# Universität Augsburg

## Multiversioning Hardware Transactional Memory for Fail-Operational Multithreaded Applications

Rico Amslinger, Christian Piatka, Florian Haas,
Sebastian Weis, Theo Ungerer, Sebastian Altmeyer

## Institut für Informatik
### D-86135 Augsburg

# Multiversioning Hardware Transactional Memory for Fail-Operational Multithreaded Applications[*]

Rico Amslinger[a,*], Christian Piatka[a], Florian Haas[a], Sebastian Weis[b], Theo Ungerer[a], Sebastian Altmeyer[a]

[a]*University of Augsburg, Universitätsstraße 6a, 86159 Augsburg, Germany*
[b]*TTTech Auto Germany GmbH, Emmy-Noether-Ring 16, 85716 Unterschleißheim, Germany*

## Abstract

Modern safety-critical embedded applications like autonomous driving need to be fail-operational, while high performance and low power consumption are demanded simultaneously. The prevalent fault tolerance mechanisms suffer from disadvantages: Some (e.g. triple modular redundancy) require a substantial amount of duplication, resulting in high hardware costs and power consumption. Others, like lockstep, require supplementary checkpointing mechanisms to recover from errors. Further approaches (e.g. software-based process-level redundancy) cannot handle the indeterminism caused by multithreaded execution. This paper presents a novel approach for fail-operational systems using hardware transactional memory for embedded systems. The hardware transactional memory is extended to support multiple versions, enabling redundant 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. The evaluation shows that multiversioning can successfully recover from all transient errors with an overhead comparable to fault tolerance mechanisms without recovery.

*Keywords:* fault tolerance, redundancy, hardware-transactional-memory, multiversioning, multi-core

## 1. 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 must be fail-operational, as a failure could directly endanger human lives. If a transient error due to a single-event upset occurs, detection and recovery need to be quick, as deadlines still have 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 battery power, which makes a low power consumption essential. Therefore, we consider heterogeneous multi-cores, which consist of fast cores and energy efficient cores executing the same instruction set, to be 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 transient errors, but has to rely on alternative mechanisms like checkpointing for recovery [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, checkpointing is often only performed infrequently in order to limit the overhead, which results in a large loss of progress when recovering.

Triple modular redundancy is an alternative approach to fault tolerance, which is often used in the aerospace industry. While it solves the error recovery problem of lockstep execution, it introduces new issues. The need for three instances of the CPU increases power consumption and production costs in significant ways. Further, the inherent indeterminism of multithreaded 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. 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 can lead to an inoperable system or data loss [3]. Additionally, software-based fault tolerance often suffers from a high performance overhead. Only a few implementations can handle multithreaded execution, as differences in execution order between the redundant instances can result in divergences (see Fig. 1 for an example). These divergences can occur even with proper synchronization, as two threads can enter a critical section in different orders in the redundant instances. The common solution is to wait in the second redundant instance to ensure an identical order. However, this causes an additional performance overhead. Error detection latencies can also be a problem, as many implementations trade high
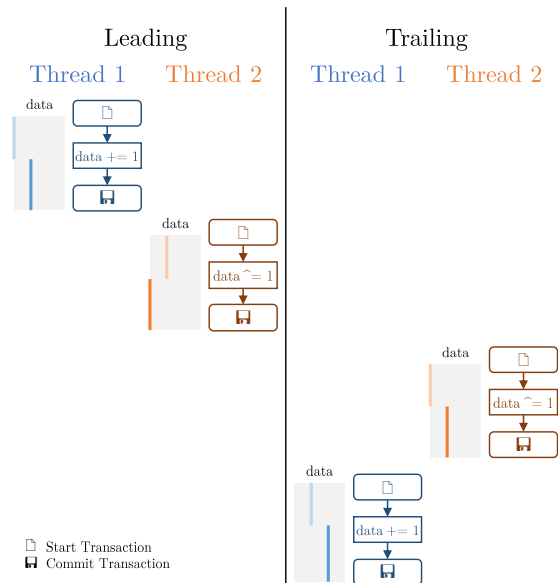
Figure 1: An incorrect execution can occur if the order of the transactions is not preserved between the leading and trailing threads. In this example, thread 1 executes a transaction which increments data by 1, and thread 2 executes a transaction which XORs data by 1. If thread 1 is executed first (leading), the final result is 0. If thread 2 is executed first (trailing), the final result is 2. Thereby, checksums differ and a rollback occurs.

error detection latencies for improved performance.

We present a novel approach for fault tolerance on embedded systems based on multiversioning to mitigate those disadvantages. In addition to homogeneous multi-cores, heterogeneous multi-cores are also supported as long as they execute the same instruction set. In our approach, the application is only executed twice in order to keep the overhead minimal. The underlying fault model regards transient faults, which may corrupt data in a register, but main memory and caches are assumed to be protected by ECC (see Fig. 2). Hardware checksum calculation ensures that every single-bit error is caught. The duplicate execution can also detect errors in instructions like the sine function, which are difficult to check otherwise. Error detection is fast, as transactions offer a quick validation interval. A transactional memory based rollback mechanism allows for cheap recovery after detecting an error. 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 regular transactional memory system cannot fulfill these requirements. One key problem is the preservation of the order of the transactions between the leading and trailing execution to avoid unnecessary and potentially infinite rollbacks. Fig. 1 depicts in detail why this is a problem. To solve this and other issues we developed a transactional memory which supports multiple versions of
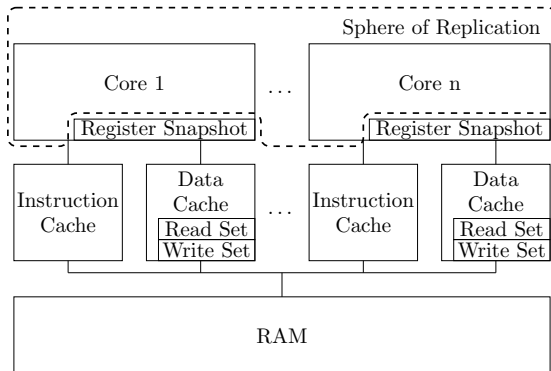
3

Figure 2: The cores and data caches are extended to support transactional memory and multiversioning. The register snapshot can also be stored external to the core, as its access latency is not performance critical. The sphere of replication, marked by the dotted line, covers the pipeline in the cores. The remaining components are protected by ECC.

the same data word. The multiversioning mechanism is described in more detail in Section 4. A transactional memory based pthreads implementation ensures backward compatibility for classic multithreaded applications, which rely on atomic operations and cache coherence.

