed. If a conflict is detected between two leading transactions, they behave like regular transactions (i.e. modifications are not visible to other transactions before commit and in case of a conflict one is aborted and repeated).

Therefore, it is possible to execute shared memory multi-threaded applications on our system. The preferred synchronization approach is to use transactions. However, atomic operations are also supported. All randomness like different execution orders or pseudo random number generator seeds are synchronized between leading and trailing cores. Cache coherency is also maintained.

After a commit, the leading core continues with the next leading transaction. In addition, a new transaction on a trailing core is started with the same base version, executing the same code. Its results are invisible for all other cores and dropped after the trailing core commits. Once this transaction commits, the checksum is compared to its leading counterpart. If they and all previous checksums match, the leading version is merged with the safe version and all speculative data belonging to those transactions is freed.

If a mismatch is detected, a global rollback is initiated. To perform the rollback, all currently running transactions on the leading and trailing cores are aborted. It is then determined, which transaction is the last consecutively confirmed. The memory is restored to the version that was produced by the commit of this transaction. The register set of the corresponding leading core is reset to the commit of that transaction. All other leading cores are reset to their last previous commit. The trailing cores require no explicit reset, as they will receive a new register set from the leading cores.

As the speculative values within transactions are invisible for other cores, adjustments to synchronization constructs are necessary. For the most part, these only happen at the library level. Changes to the application source code are rarely necessary. Atomic operations can be replaced by a small transaction, which only encapsulates the memory accesses and corresponding operations, which should be executed atomically. For example, an atomic increment is replaced by a transaction containing a regular load, the add instruction and a regular store. As transactions guarantee atomicity, the semantic of the operation does not change. By committing the transaction right away, the modified value becomes visible to other cores immediately.

Further optimizations can be used to improve performance. We have replaced all operations in our implementation of the pthreads library to use native transactions instead of atomic operations. For example, we have replaced the atomic swap in the routine to lock a mutex with a transaction, which aborts if the mutex is already locked. This reduces the amount of committing transactions significantly in comparison to simply replacing the atomic operation. Thus, the memory load is reduced. It is not necessary for the trailing cores to validate aborted transactions, as they have no influence on the system state, allowing them to catch up if they fell behind.

One general issue, which affects many transactional memory applications to varying degrees, is false sharing. False sharing occurs when two threads access different words in the same cache line without synchronization. This is also a minor issue for systems without transactional memory, as it causes cache line bouncing. This issue mainly occurs, because applications access an array with their thread id as an index. It can also occur if different data structures share the same cache line by chance.

Our system provides several optimizations to minimize this issue for automatic transactions: If both threads only access the cache line for reading, the cache line is put into shared state and no transaction is aborted. If both threads access the cache line for writing, the first transaction will try to commit prematurely. This costs some performance, but is better than aborting. The same happens if the first access is a write and the second a read. If the first access is a read and the second is a write, the conflict is silently ignored. Note that this can cause a checksum mismatch and rollback if the conflict was actually true (same word in the same cache line).

V. Evaluation

A. Methodology

We implemented our approach on the Xilinx Virtex UltraScale+ FPGA VCU118 Evaluation Kit. This board features the XCVU9P FPGA and two 4 GB DDR4 memories. A USB port is available for JTAG and UART. The other components on the board were not used for our evaluation.

Our design features 12 MicroBlaze cores with support for single precision floating point operations. The cores are connected to coherent private data caches and instruction caches (each 16 kB, 4x set associative). The caches are interconnected to both memory controllers and an UART module. Our extensions are implemented in the caches and addressed by a memory-mapped interface. Registers are backed up using the trace port. Thus, no changes to the closed-source MicroBlaze cores were necessary. The design runs at 50 MHz. The main limiting factors for the clock rate are the performance counters and assertions.

We used the PARSEC Benchmark Suite [12] for the evaluation. Note that this is not a throughput evaluation with multiple singlethreaded processes, but a runtime evaluation, where the benchmarks are run with multiple synchronized threads. To support the benchmark execution on the MicroBlaze, we had to port the pthreads library. As the benchmark *freqmine* does not support pthreads execution, it is missing from the evaluation. *fluidanimate* and *facesim* are also missing, as they do not support arbitrary thread counts. In particular, 12 threads are not supported. Some other modifications were necessary to compile the benchmarks, as the MicroBlaze compiler does not support some old constructs and reserves additional keywords. We avoided performance affecting changes as good as possible.

