Redundant Execution on Heterogeneous Multi-Cores Utilizing Transactional Memory

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

University of Augsburg, Germany
{rico.amslinger, sebastian.weis, christian.piatka, florian.haas, theo.ungerer}@informatik.uni-augsburg.de

Abstract. Cycle-by-cycle lockstep execution as implemented by current embedded processors is unsuitable for heterogeneous multi-cores, because the different cores are not cycle synchronous. Furthermore, current and future safety-critical applications demand fail-operational execution, which requires mechanisms for error recovery.
In this paper, we propose a loosely-coupled redundancy approach which combines an in-order with an out-of-order core and utilizes transactional memory for error recovery. The critical program is run in dual-modular redundancy on the out-of-order and the in-order core. The memory accesses of the out-of-order core are used to prefetch for the in-order core. The transactional memory system's checkpointing mechanism is leveraged to recover from errors. The resulting system runs up to 2.9 times faster than a lockstep system consisting of two in-order cores and consumes up to 35% less energy at the same performance than a lockstep system consisting of two out-of-order cores.

Keywords: fault tolerance, multi-core, heterogeneous system, transactional memory, cache

1 Introduction

Heterogeneous multi-cores like ARM big.LITTLE™-systems [2] combine fast and complex (i. e. out-of-order) cores with slow and simple (i. e. in-order) cores to achieve both high peak performance and long battery life. While these architectures are mainly designed for mobile devices, modern embedded applications, e. g. those used for autonomous driving, also require high performance and power efficiency.

Additionally, these applications require high safety levels, as they are supported by current safety-critical lockstep processors [1,7,11]. However, cycle-by-cycle lockstep execution requires determinism at cycle granularity, because the core states are compared after every cycle. This strict determinism complicates the use of modern out-of-order pipelines, limits dynamic power management mechanisms [5], and also prevents the combination of a fast out-of-order core with an energy-efficient in-order core, even if both execute the same instruction

Cite as: Amslinger R., Weis S., Piatka C., Haas F., Ungerer T. (2018) Redundant Execution on Heterogeneous Multi-cores Utilizing Transactional Memory. In: Berekovic M., Buchty R., Hamann H., Koch D., Pionteck T. (eds) Architecture of Computing Systems - ARCS 2018. ARCS 2018. Lecture Notes in Computer Science, vol 10793. Springer, Cham.
https://doi.org/10.1007/978-3-319-77610-1_12

set. In contrast to lockstep execution, loosely-coupled redundant execution approaches [13,14,15], where the cores are not synchronized every cycle, allow the cores to execute more independently. As a cycle-based synchronization between the redundant cores is not necessary, resource sharing of parts of the memory hierarchy becomes possible. In that case, a heterogeneous dual-core may benefit from synergies between the cores, where a slower in-order core checks the results of a faster out-of-order core. In case an application does not need result verification, the redundant core can be switched off for energy savings or used as a separate unit for parallel execution.

Furthermore, current safety-critical lockstep cores only support fail-safe execution, since they are only able to detect errors. However, future safety-critical applications may additionally demand a fail-operational execution, which requires the implementation of recovery mechanisms. In this paper, we present a loosely-coupled fault-tolerance approach, combining a heterogenous multi-core with hardware transactional memory for error isolation and recovery. Its advantages are a more energy efficient execution than an out-of-order lockstep system, more performance than an in-order lockstep system, and less hardware and energy consumption than a triple modular redundant system.

Due to the loose coupling it is possible to combine different cores and to employ fault-tolerance on a heterogeneous dual-core. In this case, the out-of-order core can run ahead of the in-order core. This enables the leading (out-of-order) core to forward its information about memory accesses and branch outcomes to the trailing (in-order) core to increase its performance. Therefore, the approach provides a desirable trade-off between a homogeneous lockstep system consisting of either out-of-order or in-order cores as it is more power efficient or faster, respectively. The hardware cost for the implementation can be reduced by utilizing existing hardware transactional memory (HTM) structures. The HTM system provides rollback capabilities, which enable the system to make progress even if faults occur. The affected transactions are re-executed, until they succeed. No additional main memory is required, as the HTM system isolates speculative values in the caches. If a parallel workload does not require a fault-tolerant execution, the loose coupling can be switched off at run-time to benefit from the multi-core CPU and the transactional memory for multi-threading.