We performed an FPGA-based evaluation running the PARSEC benchmark suite [5] on Xilinx MicroBlaze soft cores [6]. Although these cores are closed-source, our approach can be integrated with unmodified processor cores like the MicroBlaze, as it is mostly contained in the caches. All necessary communication is handled over existing infrastructure like AXI busses, interrupt lines and trace ports.

Altogether, our approach has the following five advantages:

- The system is fail-operational, as it can recover from errors.

- Homogeneous and heterogeneous multi-cores executing an identical instruction set are supported.

- Shared memory multithreaded applications can be executed redundantly.

- Transactional memory can be used for synchronization.

- No modifications to the cores are required.

Our work is structured as follows. First, we present related work. In Section 3, our redundancy concept is explained. Section 4 describes the extension to multithreaded execution using multiversioning. Section 5 consists of optimizations, which improve the performance of our approach. In Section 6, we present the evaluation of our approach. We evaluate runtime overhead, impact of the optimizations, fault injection and error detection latency. This section contains an explanation of the methodology and an analysis of the results. At the end of the paper, a conclusion and an outlook on future work is given.

4

This publication extends our previous paper [7] in the following ways:

- Description and evaluation of hardware optimizations

- Implementation of fault injection techniques

- Evaluation of the benchmarks' susceptibility to errors and of the multi-versioning recovery mechanism

## 2. Related Work

Process-Level Redundancy [8] is a software-based approach to provide fault tolerance for single threaded 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 number of free cores is available for the redundant processes. As system calls can be far apart, we expect the error detection latency to be unsuitable for embedded systems which are typically subject to stringent real-time constraints.

The software approach of RomainMT enforces determinism for multi-threaded applications to enable a redundant execution on the L4 microkernel [4]. On externalization events, which for example are system calls, the states of the redundant threads are compared to detect errors. This requires the redundant threads to be in an identical state to avoid the false detection of an error. By enforcing the same locking order of mutexes on every execution of a program, the observable behavior will be identical, but only if no race-conditions exist and no lock-free atomic memory accesses occur. RomainMT relies on an external checkpointing and recovery mechanism for DMR, but for triple modular redundancy, forward error correction is possible by selecting the two error-free threads.

In our previous work [9], we have already presented a hardware fault tolerance mechanism for single threaded applications, which is the foundation of this approach. However, multiversioning, which is essential to realize multi-threaded 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.

FaulTM-multi [10] is a hardware fault tolerance implementation utilizing transactional memory. In contrast to our approach, FaulTM-multi is tightly coupled. This results in several restrictions: It is not possible to have unequal numbers 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. Unsteady optimizations like those described in [9] cannot be used either, as the original thread cannot run ahead to compensate for any fluctuation.

In other previous work [11], we have described a software-based approach to fault tolerance by utilizing the hardware transactional memory 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.

HAFT [12] 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.

Execution Replay of Multiprocessor Virtual Machines [13] implements a software approach that allows the reproduction of the execution of virtual machines, which can be applied for fault tolerance. The approach uses the MMU to mark pages for either concurrent-read or exclusive-write. When a core executes an operation that is not allowed in the current state of the page, the state is changed and the transition is recorded as dependency in a log. During the repeated execution, these recorded dependencies are used to exactly reproduce multithreaded execution. The conflict detection granularity is at page-level, while we detect them a cache-line-level. Additionally, our ownership transitions are quicker, since they are realized in hardware instead of software.

Samsara [14] improves upon the previous approach by splitting the execution into chunks, comparable to our transactions. At the end of a chunk's execution, the accessed and dirty bits in the page table are used to detect conflicts. Similarly, to transactional memory, conflicting chunks need to be repeated, which requires the use of copy-on-write for each page written in a chunk. Since the approach is also based on the page table, granularity is more coarse than with our approach. Additionally, chunks have to be larger than our transactions to accommodate for the copy-on-write and commit overhead. Therefore, we expect the error detection latency in Samsara to be higher than with our approach.

Multiversion concurrency control [15] is a method used by databases to manage concurrent accesses. This technique is used by PostgreSQL, for example [16]. We suppose that hardware multiversioning support, like the one provided by our approach, can be exploited by such applications to benefit performance.

## 3. Error Detection Mechanism

We assume a standard shared memory hierarchy, as depicted in Fig. 2, consisting of multiple cores. All cores execute the same instruction set, but their microarchitectural implementation can differ, e.g. to realize a combination of fast and energy-efficient cores. The 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.
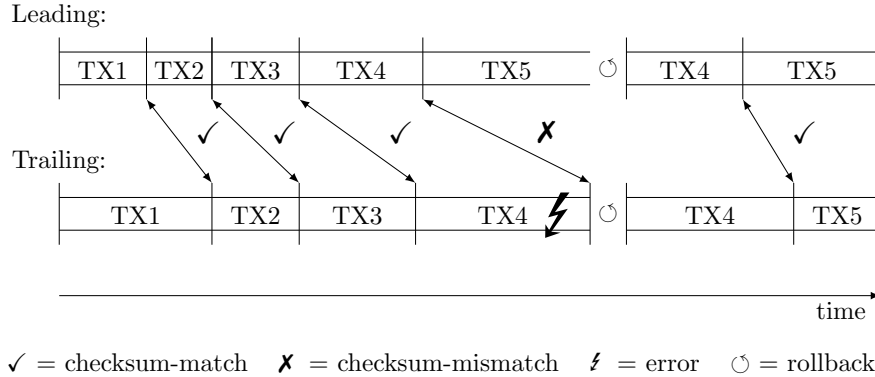
Figure 3: The single threaded application is split into transactions TXi, which are executed redundantly on a leading core and a trailing core. For some time the checksums match, but after TX4 a bitflip causes a mismatch. This results in a rollback and a restart of TX4.

