Fault-tolerant Execution on COTS Multi-core Processors with Hardware Transactional Memory Support

Florian Haas[1], Sebastian Weis[1], Theo Ungerer[1],
Gilles Pokam[2], and Youfeng Wu[2]

[1] Department of Computer Science, University of Augsburg, Germany
{haas,weis,ungerer}@informatik.uni-augsburg.de
[2] Intel Corporation, Santa Clara, USA
{gilles.a.pokam,youfeng.wu}@intel.com

Abstract. The demand for fault-tolerant execution on high performance computer systems increases due to higher fault rates resulting from smaller structure sizes. As an alternative to hardware-based lockstep solutions, software-based fault-tolerance mechanisms can increase the reliability of multi-core commercial-of-the-shelf (COTS) CPUs while being cheaper and more flexible. This paper proposes a software/hardware hybrid approach, which targets Intel's current x86 multi-core platforms of the Core and Xeon family. We leverage hardware transactional memory (Intel TSX) to support implicit checkpoint creation and fast rollback. Redundant execution of processes and signature-based comparison of their computations provides error detection, and transactional wrapping enables error recovery. Existing applications are enhanced towards fault-tolerant redundant execution by post-link binary instrumentation. Hardware enhancements to further increase the applicability of the approach are proposed and evaluated with SPEC CPU 2006 benchmarks. The resulting performance overhead is 47 % on average, assuming the existence of the proposed hardware support.

1 Introduction

Errors in computer systems can never be avoided completely and the field of application dictates the required counter-measures. With error detection, erroneous computations can be identified. However, re-execution is required, inducing down time for error correction and system restart. Dependable server systems are usually designed as *fail-stop systems*, i. e. erroneous execution is detected and the system is stopped or restarted. In contrary, *fail-operational systems* provide built-in error correction and thus can continue to operate correctly in case of errors. While current processors are already protected against faults on different memory levels with error correcting codes (ECC), transient faults can still occur within the data and control paths of the processor's pipelines. Processors often implement tightly coupled lockstep execution to detect transient faults, which requires a complete duplication of the hardware resources and cycle-by-cycle

synchronization. However, the integration of hardware-based lockstep mechanisms in up-to-date COTS multi-core processors is complex and would require deep changes to the microarchitecture, since COTS processors also implement various power management and performance optimization mechanisms, which complicate synchronization at a cycle-by-cycle granularity [1]. Furthermore, current dual-modular redundant (DMR) lockstep processors only detect faults and therefore only support fail-stop execution. Triple-modular redundancy (TMR) is required to provide fault tolerance through forward-error correction. While hardware fault-tolerance mechanisms require complex and costly modifications to the microarchitecture of state-of-the-art COTS processors, pure software-based fault tolerance techniques, which duplicate instructions or processes to detect faults, e.g. [13, 14], usually have a high performance impact, require specific compilers, and support limited recovery capabilities.

Hardware Transactional Memory (HTM) was first proposed for concurrency control [7] and is able to increase the parallelism and ease the programmability of parallel applications. Intel introduced the *Transactional Synchronization Extensions* (TSX) as part of the Haswell instruction set architecture [5]. The rollback mechanism of TSX, where all modified data from within a transaction is restorable, can be utilized for fault tolerance. A rollback to the implicitly created checkpoint helps to recover from detected errors. In this paper, we propose a software/hardware hybrid fault-tolerance mechanism to lift check-point restart systems from fail-stop to fail-operational by fast and fine-grained checkpoint generation and restart using Intel TSX. We leverage the recovery capabilities of TSX to support fault-tolerant execution of arbitrary, single-threaded applications. Our approach combines redundant execution of processes to detect differences in their executions, and backward error recovery by restarting transactions in the case of detected errors. Modifications of the existing hardware to facilitate error detection can increase the performance of our approach. We therefore evaluated the benefits to be expected by hardware support for signature generation and exchange. Our approach is based on binary instrumentation to achieve fault-tolerant execution. Post-link binary instrumentation allows the fault-tolerant execution of existing and already compiled programs on POSIX-compatible platforms.

We proposed the idea of leveraging Intel TSX for recoverability on existing hardware in [3, 4]. A software-based implementation was presented to show the general applicability of Intel TSX for checkpointing in redundant processes. In this first approach, source code modifications were required and no full coverage of the instrumented application was achieved. This paper makes the following new contributions: (1) We present a software/hardware hybrid fault-tolerance approach, which uses redundant process execution and post-link binary instrumentation to detect faults, and exploits the rollback-capabilities of Intel TSX to support efficient and low-overhead checkpointing and error recovery. (2) We propose hardware enhancements, which can significantly speed up our fault-tolerance mechanism. (3) We present a detailed evaluation of our approach for SPEC2006 integer and floating point benchmarks and show a performance

overhead of 47 % on average, assuming the existence of the proposed hardware support.