The contributions of this paper are: (1) A mechanism to couple heterogeneous cores for redundancy that speeds up the trailing core by forwarding data cache accesses and branch outcomes. (2) A design of a HTM for embedded multi-cores to support loosely-coupled redundancy with implicit checkpoints. (3) An evaluation of throughput and power consumption of our proposed heterogeneous redundant system compared to a lockstep processor.

The remainder of this paper is structured as follows. Related work is discussed in Section 2. Section 3 describes our redundant execution implementation. The baseline system is depicted first. Then our loose coupling and the rollback mechanism are explained. The following subsection describes the necessary changes to the HTM system. The last subsection specifies the advantages for heterogeneous systems. Section 4 contains a performance evaluation of several microbench-

marks. Our approach is compared to a lockstep system and a stride prefetching mechanism. The paper is concluded in Section 5.

2 Related Work

Reinhardt et al. [14] propose to use the simultaneous multithreading capabilities of modern processors for error detection. The program is executed twice on the same core. The executions are shifted and can use different execution units for the same instruction. It is proposed to maintain a constant slack to minimize memory stalls.

AR-SMT [15] is a time redundant fault-tolerance approach. An SMT-processor executes the same program twice with some delay. The execution state of the second thread is used to restore the first thread to a safe state if an error occurs.

The Slipstream Processor [17] is a concept which does not only provide fault tolerance, but also higher performance by executing a second, slimmer version of the program on either the same or another core. The second version of the program is generated by leaving out instructions which are predicted to be ineffective. The resulting branch outcomes and prefetches are used to accelerate the full version of the program. Errors are detected by comparing stores.

LBRA [16] is a loosely-coupled redundant architecture extending the transaction system LogTM-SE [19]. The old value for every memory location accessed by the leading thread is stored in a log. For writes the new value is immediately stored in memory. The trailing thread uses the log values for input duplication. Both cores calculate signatures for every transaction. If an error is detected, the log is traversed backwards to restore the old memory state.

FaulTM [18] is a fault-tolerance system utilizing transactional memory. Their approach differs from ours in that it executes redundant transactions synchronously. The write-sets and registers of both transactions are compared simultaneously, with one transaction committing to memory. This prohibits one thread to run ahead of the other and thus suppresses possible cache-related performance benefits.

Haas et al. [8,9] use the existing Intel HTM system (TSX) for error detection and recovery. As Intel TSX does not support automatic transaction creation or write set comparison, the protected software is instrumented to provide those features itself. Transactional blocks are executed with an offset in the trailing process, as TSX does not allow checksum transfer within active transactions. As the redundant processes use the same virtual addresses, but different physical memory locations, no speedups due to positive cache effects occur in the trailing process.

3 Transaction-Based Redundant Execution Model

The baseline multi-core architecture with HTM is shown in Figure 1 as an abstraction of a state-of-the-art heterogeneous multi-core. The architecture con-

sists of two different cores with private L1 data and instruction caches that are connected to a shared L2 cache. Cache coherence is guaranteed by hardware.

Hardware facilities support the execution of transactions, similar to Intel Haswell [10]. The register sets of the cores are extended to provide snapshot capabilities. The data caches and the coherence protocol are enhanced to manage the read and write sets. The affected cache blocks cannot be evicted during the transaction. The instruction caches do not require read and write sets since self-modifying code is not supported. Transactional conflicts are detected by an extended cache-coherence protocol.