## 3.1. Transactional Memory

In order to implement multiversioning, we have extended the memory hierarchy in several ways. 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.

Our approach implements a leading/trailing execution concept similar to the one we have used in [9]. The program is first executed on one or more leading cores. Their results are then validated by one or more trailing cores, which execute the same code.

To realize this concept, the execution is split into transactions (TXi in Fig. 3). 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, where the 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

7

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.

*3.2. Redundancy*

An error cannot propagate to the other core, as both cores can only see their own modifications to the memory. In the single threaded 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 single checksum of all instruction outcomes is calculated. Our definition of instruction outcome includes every output signal caused by an instruction that affects the architectural state. For an arithmetic instruction, for example, this can be the output register value and state flags like carry, but also the next instruction address. The information of one instruction outcome is concatenated and streamed to the hardware checksum unit. Here a checksum is calculated by hashing the arbitrary sized input to a constant size. Note that the input is not required as a whole to calculate the checksum. 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. 3), both cores need to roll back to the beginning of their transactions. Consequently, 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 does not occur again and the checksums match after both transactions have been repeated.

## 4. 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 and sensor inputs 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

Figure 4: Above, the same application as in Fig. 1 is shown, 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 V1 and the result after the second transaction in version V2. The version numbers of the leading transactions are transferred to the trailing threads. Even if the thread 2 is executed first in the trailing execution, it still loads version V1 as base. This is possible, as the version is still retained from the leading execution. Thus, the final result is also 0. Version V1 is validated later and the speculative resources are dropped, as version V2 is already available.

one. It is possible to use slower but more energy-efficient cores as trailing cores as long as they execute the same instruction set. In this case, a different ratio might be chosen. If an application has plenty of waiting time, the number 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 [9]. The trailing cores will simply switch between validating those leading cores that are currently not waiting.

Not preserving the order between the leading and trailing execution within a regular transactional memory system can cause unnecessary and potentially infinite rollbacks (see Fig. 1). If situations like this happen frequently, 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. Further, consistent rollback checkpoints between all threads are necessary. 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. Those will be made visible to the other leading cores, as soon as the transaction has committed. 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 restarted.

Therefore, it is possible to execute shared memory multithreaded 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. A hardware queue is used to transfer the register set at the start of the previous transaction to the trailing cores. Each queue entry also contains additional data like the checksum and version number. It is possible to use the memory bus for communication between the cores and the queue. However, a dedicated interface reduces the overhead. A single queue can be shared between all cores, as queue entries are quickly picked up by a trailing core. This trailing core starts a new transaction with the same base version and executing the same code as the corresponding leading core. 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. This reset procedure can be realized by either sending a reset signal to the cores or the use of interrupts. As errors are expected to be rare, performance of the reset procedure is not of primary concern.

Implementing multiversioning requires major changes to the caches. However, some systems might already offer many of the required features. For example, systems with Intel TSX already offer conflict detection and support at least two versions of each cache line. Therefore, the adaption of such a system is

expected to be easier than the adaption of a system without any native support for versioning. The approach is largely independent of the microarchitectural implementation and can be implemented in its basic variant without changes to the cores. However, some modifications to the cores can be made to improve performance or fault detection.

For user space applications, our system offers the same interface as regular transactional memory systems. Explicit transactions for synchronization, similar to those in classic transactional memory, are available. The automatic transactions that are used for redundancy do not require any special handling by the programmer. The `tm_begin` operation commits the running automatic transaction and starts a new explicit transaction. It is possible to either commit the explicit transaction with the `tm_commit` operation or abort it with `tm_abort`. In both cases, the core will start a new automatic transaction.

The operating system uses an additional control register to manage multi-versioning. This control register can be used to configure cores as leading or trailing cores. It can also be used to disable redundancy or automatic transactions, when code should not be executed redundantly, e.g. for I/O.

Porting a user space application to our platform mainly concerns synchronization. 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 only necessary if synchronization constructs are implemented directly. Atomic operations can be replaced by small transactions, which only encapsulate 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 semantics of the operation does not change. By committing the transaction right away, the modified value becomes visible to other cores immediately. For more details regarding the implementation see [17].


## 5. Optimizations

Various optimizations were implemented to improve the performance of the multiversioning system.


### 5.1. pthreads Library

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 number 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.

## 5.2. Validation of Automatic Transactions

On ordinary transactional memory systems, every transaction performs conflict detection. In our approach, the code executed in automatic transactions does not require conflict detection, since it is functional without transactions in the non-redundant case. Some care still needs to be taken to ensure that changes to the same cache line by different cores are merged properly. However, a major part of conflict detection, referred to as "validation", can be disabled to improve performance. Synchronization constructs can still use explicit transactions with conflict detection even if the validation of automatic transactions is disabled.

One general issue, which affects many transactional memory applications to varying degrees, is false sharing. This effect 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 when 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.

By default, every transaction performs full conflict detection. If a conflict is detected, all except one conflicting transactions are restarted. Due to false sharing this can happen quite often. If an application is ported from a non-transactional implementation, the conflict detection is not necessary, as the same code works without conflict detection. It is not possible to disable write-after-write conflict detection, as this would make complicated merges of cache lines that were changed by two or more cores simultaneously necessary. However, the second core can request the first transaction to commit prematurely. This costs some performance, but is better than aborting. Read-after-write conflicts can be handled likewise. Read-after-read conflicts cause no issues, even in the base implementation, as both transactions can continue with the cache line in shared state.

Write-after-read conflicts require special consideration. We observed that write-after-read conflicts can often be ignored. Write-after-read conflict detection is the most expensive, as conflicts, in which the write occurs first, can be detected by the existence of the new version without any overhead. Therefore, ignoring write-after-read conflicts does not only reduce overhead from the repeated execution of aborted transactions, but also from conflict detection itself. However, if a benchmark contains race conditions, these can cause trouble with redundant execution. If the order of the trailing execution is swapped, the trailing cores might now return a different result. This then causes a rollback of the whole system. Contrary, if write-after-read conflicts are handled on the leading cores, only a single core has to roll back for each conflict. Note that a write to a single cache line can still cause conflicts with multiple reading cores, which can result in additional rollback even if the conflicts are handled on the leading cores.