The benchmarks were run in the *simmedium* configuration. The limiting factors for input set size are the slow transfer of the input files via JTAG and the large number of different configurations. The benchmarks were executed entirely, but to measure the execution times only the region of interest was considered. As we only have a single FPGA board available and due to time constraints, we have executed each configuration only once. As there is no operating system and the per benchmark runtime is rather long (1 minute for *streamcluster* with 12 threads to 2 hours for *swaptions* with 1 thread), the variance is quite low. To validate the correct execution, the outputs were copied back to the host machine and compared to x86 executions with the same thread count. We had to add additional outputs to the benchmark *raytrace*, as it throws its results away.

B. Execution Time Overhead

We have analyzed the runtime and scalability of our approach. We expect the execution time of a dual modular redundant approach to be between the runtime of the non-redundant variant with half the core count and twice the runtime of the non-redundant variant with the same core count.

The execution with half the core count forms a lower bound, as our approach executes the benchmark twice (once on the leading cores and once on the trailing cores). This estimation is however overly optimistic, as our approach requires continuous communication. Issues like false sharing, which reduce the scaling of the non-redundant multithreaded application, also affect our approach negatively. In theory, it is possible for our approach to outperform this bound, if the application blocks frequently due to synchronization, but still scales very well. However, we consider this kind of application as purely academically and do not expect it to occur in practice.

Executing the application twice one after another forms an upper bound for the runtime. A fault tolerance approach should