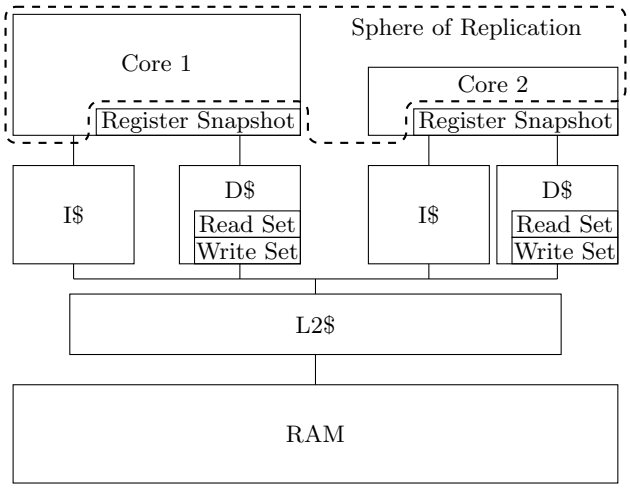

Fig. 1. Baseline multi-core architecture, enhanced to support hardware transactions.

3.1 Loosely-Coupled Redundancy with Checkpoints

The proposed fault-tolerance mechanisms can detect and recover from faults that occur in the pipelines of the cores, which is shown by the Sphere of Replication in Figure 1. While there are two instances of the L1 caches and register sets, a distinct fault-tolerance mechanism like ECC is still required for all caches and the registers sets to ensure the consistency of the generated checkpoints.

Figure 2 shows the redundant execution approach. The redundant system proposed in this paper executes the same code on both cores. The execution on the cores is shifted by a dynamic time offset, called slack. Both cores will start execution at the same instruction, when redundancy is enabled. Later the out-of-order core is running ahead of the in-order core, except for I/O operations or error recovery, that require synchronization of the cores to be resolved. As the leading core usually runs faster, the slack increases. For implementation reasons

like buffer sizes the slack is limited to a certain number of instructions. This hard limit will not be hit often, as accelerating effects for the trailing core like forwarding memory accesses will become more effective with increasing slack.

To enable recoverability, checkpoints are automatically created by the cores, when a transaction is started. The cores will automatically start and commit transactions in a way that minimizes transaction and comparison overhead, while ensuring that they fit in the cache. Instrumentation of the program with explicit transaction instructions is not required.

While the trailing core only keeps the last checkpoint, the leading core must retain two. This enables the rollback after an error, regardless of the core it occurs on. If the leading core gets more than two checkpoints ahead (e. g. after TX2), it has to wait. As the HTM system uses the caches to hold unverified data, the cache size also limits the slack. When the trailing core reaches a checkpoint, the current state is compared to the corresponding checkpoint of the leading core. The new checkpoint for the trailing core is only created after a successful comparison. The leading core on the other hand can speculatively progress as long as it keeps at least one confirmed checkpoint.

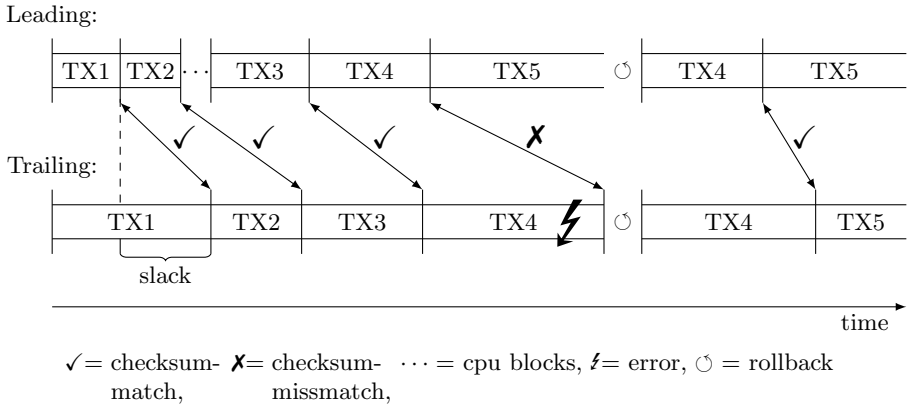

Fig. 2. Redundant execution with transactions.