*5.3. Bloom Filter*

Another way to reduce conflict detection overhead is the use of a Bloom filter [18]. A Bloom filter is a constant size, probabilistic data structure implementing a set. It is possible to add memory addresses to the set. One can also query, whether a specific address is contained in the set. However, the more addresses are added, the higher the probability that a query falsely returns that the address is contained in the set (false positive). It is not possible for a query to falsely return that an address is not contained (false negative). Therefore, this data structure is well suited for conflict detection, as a false conflict only costs performance, while a missed conflict could cause wrong outputs.

In our implementation, the Bloom filter is used to optimize write-after-read conflict detection. Every core has its own associated Bloom filter. The Bloom filter can be placed outside of the core, since it only requires the address of the accessed cache line and some control signals. When a transaction reads a cache line, its address is added to that core's Bloom filter. When a transaction writes a cache line, every other Bloom filter is queried for its address. If a Bloom filter returns that it contains the address, regular conflict detection and resolution is performed for the corresponding core. This optimization reduces the number of cache misses and the overhead for commits, as it is not necessary to iterate the readset if the bloom filter did not indicate any hits. The Bloom filter is reset on commit or abort of the transaction on the corresponding core.

The implementation used in the FPGA is kept simple, as the number of accessed cache lines in each transaction is rather small, and fast lookups are required. Therefore, only a single hash function is calculated for each entry. The corresponding bit in the constant size memory is set to one, when the entry is inserted. To query whether the entry is contained, it is sufficient to return the corresponding bit. If the allowed transaction size is increased, a more complex Bloom filter, which uses multiple hash functions, should be used. In such a Bloom filter an entry is only considered contained if the corresponding bits of all hash functions are one. Using the optimum number of hash functions results in a minimal false positive probability.

*5.4. Cache Line Compression*

In the basic variant, an additional cache line is allocated for each new version of a cache line. The additional cache line is referenced by a pointer (e.g. stored in the ECC bits). This can be optimized by directly storing the changed data in the field for the address of the version. Note that this optimization is in contrast to the overall approach, processor dependent and the precise may change when a different instruction-set-architecture is used.

Cache line compression is possible if at most eight bytes have changed. The change has to consist of two aligned words of four bytes each, one at an even index (third address bit is zero) and one at an odd index (third address bit is one) in the cache line. Allowing this split increases the likelihood that a stack cache line can be compressed. Because of the index requirement only one comparison is required per memory access, as the CPU must split accesses

so that every access is aligned to the bus width of four bytes. If more data is written, the additional cache line is allocated and the existing changes are transferred to free the address field.

This optimization reduces the number of cache misses, as the cache miss for version allocation is avoided. Additionally, fewer cache lines are evicted prematurely. The optimization also reduces the overhead when reading or writing a version that is not the safe version. If the required version is stored in another cache line, at least one additional cycle is required to access it. This overhead is avoided, as the address field can be accessed in the first cycle.

### 5.5. Fresh Fetch

When a new version is allocated, the data of the cache line is copied to a fallback cache line. This means that a cache line, which has not been accessed for a long time, is written. In most cases, this causes a cache miss. This optimization avoids the overhead of the actual fetch by clearing cache lines to zero instead of fetching them if they will be completely overwritten. This is always the case for version allocations. Note that the overhead for the writeback of dirty cache lines can still occur.

### 5.6. Sticky Trailing Threads

When a transaction commits on the leading core and multiple trailing cores are ready, the trailing core that confirms the transaction is selected arbitrarily. If sticky trailing threads are enabled, trailing cores, whose previous transaction was from the same leading core, will be preferred. Most of the time, one trailing core will stick to validating exactly one leading core. This reduces cache misses, as the trailing core's cache already contains some reused cache lines from the last transaction, e.g. the stack.

### 5.7. Important Writes

The data written by trailing cores does not need to be preserved for further execution, as all later transactions use the leading core's version. For fault detection, it is included in the checksum, but the actual changed value is not stored. We assume the cache and main memory are protected by other means like ECC. Therefore, the data written by the trailing core is not needed to ensure their correctness.

However, sometimes the data is necessary to correctly execute the trailing transaction. Assume a case where a transaction first writes a location, then reads it and finally overwrites it. The read value is stored neither in the safe version nor the leading version. Therefore, the trailing core has to actually perform these stores.

We use a 128 bit Bloom filter with a single hash function to detect these stores. When the leading core reads a cache line, which is already contained in the writeset, it adds its address to the Bloom filter. The Bloom filter is transferred to the trailing core together with the registers using the shared hardware queue. The trailing core then checks this Bloom filter on every write

14

and only performs it if the address is contained. This ensures that every write, which can influence the rest of the transaction is actually performed. The probabilistic nature of the Bloom filter can only result in unnecessary writes, but never in missed writes.

This optimization reduces cache misses in the trailing core, as cache lines, which are written but never read, are not fetched. Additionally, less cleanup is required at the end of the trailing transaction, as less versions were created. Therefore, the trailing core is ready to execute the next transaction more quickly, which results in less waiting time for the leading cores.

## 6. Evaluation

We evaluated our multiversioning approach in concern to runtime overhead, scaling, impact of optimizations, fault injection and error detection latency. First, we describe our methodology. Then, separate analyses of the evaluated metrics follow.

### 6.1. 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 [6] cores with support for single precision floating point operations. The cores are connected to coherent private data caches and instruction caches (each 16 kB, 4-way 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. CPU 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 with the main limiting factors for the clock rate being the performance counters and assertions.

We used the PARSEC Benchmark Suite [5] version 3.0 for the evaluation. Note that this is not a throughput evaluation with multiple single threaded 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 their targeted C standard is too old for our build environment.