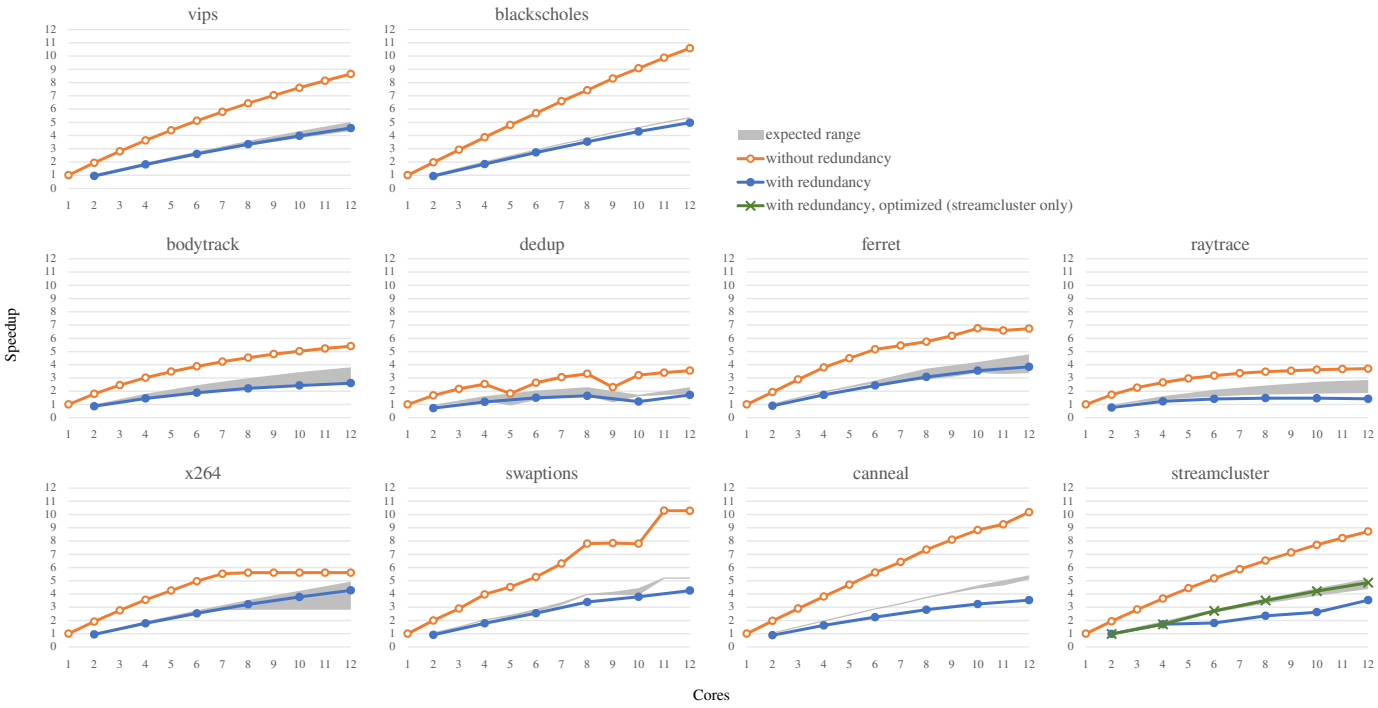

Fig. 5. This chart shows the speedup of the various PARSEC benchmarks in the different configurations at a certain core count. All speedups are relative to the singlethreaded execution without redundancy and consider only the region of interest. For the variant without redundancy, the benchmarks were launched with the core count as thread count parameter. For the variant with redundancy, half the core count (i.e. the leading core count) was used as parameter. The lower bound of the expected range is the execution without redundancy repeated twice. The upper bound of the expected range is the execution without redundancy executed twice in parallel with half the core count. Note that especially at low core counts the expected range is hidden behind the line for redundancy for some benchmarks, as the runtimes are so close together.

be able to outperform this bound in order to best the naive "execute it twice and compare the results" software fault tolerance approach. Notice that this naive approach does however suffer from two major disadvantages: The error detection latency is very long, as it lasts from the first instruction of the first run to the comparison after the execution of the second run. In addition, it is not possible to easily implement recovery, as executing the program a third time takes a long time and the initial state might not be available anymore.

For 12 cores, the geometric mean of the slowdown comparing the redundant variant to the non-redundant baseline is 2.11. Below, we describe the results shown in Fig. 5 in detail.

The benchmark *vips* behaves as expected. The execution without redundancy follows Amdahl's Law. The execution time with redundancy is in the range, in which one would expect an approach like ours.

The benchmark *blackscholes* is embarrassingly parallel. Some of the used floating point operations are implemented in software on the MicroBlaze, which results in the threads having different runtimes. As there is no work balancing and due to the memory controller acting as a bottleneck, the benchmark does not reach a perfect speedup. There are minor false sharing issues with our approach, as the data is split over multiple arrays with no padding between threads. However, very little global data is written, which reduces the impact of this issue.

The benchmark *bodytrack* uses a thread pool for parallelization. As this thread pool contains as many threads as cores and there is an additional main thread, common context switches are required. In addition, some parts of the application in the region of interest are not parallelized. These two aspects result in a shallow speedup curve. Our approach cannot take advantage of the lack of parallelization, as the sections are too long to cover them with the loose coupling. Thus, the speedup of our approach behaves more similar to the non-redundant variant, which executes one after the other, than the parallel one.

The benchmarks *dedup*, *ferret* and *raytrace* are limited by main memory bandwidth. A large shared l2 cache would most likely improve the performance for both the baseline and our approach. Our approach would profit further from it, because the trailing cores access the same data as the leading cores with some delay. Thus, a sufficiently large shared l2 cache would eliminate the trailing memory access altogether.

The benchmark *x264* stops scaling at high thread counts due to synchronization constructs. As the waiting threads do not consume any resources like memory bandwidth or trailing runtime, our approach does well.

The benchmark *swaptions* shows irregular scaling. This is however unrelated to the platform or the approach, as it is caused by the small input size. In the *simmedium* configuration, 32 work items are partitioned between the threads. This works out well for e.g. 4, 8 and 11, but poorly for e.g. 9, 10 and 12, which results in the steps in the speedup graph.

The benchmark *canneal* mostly uses a custom synchronization mechanism. Pointers are accessed automatically and can contain a special value to signal that the object is locked for writing. If the object is locked, busy waiting is performed. This synchronization approach does not scale well with multiversioning, as the many explicit atomic operations result in many small transactions. In addition, the system cannot detect when a thread is waiting, which means the trailing cores waste much time to confirm waiting loops. To optimize such an application for our system, one would need to replace the atomic operations by transactions. In this case, this would be easily possible, as the main operation is a simple swap, which easily fits in a single transaction, eliminating the need for locking altogether. Note however that regular transactional memory optimization rules still apply (i.e. those transactions would most likely be too small).

The benchmark *streamcluster* uses many barriers. Sometimes barriers follow directly after each other with no code between them. Barriers are problematic for our approach, as they can force the leading cores to wait for the trailing cores to catch up, if just one thread accesses a cache line, whose available versions have been used up. This cache line can even be the cache line containing the barrier. If the code executed between the barriers is too short, the transaction will not reach its intended length, meaning that even more versions are consumed. To optimize the benchmark one should try to reduce the number of barriers needed. For our approach, it is better to execute short serial work (like adding the result of all threads) redundantly on all threads instead of just one, as this benchmark does, to remove additional barriers.