Unconfirmed data is never written back to memory or transferred to another cache. Only after confirming correct execution, the trailing core writes modified data back to memory or marks it as valid for the cache coherence protocol. The leading core's cache silently evicts modified cache lines that were already verified instead of writing them back. It relies on getting their data back from memory or by cache to cache transfer from the trailing core. As none of the caches can evict data written after active checkpoints, the cache size clearly limits the distance between subsequent checkpoints.

Figure 2 also shows the handling of an error. After the error occurs, the next comparison after TX4 results in a mismatch. The leading core has already advanced past the potential erroneous checkpoint, but is still keeping an older confirmed checkpoint at the start of TX4. Thus both cores rollback to the start of TX4. As all changes after the confirmed checkpoint are confined in the cores and their L1 caches, the rollback is fast. If the fault was transient, the next comparison will succeed and the execution resumes regularly.

3.2 Extension of HTM to Support Fault Tolerance

If the processor already includes support for HTM to speculatively speed up parallel execution, the implementation of the redundant system can be simplified. The HTM system already provides support for isolation and checkpointing. Thus those components can simply be reused. As the leading core requires multiple checkpoints, the checkpointing capabilities of some simple transaction systems may be insufficient.

The conflict detection component of HTM is unnecessary for redundant execution on a dual-core, as both cores execute the same thread. The obvious approach is to disable it completely, in order to avoid detection of false conflicts between the leading and trailing core. The commit logic of the leading core can be simplified, as writeback of confirmed data is handled by the trailing core.

Some additions are still required to support full fault tolerance. First, HTM usually relies on special instructions to start and end transactions. For fault tolerance another approach is preferred: The transaction boundaries should be determined automatically at run-time, as it is hard to predict exact cache usage at compile time. Second, regular HTM systems do not care about the content of written cache blocks. Only their addresses need to be compared for conflict detection. It is thus necessary to implement an additional unit to compare registers and cache block content. This unit will also need to restart both cores at the correct checkpoints when a error has been detected.

3.3 Heterogeneous Redundant Systems

Tightly-coupled redundancy approaches like lockstep execution are not applicable when heterogeneous cores are employed. Once a core executes an instruction faster than the other core, a false error will be detected, causing the abort of the application or a costly recovery attempt. Loosely-coupled redundant execution does not suffer from the disadvantage of false positives caused by different microarchitectural implementations.

If the slack is sufficiently large, a cache miss is detected in the leading core before the trailing core even reaches the instruction that causes the fetch. Thus the leading core's memory access addresses can be forwarded to the trailing core, so it can prefetch them. This increases the performance, as cache misses are often more expensive for simpler cores. Since the total run-time of the system is determined by the slower core, this optimization improves total system performance.

The trailing core can also benefit from other information. Even simple in-order cores like the ARM Cortex-A7 feature branch prediction. As the cores' data structures used for branch prediction are smaller than those of complex cores, mispredictions are more common. These mispredictions can be eliminated by forwarding branch outcomes from the leading core to the trailing core by using a branch outcome queue [14]. This requires the leading core to stay far enough ahead, so that it can retire the branch before the trailing core decodes it. If the leading core is sufficiently fast, all branches in the trailing core are predicted correctly. Thus, the performance improves in programs with many data dependent branches. Error detection is not impacted, as the trailing core will interpret different branch outcomes as mispredict.

With increasing differences between the cores, the implementation of such enhancements becomes more difficult. For instance, a complex core may replace short branches with predicated execution [12]. Thus the branch will not reach the core's commit stage. As a result the trailing core will not find a corresponding entry in the branch outcome queue, when it reaches the branch. Such problems can cause the cores to lose synchronization and therefore decrease performance, as shifted branch outcomes can be more inaccurate than the trailing core's regular branch prediction.