Except for fault injection, the benchmarks used 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. In total, this evaluation requires 1456 individual benchmark runs. The benchmarks were executed

entirely, but only the region of interest was considered for the measurement the execution times. The region of interest is predefined by the creators of the benchmarks and contains the parallel part of the execution. It excludes the initialization logic which is irrelevant for our evaluation. As we only have a single FPGA board available and due to time constraints, we have executed each configuration only once for runtime analysis. 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 discards its outputs.

To ensure proper operation of our system, we performed a fault injection analysis. The injection is performed in hardware, but is triggered by software running on the host machine. As the cores are closed source, we had to add the injection logic at gate level. We are also limited in the ways, in which we can inject faults. The injection is implemented by xor gates in front of the register set's write port. The other input is set to zero by default, but the host machine can change this value. If a bit is set to one, it remains high until a register is written. This is done to increase the number of faults, which have a chance to affect the application output. Otherwise, many attempts would be wasted to cache misses or instructions, which do not write registers.

We collected the region of interest's start and end times for each benchmark. Before the benchmark starts, the host machine selects a random time in this period. It then triggers the injection once the selected time has elapsed. The core and affected bit are also chosen randomly. The only condition for a core to be eligible is that it has to be active. Therefore, the injection includes non-redundant, leading and trailing cores.

Every benchmark was executed 50 times on the variant without as well as on the variant with redundancy. The validation of automatic transactions is enabled to ensure that race conditions cannot cause checksum mismatches. We used all 12 available cores and the maximum thread count. The benchmark *raytrace* was not included in the analysis, as it takes a long time ($\sim$ 1h) for initialization. The benchmark *canneal* was excluded, too, as its output fluctuates too much to detect errors. Except for the benchmark *x264*, all benchmarks use the *simsmall* configuration to allow for quicker injection attempts. The benchmark *x264* crashes if the number of threads is greater than the number of frames. Therefore, it requires the *simmedium* configuration.

We can validate the outputs of all used benchmarks. Therefore, we can differentiate the following groups of outcomes:

- Valid output, benign error, error corrected

- Invalid output, silent data corruption

- Freeze

- Crash

16

Furthermore, we can detect, whether a global rollback was performed in the fault tolerant variant. Remember that a global rollback can only be caused by a checksum mismatch and the first checksum mismatch always causes a global rollback, but additional mismatches before the rollback do not cause another one. Therefore, the presence of the global rollback indicates, whether the fault was detected.

Part of this evaluation was already performed in our previous publication [7]. All benchmark outputs in the old evaluation were successfully validated. However, while performing the fault injection testing, we noticed that some benchmarks occasionally crash even without injected fault. Resolving these crashes required changes to the linker scripts and compiler flags, which sometimes impacted performance. Notice that this does not necessarily result in worse speedups, as the the baseline is affected as well. We decided that consistency in this publication is more important than comparability with our previous work. Therefore, we use the updated versions of the benchmarks for all figures in this evaluation.

*6.2. Execution Time Overhead*

We 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 academic and do not expect it to occur in practice.

Executing the application twice one after another forms an upper bound for the runtime. An approach to fault tolerance should be able to outperform this bound in order to best the naive "execute it twice and compare the results" software 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 2.16. Below, we describe the results shown in Fig. 5 in detail.

The benchmark *vips* behaves as expected. An analysis of the benchmark shows that the execution without redundancy follows Amdahl's Law. The ex-
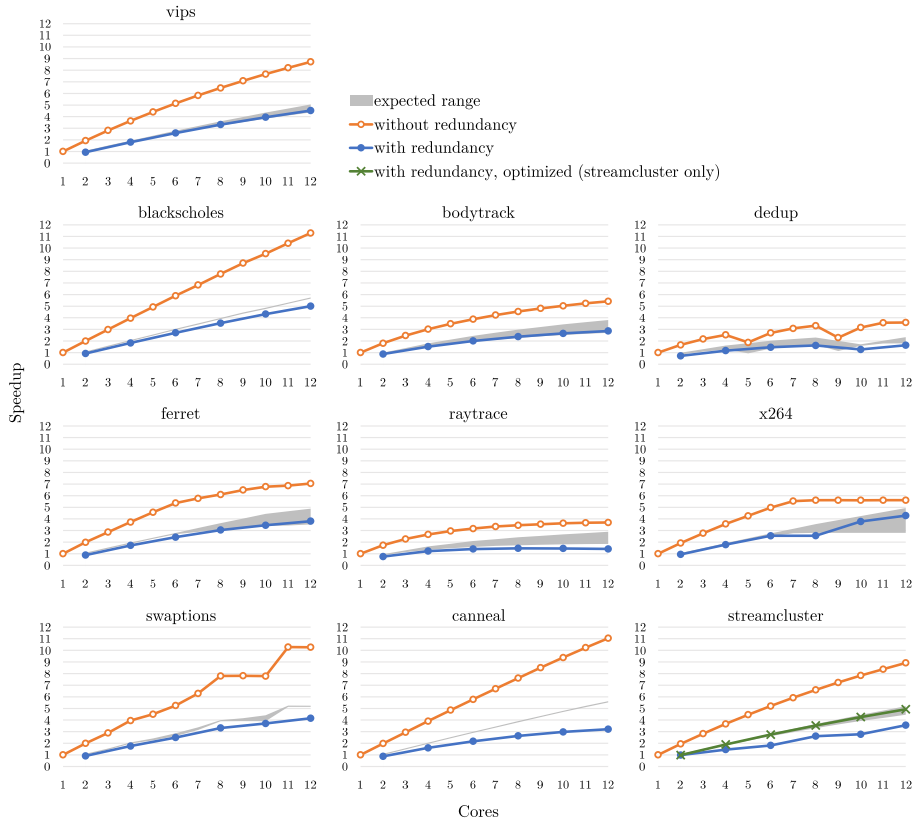
Figure 5: These charts show the speedup of the various PARSEC benchmarks in the different configurations at a certain core count. All speedups are relative to the single threaded execution without redundancy and consider only the region of interest. For the variant without redundancy, the benchmarks 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.

ecution 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 8 core redundant run shows that executing more iterations for benchmarking would be favorable. Sometimes the benchmark *x264* takes longer to complete for no obvious reason. We suspect that an unlucky memory allocation ordering leads to an unfavorable interleaving of memory regions.