It can also be seen that the benchmark prefers even thread counts in the multiversioning variant. If all threads try to write to the same cache line at the same time, the available speculative versions (two in this implementation) for this cache line will run out quickly. Further threads then have to wait until those versions get confirmed by the corresponding trailing transactions. As this benchmark makes heavy use of barriers, threads will always reach such code sections at the same time, which means it will be completed in batches of two. Thus, an odd thread count will result in another batch, which contains only one thread. Writing such code should be avoided, as there will also be some serialization, when executed without redundancy, due to the cache line bouncing between the cores. However, a cache miss is significantly cheaper than a trailing transaction, which makes the effect less prevalent for the baseline.

The benchmark *streamcluster* is also a prime example for the impact of false sharing. The source code contains a constant called `CACHE_LINE`, which controls the padding between the memory regions of the different threads. It is initially set to 32. However, the cache line size, which is used in our platform, is 64. Changing this value accelerates the application by 37.5 %.

Some benchmarks suffer from race conditions. This causes problems, when those benchmarks are executed redundantly, as it can no longer be guaranteed that the trailing core reads the same data as the leading core. The mismatch is then detected as a transient fault and all cores are rolled back in order to retry. Depending on the frequency of the race condition, it can immediately occur again.

The optimal solution would be to fix the source code, as those race conditions can result in wrong results even on other architectures. If this is too difficult, it is also possible to enable conflict detection not just for explicit but also for automatic transactions. This enables the leading cores to detect the conflict, before it can affect the trailing cores. Thus, only a single leading core will roll back instead of the whole system. However, this does not resolve the race condition itself, meaning the application might still output wrong results. Though, it can reduce the frequency of the race condition, as the transactions produce a coarser interleaving. In addition, enabling the conflict detection for automatic transactions will increase the impact of false sharing.

We observed race conditions in two of the tested benchmarks. Enabling conflict detection for automatic transactions in those benchmarks results in an additional overhead of 4.1 % for *x264* and 10.7 % for *canneal*.

C. Error Detection Latency

We have also analyzed the error detection latency. The average and maximum values are shown in Table I.

The resulting average values clearly reflect the targeted transaction duration of 10,000 cycles. Many automatic transactions hit this target quiet accurately and a trailing core is ready right away to validate the transaction. However, for most benchmarks the average is lower, as synchronization operations explicitly commit the transaction before the time limit is reached. Some automatic transactions are longer than the target. This happens, as we can only commit transactions at memory instructions. We suffer from this constraint, because the MicroBlaze is closed-source. In a real implementation, this would most likely not be an issue. If a benchmark

TABLE I
ERROR DETECTION LATENCY

Benchmark	Average [cycles]	Maximum [cycles]
vips	9,809	186,071
blackscholes	9,954	27,606
bodytrack	9,812	51,326
dedup	8,570	199,024
ferret	9,649	90,396
raytrace	8,588	32,290
x264	10,310	131,229
swaptions	9,817	36,626
canneal	18,075	138,877
streamcluster	8,407	59,100
overall	10,299	199,024

This table shows the average and maximum error detection latency in cycles. The average latency is the average time between every instruction and its corresponding checksum comparison. The maximum latency spans from the first cycle in a leading transaction to the checksum comparison of the corresponding trailing transaction. These values were measured for the whole benchmark with 6 leading cores and 6 trailing cores.

makes heavy use of software floating point, this constraint can have a significant impact, as the gcc software floating point library avoids memory operations whenever possible and some operations take relatively long. For example, the benchmark *canneal* makes heavy use of doubles, which are not hardware accelerated by the 32-bit MicroBlaze.

The worst case error detection latency is significantly higher for most benchmarks, as the speculative nature of transactional memory can result in load spikes on the trailing cores. Those load spikes result in waiting times before a transaction can be validated. Another reason for large error detection latency are threads that switch from one trailing core to another and incur more cache misses than the corresponding leading transaction. Thus, the trailing core takes longer than the planned 10,000 cycles to complete validation. These extreme cases occur very rarely, though, which makes it very likely that an error will occur during a time, when the error detection latency is short. Note that there already is a two-fold increase between the maximum and average latency, as, for each transaction, the maximum latency spans from first instruction to the checksum comparison, while the average is taken from the latency between each instruction and the checksum comparison.