4 Evaluation

The suggested approach was modeled in Gem5 [6]. The fast and slow cores were configured to match the ARM Cortex-A15 and ARM Cortex-A7, respectively. Power consumption was approximated as the product of the simulated benchmark run-time and the average power consumption of an Exynos 5430 SoC. It was assumed that a lockstep system runs as fast as a corresponding single-core machine, but consumes twice the energy. For our approach, a limit was put in place to prevent the leading core from running too far ahead. The leading core stalls when it has committed 1,000 instructions more than the trailing core, or if it tries to evict a modified cache line that the trailing core has not written yet.

We implemented several sequential microbenchmarks with different memory access patterns. The following microbenchmarks were used to assess the performance of the approach:

- The *breadth-first* benchmark calculates the size of a tree by traversing it in breadth-first order. Each node has a random number of children.
- The *heapsort* benchmark sorts an array using heapsort. The array is initialized with random positive integers.
- The *matrixmul* benchmark calculates the product of two dense matrices.
- The *quicksort* benchmark sorts an array using quicksort. The array is initialized with random positive integers.
- The *red-black tree* benchmark inserts random entries into a red-black tree.
- The *shuffle* benchmark shuffles an array using the Fisher-Yates shuffle.

The seed of the random number generator was constant for all runs.

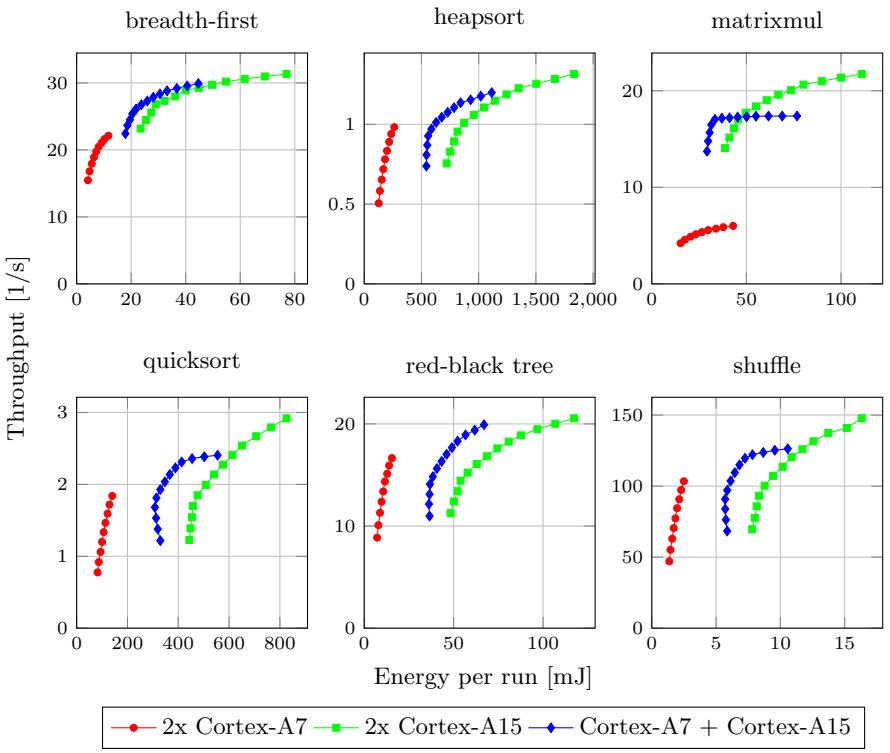

Fig. 3. Trade-off between throughput and power consumption

Figure 3 shows the microbenchmarks' throughput on the y-axis and the corresponding energy consumption per run on the x-axis. The microbenchmarks were executed on a lockstep system consisting of two Cortex-A7, another lockstep system consisting of two Cortex-A15 and our approach, using a Cortex-A15 as leading core and a Cortex-A7 as trailing core. To observe the maximum effect, the integrated prefetchers were disabled. The cores' clock frequency were varied in their frequency ranges (Cortex-A7: 500-1300 MHz, Cortex-A15: 700-1900 MHz) in 100 MHz steps. For our approach the trailing core's frequency was fixed at 1300 MHz, while the leading core's frequency was varied from 700 MHz to 1900 MHz.