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 38.6 %.

### 6.3. Optimizations

We have implemented several optimizations (see Section 5) to improve the performance. These optimizations are already enabled in Fig. 5.

Fig. 6 shows the effects of each hardware optimization by itself. We did not enhance the baseline pthread library with support for redundant execution. Additionally, using a version of a library that is unsuitable for the employed hardware is unreasonable. Therefore, we did not evaluate it. For this evaluation each configuration was executed three times. The median execution time was used to calculate the speedups shown in Fig. 6.

The speedup of the different optimizations does not stack additively, as they aim at similar aspects. If one optimization has already reduced the number of cache misses, the possible further acceleration by other similar optimizations is limited. Furthermore, some optimizations directly reduce the effect of others. For example, not validating automatic transactions reduces the effect of the
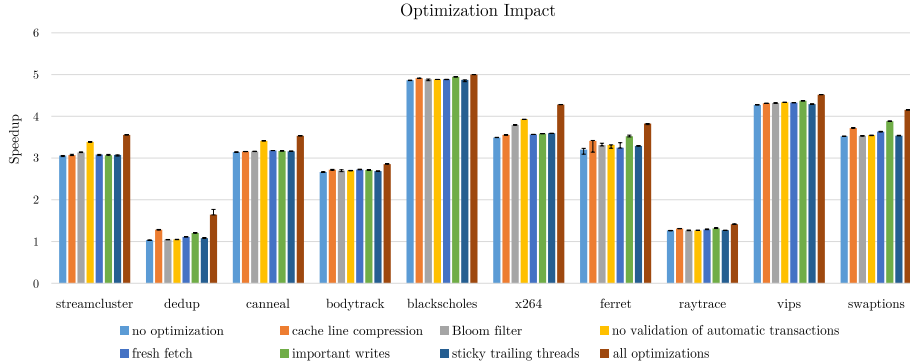
Figure 6: This bar chart shows the speedups of various configurations relative to the single threaded baseline without transactions or redundancy. The configurations were run on 6 leading and 6 trailing cores with multiversioning enabled. Only the region of interest is considered. Each bar represents the median runtime for that benchmark configuration. The error indicators extend to the slowest and fastest runtimes, respectively. The bars are in the same order as the legend entries.

Bloom filter as less conflict detection is performed. Compression reduces the effect of fresh fetches, as fallback cache lines are required less often.

At the same time, it is also possible that the combination of two optimizations results in a larger acceleration than the sum of them. For example, a benchmark might be limited by the performance of its trailing cores. This makes all optimizations affecting the leading cores seem useless by themselves. Optimizations affecting the trailing cores only improve runtime to the level of the leading cores. However, if all optimizations are applied at once, the achieved acceleration is greater, as now neither the initial runtime of the leading cores nor the trailing cores is limiting.

Disabling the validation of automatic transactions has a large effect on some benchmarks. If the effect is that large, it cannot be explained by reduced conflict detection overhead alone. This means that there are transactions, which aborted and retried if automatic transactions are validated. The explanations for this behavior are false sharing and race conditions, as we have seen from a detailed analysis of the results. The variant of the benchmark *streamcluster*, which is run here, uses the wrong cache line size (see Section 6.2). Therefore, this benchmark suffers from false sharing.

However, the benchmarks *canneal* and *x264* are not free from race conditions. Disabling the validation of automatic transactions for those benchmarks is not without downsides. 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. In this evaluation, there was an average of 20.5 global rollbacks in *canneal* and 0.8 in *x264* when the validation of automatic transactions is disabled. The optimal solution would be to fix the source code, as those race conditions can result in

Table 1: Error Detection Latency

| Benchmark | Average [cycles] | Maximum [cycles] |
|---|---|---|
| vips | 9,833 | 181,129 |
| blackscholes | 9,985 | 25,263 |
| bodytrack | 9,855 | 30,976 |
| dedup | 8,621 | 150,333 |
| ferret | 9,697 | 156,534 |
| raytrace | 8,359 | 59,790 |
| x264 | 10,492 | 108,198 |
| swaptions | 9,852 | 29,083 |
| canneal | 7,765 | 31,780 |
| streamcluster | 8,890 | 49,056 |
| overall | 9,335 | 181,129 |

The average error detection latency is the average number of cycles 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.

wrong results even on other architectures.

Other effective optimizations include cache line compression and important writes. We did not observe any benchmark becoming slower if a certain optimization is enabled. Additionally, the hardware cost for most optimizations is negligible. For example, disabling the validation of automatic transactions only involves a single AND gate, since the signals for automatic transaction and conflict already exist. Other optimizations like fresh fetch are probably already included in more sophisticated systems. Therefore, there is no downside to enabling most optimizations even if the achieved acceleration is low. Disabling the validation of automatic transactions is the obvious exception because it can lead to livelocks due to infinite global rollbacks as described above.

### 6.4. Error Detection Latency

We have also analyzed the error detection latency on a system with 6 leading and 6 trailing cores. The average and maximum values are shown in Table 1.

The resulting average values clearly reflect the targeted transaction duration of 10,000 cycles. Many automatic transactions hit this target quite 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 more comprehensive implementation, this would most likely not be an issue.

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 an increased 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 a low cache miss rate.

### 6.5. Fault Injection

Fig. 7 shows the results of the fault injection without redundancy. Most runs complete correctly even if faults are present. Especially, the benchmarks *blackscholes*, *vips*, *x264* and *bodytrack* mask faults effectively. In the cases in which the bitflip influences the output, the result is likely a crash. These crashes are mostly invalid instruction or unaligned access exceptions. An invalid instruction exception can be caused by a corrupted return address. Since no MMU is used, the core tries to execute the data at the invalid address as instructions instead of throwing a segmentation fault. However, it is very unlikely that a random memory location contains a sufficiently long sequence of valid instructions. The MicroBlaze cores can only handle aligned memory accesses. Therefore, an unaligned access exception is thrown if the lower bits of a pointer are corrupted. The benchmarks *streamcluster*, *dedup* and *ferret* also tend to freeze. This can happen if the most significant bit of a loop counter is flipped. As most loops use signed counters, this means that the loop will be executed around 2 billion times, which takes so long that the timeout in our benchmarking script triggers. Faulty outputs are possible if the flipped bit affects actual data. However, they are quite unlikely. In our evaluation, faulty outputs have only occurred in the benchmarks *streamcluster*, *dedup*, *swaptions* and *bodytrack*.