If the error detection latency is too high for the intended application, it can be lowered by reducing the targeted transaction length. This does not only reduce the average, but also the maximum. One has to expect a decline in performance, though, as this will cause more transaction boundary overhead. It can be considered to increase the targeted transaction length to reduce overhead. However, this will only work for some benchmarks. If the error detection latency becomes too large, cache lines will be evicted, before the trailing cores have validated them, which results in more cache misses. Thus, increasing the targeted transaction length, will only improve performance for benchmarks with low cache miss rate.

VI. Conclusion & Future Work

In this paper, we have shown that multiversioning is a viable approach to implement fail-operational multithreaded applications. For the evaluation, we have ported the pthreads library. Atomic operations are supported, too. Therefore, we assume that most shared memory parallelization libraries could also be easily ported. Most benchmarks already perform well (2.11 geometric mean slowdown) even without changes. If an application runs slowly, simple changes like ensuring proper padding to avoid false sharing can result in large performance gains, e.g. 37.5 % in *streamcluster*. Only if the application is optimized heavily for execution on a specific non-transactional-memory architecture, larger changes might be necessary. The approach also features a low error detection latency of on average 10,299 cycles, making it suitable for use in systems, which require common output. Thus, we conclude that our approach should be applicable to most shared memory applications on general purpose and embedded systems.

The results, which are presented in this paper, do not even show the full potential of this approach, as further optimizations like perfect prefetching and branch outcome forwarding, which we have already presented in [6] are not yet included. These optimizations accelerate the trailing cores, which reduces the amount of trailing cores needed. This enables a system developer to either improve the performance by switching them to leading cores or reduce the power consumption by turning them off. Due to our previous work [6], we see great potential in applying this approach to large heterogeneous systems to allow for high performance, energy efficiency and fail-operational execution. Thus, we plan to replace the currently used closed-source MicroBlaze core with diverse open-source RISC-V cores in order to integrate the previously released optimizations. This would also allow us to isolate the LUTs on the FPGA, which are contained in the Sphere of Replication, in order to perform fault injection testing.

References

[1] ARM. Cortex-r - arm developer. [Online]. Available: https://developer.arm.com/ip-products/processors/cortex-r

[2] R. Wegrzyn, S. Gross, V. Patel, A. Bulmus, and C. M. Rimoldi, "Safety design for modern vehicles - including ev/hev." [Online]. Available: https://www.infineon.com/dgdl/Infineon-Safety%20Design%20for%20Modern%20Vehicles%20Whitepaper-Whitepaper-v01_00-EN.pdf?fileId=5546d4626cb27db2016d24bcb8b26396

[3] S. Mukherjee, *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers Inc., 2008.

[4] B. Döbel and H. Härtig, "Can we put concurrency back into redundant multithreading?" in *Proceedings of the International Conference on Embedded Software (EMSOFT)*. ACM, 2014.

[5] A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, and D. A. Connors, "Using process-level redundancy to exploit multiple cores for transient fault tolerance," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE, 2007.

[6] R. Amslinger, S. Weis, C. Piatka, F. Haas, and T. Ungerer, "Redundant execution on heterogeneous multi-cores utilizing transactional memory," in *International Conference on Architecture of Computing Systems*. Springer, 2018.

[7] F. Haas, S. Weis, T. Ungerer, G. Pokam, and Y. Wu, "Fault-tolerant execution on cots multi-core processors with hardware transactional memory support," in *International Conference on Architecture of Computing Systems*. Springer, 2017.

[8] G. Yalcin, O. S. Unsal, and A. Cristal, "Fault tolerance for multi-threaded applications by leveraging hardware transactional memory," in *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, 2013.

[9] D. Kuvaiskii, R. Faqeh, P. Bhatotia, P. Felber, and C. Fetzer, "Haft: Hardware-assisted fault tolerance," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016.

[10] P. A. Bernstein and N. Goodman, "Multiversion concurrency control—theory and algorithms," *ACM Transactions on Database Systems (TODS)*, vol. 8, no. 4, 1983.

[11] The PostgreSQL Global Development Group. Postgresql: Documentation: 12: 13.1. introduction. [Online]. Available: https://www.postgresql.org/docs/12/mvcc-intro.html

[12] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, 2011.