At first only a small increase in voltage per 100 MHz step is required to allow the core to run stable. Thus for the lockstep systems, a large increase in throughput can be achieved at low frequencies by a small increase in power consumption. Note that the frequency itself has only minor influence on the results, as power consumption is measured per benchmark run and not per time unit. When the core's approach their maximum frequency, the required increase in voltage raises. At the same time the achieved acceleration decreases, as the

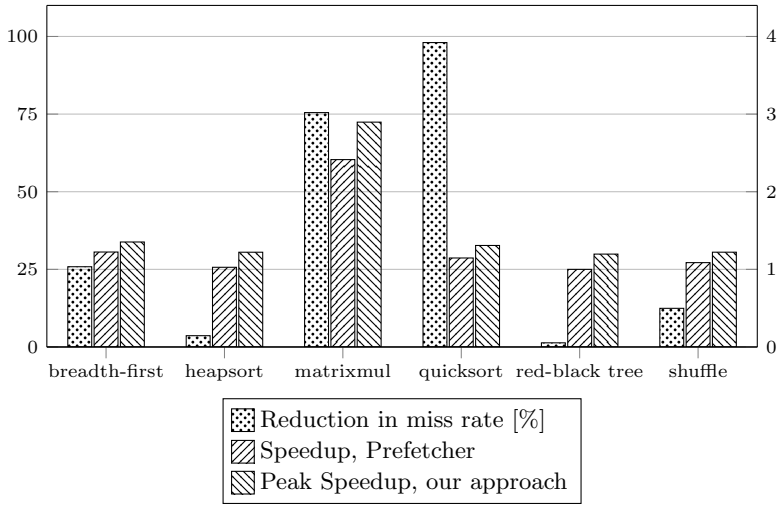

Fig. 4. Performance gain by adding a stride prefetcher.

memory clock frequency remains constant. Thus for high frequencies only a small increase in throughput can be achieved by a large increase in power consumption. The effect is more pronounced on the Cortex-A15, as its IPC and maximum frequency are higher.

Our approach shows a different pattern. For the frequency range, in which the out-of-order core's performance does not exceed the in-order core's performance at maximum frequency, the leading core slows down the entire system. Increasing the leading core's frequency, can reduce total power consumption (e. g. in *quicksort* or *shuffle*), as the task finishes quicker, thus reducing the time the trailing core is running at maximum voltage. After the leading core's performance exceeds the trailing core's, there is a phase in which the trailing core can be accelerated to the leading core's level by prefetching. This area is the most interesting as it offers higher performance than the trailing core, at a lower power consumption than a lockstep system of two out-of-order cores. If the leading core's frequency is increased further, eventually a point will be reached, at which every memory access is prefetched. The graph asymptotically approaches this performance level. As the leading core's power consumption still raises, the combination will eventually consume more energy than a lockstep system consisting of two out-of-order cores would at the same performance level (apparent in *matrixmul*). Thus further increasing the frequency of the leading core should be avoided.

Figure 4 shows the performance improvement achieved when enabling the stride prefetcher [4] on the Cortex-A7, which was clocked at 1300 MHz. Obviously the prefetcher by itself does not provide fault tolerance. The Cortex-A7 utilizes early issue of memory operations [3] in all variants. The stride prefetcher

tries to detect regular access patterns on a per instruction basis. If an instruction accesses memory locations with a constant distance, the prefetcher will predict the next addresses and preload them into the L1 cache. As the stride prefetcher works on physical addresses, a detected stream will be terminated at the next page boundary. For this evaluation the prefetcher was configured to observe 32 distinct instructions on a LRU basis and prefetch up to 4 cache lines ahead of the last actual access. The amount by which the L1 cache miss rate (left y-axis) was reduced and the total speedup (right y-axis) were measured. For comparison the peak speedup (right y-axis) achieved by our approach was also included.