As shown in Fig. 8, our multiversioning approach prevented all undesirable outcomes in our evaluation. In 72.8 % of the runs a global rollback was necessary. A global rollback only occurs if a checksum mismatch was detected. The main cause why a bitflip does not result in a checksum mismatch is that the containing transaction was aborted. Aborted transactions are not validated by the trailing cores, as they cannot influence the program output. Many aborted transactions
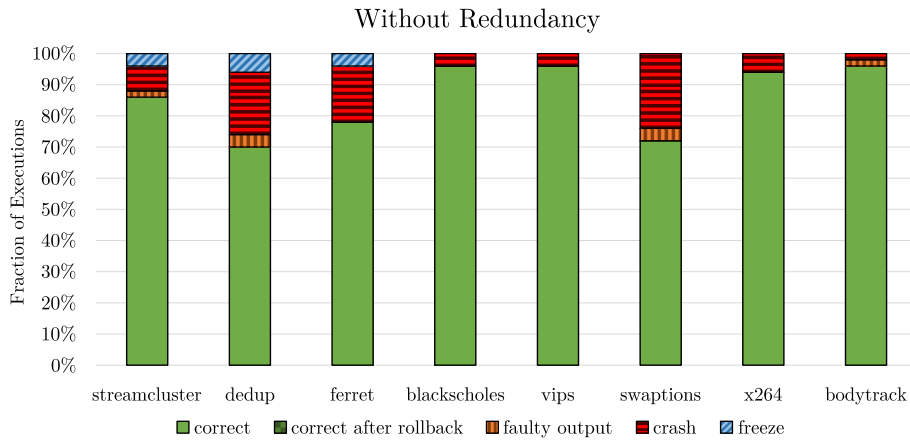
## Without Redundancy



Figure 7: Without redundancy the effect of injected faults can be observed. Each benchmark was executed 50 times and a single bitflip was injected to a random core in each run. After the benchmark completed or timed out, the output was validated and classified as correct, faulty output, crash or freeze. Rollbacks can only occur in the redundant variant with multiversioning.

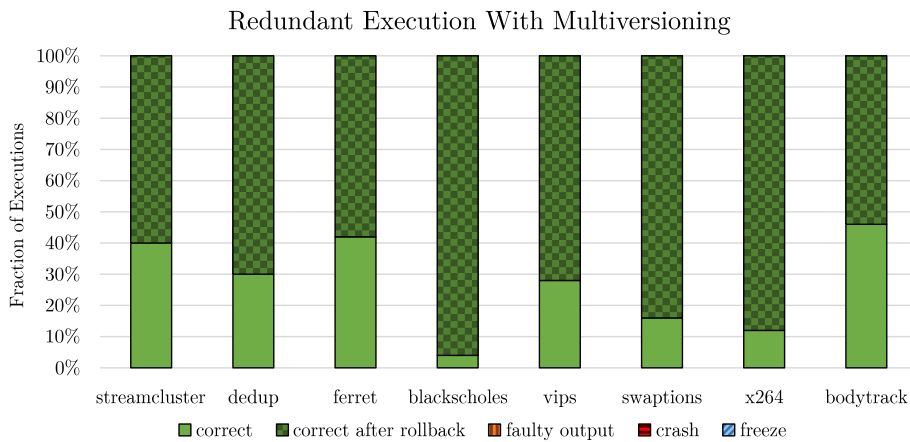## Redundant Execution With Multiversioning



Figure 8: This bar chart shows the results of the fault injection, when redundancy with multiversioning is used. Each benchmark was executed 50 times and a single bitflip was injected to a random core in each run. After the benchmark completed or timed out, the output was validated and classified as correct, faulty output, crash or freeze. In this evaluation, every output was correct. Sometimes, a global rollback was required to achieve this.

happen while the thread is waiting for a synchronization construct or the core is idle. These bitflips would not have affected program output in the baseline variant either. Therefore, the number of rollbacks in Fig. 8 exceed the number of faulty executions in Fig. 7. The bitflip itself can also be the reason for the transaction abort. For example, it can shift the stack pointer to overlap with a different thread. The conflicting memory accesses are detected and one of the transactions is aborted.

Note that a checksum mismatch does not necessarily mean that an undesirable outcome would have happened without redundancy. For example, it might hit a comparison, for which only the sign bit is relevant. In this case, bitflips in the lower bits do not influence the program output. Therefore, the likelihood of a checksum mismatch correlates more strongly with the transaction success rate than an application's susceptibility to errors. For example, the benchmark *dedup* has the most undesirable outcomes in the baseline, but has executed slightly below average global rollbacks. Contrary, the benchmark *blackscholes* has executed the most global rollbacks, but is one of the benchmarks with the least undesirable outcomes in the baseline. The high number of checksum mismatches in this benchmark is mainly caused by the low number of transaction aborts, as the threads process independent work items.

The timing impact of global rollback was below the regular runtime fluctuation caused by varying memory response times and thread ordering. The largest contribution to the timing overhead is the staggered restart of the cores. Staggered restarts have the advantage that checksum mismatches caused by false sharing are resolved, as the offending transactions run successively. Therefore, an infinite loop of global rollbacks is avoided.

## 7. 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 (2.16 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. 38.6 % in *streamcluster*. Hardware optimizations, which provide additional performance without changing the software, are also available. 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 9,335 cycles on average, making it suitable for use in systems that require common output. The evaluation has also shown that our approach can reliably detect errors and recover from them. A measurable performance overhead of the global rollback was not observed. Thus, we conclude that our approach should be applicable to most shared memory applications on general purpose and embedded systems.