The effectiveness of the stride prefetcher varies depending on the benchmark's memory access pattern. For benchmarks with regular access pattern like *matrix-mul* most cache misses can be eliminated. The first matrix is accessed linearly, so nearly all cache lines are prefetched. As the second matrix is not transposed, the stride prefetcher can only prefetch a few cache lines before encountering a page boundary. Our approach can perfectly prefetch both matrices. *Shuffle* accesses the locations for the swap target at random. Thus, the stride prefetcher is unable to predict the next access. However, forwarding the addresses from the leading core to the trailing core is possible, as both cores use the same seed.

Tree-based benchmarks like *breadth-first* or *red-black tree* show a very irregular memory access pattern. They do not benefit as much from a stride prefetcher, as it will rarely detect consistent strides when traversing the tree. However, it can improve performance for the queues and stacks used in those algorithms, as those show a regular access pattern. An overly aggressive prefetcher may reduce performance for such algorithms, as it evicts cache lines that will be reused for false prefetches. Our approach on the other hand can eliminate all cache misses even for such irregular patterns, as long as the leading core runs fast enough. Thus the resulting speedup exceeds, what is achievable with a simple prefetcher.

Our approach can achieve higher speedups than the stride prefetcher for both sorting algorithms. However the reasons differ. *Heapsort* shows an irregular access pattern, as the heap is tree-based. Thus, our approach benefits from superior prefetching performance. *Quicksort* on the other hand shows a very regular access pattern, as it linearly accesses elements from both directions. However, *quicksort* uses data dependent comparisons as loop condition in the Hoare partition scheme. Regular branch predictors can not predict those branches, as they are essentially random for random data. However, in our approach the trailing core can use the forwarded branch outcomes from the leading core to further increase performance.

5 Conclusion

Loosely-coupled redundant execution with transactional memory to support checkpointing has the potential to be an alternative to current lockstep systems. As the HTM system already provides mechanisms like isolation and checkpointing, the required hardware enhancements are small. The isolation allows both cores to operate on the same memory region, while the checkpointing mecha-

nism enables error recovery even with just two cores. The loose coupling makes it possible to use the approach in heterogenous multi-cores.

The evaluation of the proposed approach showed that a slower in-order core is able to keep up with a faster out-of order core to provide redundancy. This requires a near-optimal data prefetching in the trailing core, which is achieved by forwarding the memory accesses of the leading core. Supplying branch outcomes further increases the throughput of the slower core. The combination of heterogeneous cores for redundant execution results in a good trade-off between performance and power consumption. It offers up to 2.9 times the performance and up to 35% less power consumption than comparable lockstep systems consisting of only slow or fast cores, respectively. Additionally, flexible coupling of cores improves the flexibility for parallel applications with varying fault-tolerance requirements.

As future work, we plan to extend our approach to larger heterogeneous multi-cores, which will enable to change the coupling of cores dynamically at runtime. Programs that exhibit a sufficient amount of cache misses benefit from a heterogeneous coupling, since the in-order trailing core will be accelerated by the cached data of the leading core. Otherwise, homogeneous coupling is preferred for compute intensive programs to deliver better performance. Further, we plan to extend the approach to support multi-threaded applications, regardless of the synchronization mechanism they use.