Currently, our prototype is limited in the types of faults it can handle. For example, injecting a large number of bitflips will eventually lead to a freeze, as the core tries to access an invalid address. As it never receives a response, it cannot enter the interrupt handler to perform the rollback. Note that this situation is rare, as few address ranges behave this way. We plan to add an MMU to prevent such situations. Furthermore, we want to switch the recovery mechanism from interrupts to resets so that we can recover from more situations.

The results do not even show the full potential of this approach. Further optimizations like perfect prefetching and branch outcome forwarding, which we have already presented in [9] are not yet included. These optimizations accelerate the trailing cores, which reduces the number 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 [9], we see great potential in applying this approach to large heterogeneous systems that execute the same instruction set to allow for high performance, energy efficiency and fail-operational execution. Thus, we plan to replace the currently used closed-source MicroBlaze core with open-source in-order and out-of-order RISC-V cores. In future work, we will also investigate further applications for multiversioning. For example, an acceleration of database accesses might be possible. Furthermore, synchronization of sensor data between tasks on an embedded system can be realized using multiversioning. A possible integration of our Transaction Management Unit [19] simplifies the adoption on embedded systems.

## References

[1] ARM, Cortex-r - arm developer (Accessed: 2021-02-24).
URL https://developer.arm.com/ip-products/processors/cortex-r

[2] R. Wegrzyn, S. Gross, V. Patel, A. Bulmus, C. M. Rimoldi, Safety design for modern vehicles - including ev/hev (Accessed: 2021-02-26).
URL https://www.infineon.com/dgdl/Infineon-Safety%20Design%20for%20Modern%20Vehicles%20Whitepaper-Whitepaper-v01_00-EN.pdf?fileId=5546d4626cb27db2016d24bcb8b26396&da=t

[3] S. Mukherjee, Architecture Design for Soft Errors, Morgan Kaufmann Publishers Inc., 2008, ISBN: 978-0123695291.

[4] B. Döbel, H. Härtig, Can we put concurrency back into redundant multithreading?, in: Proceedings of the International Conference on Embedded Software (EMSOFT), ACM, 2014, pp. 1–10. doi:10.1145/2656045.2656050.

[5] C. Bienia, Benchmarking modern multiprocessors, Ph.D. thesis, Princeton University (2011).

[6] Xilinx, Inc., Microblaze processor reference guide (Accessed: 2021-02-24).
URL https://www.xilinx.com/support/documentation/sw_manuals/
xilinx2019_2/ug984-vivado-microblaze-ref.pdf

[7] R. Amslinger, C. Piatka, F. Haas, S. Weis, T. Ungerer, S. Altmeyer,
Hardware multiversioning for fail-operational multithreaded applications,
in: 2020 IEEE 32nd International Symposium on Computer Architecture
and High Performance Computing (SBAC-PAD), IEEE, 2020, pp. 20–27.
doi:10.1109/SBAC-PAD49847.2020.00014.

[8] A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, D. A. Connors, Us-
ing process-level redundancy to exploit multiple cores for transient fault
tolerance, in: 37th Annual IEEE/IFIP International Conference on De-
pendable Systems and Networks (DSN'07), IEEE, 2007, pp. 297–306.
doi:10.1109/DSN.2007.98.

[9] R. Amslinger, S. Weis, C. Piatka, F. Haas, T. Ungerer, Redundant ex-
ecution on heterogeneous multi-cores utilizing transactional memory, in:
Architecture of Computing Systems (ARCS), Springer, 2018, pp. 155–167.
doi:10.1007/978-3-319-77610-1_12.

[10] G. Yalcin, O. S. Unsal, A. Cristal, Fault tolerance for multi-threaded appli-
cations by leveraging hardware transactional memory, in: Proceedings of
the ACM International Conference on Computing Frontiers (CF'13), ACM,
2013, pp. 1–9. doi:10.1145/2482767.2482773.

[11] F. Haas, S. Weis, T. Ungerer, G. Pokam, Y. Wu, Fault-tolerant execution on
cots multi-core processors with hardware transactional memory support, in:
International Conference on Architecture of Computing Systems (ARCS),
Springer, 2017, pp. 16–30. doi:10.1007/978-3-319-54999-6_2.

[12] D. Kuvaiskii, R. Faqeh, P. Bhatotia, P. Felber, C. Fetzer, Haft: Hardware-
assisted fault tolerance, in: Proceedings of the Eleventh European Con-
ference on Computer Systems (EuroSys'16), ACM, 2016, pp. 1–17. doi:
10.1145/2901318.2901339.

[13] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, P. M. Chen, Execution
replay of multiprocessor virtual machines, in: Proceedings of the Fourth
ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments
(VEE 08), ACM, 2008, pp. 121–130. doi:10.1145/1346256.1346273.

[14] S. Ren, L. Tan, C. Li, Z. Xiao, W. Song, Samsara: Efficient deterministic
replay in multiprocessor environments with hardware virtualization exten-
sions, in: Proceedings of the 2016 USENIX Annual Technical Conference
(USENIX ATC 16), USENIX Association, 2016, pp. 551–564.

[15] P. A. Bernstein, N. Goodman, Multiversion concurrency control–theory
and algorithms, ACM Trans. Database Syst. 8 (4) (1983) 465–483. doi:
10.1145/319996.319998.

[16] The PostgreSQL Global Development Group, Postgresql: Documentation: 12: 13.1. introduction (Accessed: 2021-02-24).
URL `https://www.postgresql.org/docs/12/mvcc-intro.html`

[17] R. Amslinger, Loosely-coupled fail-operational execution on embedded heterogeneous multi-cores, Ph.D. thesis, Universität Augsburg (2021).

[18] B. H. Bloom, Space/time trade-offs in hash coding with allowable errors, Communications of the ACM 13 (7) (1970) 422–426. `doi:10.1145/362686.362692`.

[19] C. Piatka, R. Amslinger, F. Haas, S. Weis, S. Altmeyer, T. Ungerer, Investigating transactional memory for high performance embedded systems, in: International Conference on Architecture of Computing Systems (ARCS), Springer, 2020, pp. 97–108. `doi:10.1007/978-3-030-52794-5_8`.