References

1. ARM Ltd.: Cortex-R5 and Cortex-R5F - Technical Reference Manual. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0460c/DDI0460C_cortexr5_trm.pdf` (2011), Revision r1p1
2. ARM Ltd.: big.LITTLE Technology: The Future of Mobile (2013), `https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf`
3. Austin, T.M., Sohi, G.S.: Zero-cycle loads: Microarchitecture support for reducing load latency. In: Proceedings of the 28th annual international symposium on Microarchitecture. pp. 82–92 (1995)
4. Baer, J.L., Chen, T.F.: An effective on-chip preloading scheme to reduce data access penalty. In: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing. pp. 176–186. Supercomputing '91, ACM (1991)
5. Bernick, D., Bruckert, B., Vigna, P., Garcia, D., Jardine, R., Klecka, J., Smullen, J.: NonStop® Advanced Architecture. In: International Conference on Dependable Systems and Networks (DSN). pp. 12–21 (2005)
6. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., et al.: The Gem5 Simulator. ACM SIGARCH Computer Architecture News 39(2), 1–7 (2011)
7. Freescale Semiconductor: Safety Manual for Qorivva MPC5643L (2013), `https://www.nxp.com/docs/en/user-guide/MPC5643LSM.pdf`
8. Haas, F., Weis, S., Metzlaff, S., Ungerer, T.: Exploiting Intel TSX for fault-tolerant execution in safety-critical systems. In: IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT). pp. 197–202 (2014)

9. Haas, F., Weis, S., Ungerer, T., Pokam, G., Wu, Y.: Fault-tolerant Execution on COTS Multi-core Processors with Hardware Transactional Memory Support. In: 30th International Conference on Architecture of Computing Systems (ARCS) (2017)
10. Hammarlund, P., Martinez, A.J., Bajwa, A.A., Hill, D.L., Hallnor, E., Jiang, H., Dixon, M., Derr, M., Hunsaker, M., Kumar, R., et al.: Haswell: The Fourth-Generation Intel Core Processor. IEEE Micro 34(2), 6–20 (2014)
11. Infineon Technologies AG: Highly integrated and performance optimized 32-bit microcontrollers for automotive and industrial applications (2017), https://www.infineon.com/dgdl/Infineon-TriCore-Family_2017-BC-v02_00-EN.pdf?fileId=5546d4625d5945ed015dc81f47b436c7
12. Klauser, A., Austin, T., Grunwald, D., Calder, B.: Dynamic hammock predication for non-predicated instruction set architectures. In: International Conference on Parallel Architectures and Compilation Techniques (PACT). pp. 278–285 (1998)
13. LaFrieda, C., Ipek, E., Martinez, J., Manohar, R.: Utilizing Dynamically Coupled Cores to Form a Resilient Chip Multiprocessor. In: 37th International Conference on Dependable Systems and Networks (DSN). pp. 317–326 (2007)
14. Reinhardt, S.K., Mukherjee, S.S.: Transient Fault Detection via Simultaneous Multithreading. In: 27th Annual International Symposium on Computer Architecture (ISCA). pp. 25–36. ACM (2000)
15. Rotenberg, E.: AR-SMT: a microarchitectural approach to fault tolerance in microprocessors. In: 29th International Symposium on Fault-Tolerant Computing (FTCS). pp. 84–91 (1999)
16. Sánchez, D., Aragón, J., Garcıa, J.: A log-based redundant architecture for reliable parallel computation. In: International Conference on High Performance Computing (HiPC) (2010)
17. Sundaramoorthy, K., Purser, Z., Rotenberg, E.: Slipstream processors: Improving both performance and fault tolerance. ACM SIGPLAN Notices 35(11), 257–268 (2000)
18. Yalcin, G., Unsal, O., Cristal, A.: FaulTM: error detection and recovery using hardware transactional memory. In: Conference on Design, Automation and Test in Europe (DATE). pp. 220–225 (2013)
19. Yen, L., Bobba, J., Marty, M.R., Moore, K.E., Volos, H., Hill, M.D., Swift, M.M., Wood, D.A.: LogTM-SE: Decoupling hardware transactional memory from caches. In: IEEE 13th International Symposium on High Performance Computer Architecture (HPCA). pp. 261–272 (2007)