The rest of this paper is organized as follows. Section 2 provides a short overview of related work. The concept of our approach to detect errors through redundant execution and the recovery mechanism are described in Section 3. Possible hardware modifications and enhancements are discussed in Section 4. A detailed evaluation is presented in Section 5. Limitations and future work are discussed in Section 6. The paper is concluded in section 7.

2 Related Work

Various methods for fault-tolerant execution on different types of processors exist. The most well-known method is lock-stepping [11], where two cores are coupled tightly. Comparison of the equality of both cores' computations happens on cycle or instruction level. Loosely coupled lockstep architectures support SMT [12] and multi-core processors [9]. However, to support error detection and recovery in such systems, complex changes are required to the pipeline and the memory hierarchy. This renders the applicability of such approaches on current high-performance COTS processors difficult and costly. As an alternative to hardware-based approaches, different software-only fault-tolerance mechanisms exist. They need to modify the program, either after compilation by binary instrumentation, or during compilation. A well-known compiler-based approach is SWIFT [13], where instructions are duplicated within the same process and a comparison of both results ensures correct execution of single-threaded applications. Unlike our approach, SWIFT cannot guarantee error isolation among dual executed streams, since memory protection works only between individual processes. A technique that uses process level redundancy was introduced with PLR [14] where a whole single-threaded process is duplicated. On system call granularity, both processes are compared. However, these software-based approaches cannot recover from detected errors, or impose high cost on checkpoint creation and recovery.

Transactional memory originated in high performance computing to allow optimistic synchronization. As the first commercial hardware implementations became available, the application of hardware transactional memory in embedded systems was suggested [2]. Beside the possibility of increased performance by optimistic synchronization in COTS-based dependable systems, the authors encourage the use of implicit checkpointing and rollback mechanism for fast and easy recovery in case of detected errors. Yalcin et al. [15] implemented a custom hardware transactional memory to support fault-tolerant execution of multi-threaded applications. The extendability of this transactional memory system allows execution of redundant transactions in parallel and comparison of signatures in hardware. In contrary to custom hardware implementations of transactional memory with focus on fault tolerance, our paper is based on Intel TSX, an existing, performance-oriented hardware transactional memory. As an alternative to full redundant execution, instruction level redundancy [8] allows error detection without process duplication. Instead, instructions are replicated

to repeat the computation on different registers. Before writing to memory and at basic block boundaries, the data is compared to detect errors. This approach instruments applications during compilation, and also uses TSX transactions for checkpointing and rollback.

This paper presents a comprehensive evaluation of the performance overhead of redundant process execution on the Intel Haswell architecture. Furthermore, we propose hardware enhancements to increase the performance of our approach. The instrumentation mechanisms provide an increased code coverage with less performance overhead than other approaches. Combining the flexibility of software-based redundant execution with existing hardware transactional memory allows efficient checkpointing and rollback for specific applications, or selected critical parts of applications, for fault-tolerant execution.

3 Error Detection

We target a fault model where transient errors in the cores can be tolerated, and permanent errors can be detected. Transient faults (also known as single event upsets) induced for example by environmental radiation, electromagnetic interferences, or voltage fluctuations [11], occur sporadically and can lead to the so-called soft errors. We assume that all memory is protected with ECC, but errors may still appear in the CPU itself. The main objective of our approach is to ensure correct execution even when soft errors occur. For this, fail-operational execution requires error detection, error recovery, and error containment. We use the error containment and recovery capability of Intel TSX to provide fail-operational execution. Our approach consists of an enhanced binary instrumentation tool and a dynamic library, which provides the required functionality for redundant execution, comparison and rollback. No modifications to the source code of the application are required. The instrumentation enhances the application's binary with signature generation instructions to provide error detection. The remaining functionality for process management, signature comparison, and error recovery is implemented in a library to keep the required instrumentation of the original binary minimal.

3.1 Redundant Execution

The main principle of our approach is to redundantly execute user processes. The loosely coupled redundant execution allows error detection by repeatedly comparing both processes through signatures of blocks based on function boundaries. With the encapsulation of these blocks in transactions, error recovery is implemented. Our approach is based on process level redundancy, which results in two mostly identical processes with the same virtual address space. The advantage of process redundancy in contrary to redundant threads is that the virtual memory management of the operating system guarantees physical memory isolation and thus prevents error propagation from a process to its duplicate. Also, since virtual addresses are equal in both processes, no modifications of memory addresses

are required. Existing programs are enhanced with error detection and recovery capabilities by modifying the program's binary with help of the instrumentation tool PEBIL [10]. Some of the required functionality is implemented in a library, which reduces the amount of code to instrument directly. To setup redundant execution, the binary instrumentation tool inserts a call to the library's setup function at the very beginning of the program, which usually is before calling `main`. Another call to end redundant execution is inserted into the exit code. The output of the instrumentation is a new binary ready for fault-tolerant execution on a common Intel CPU with support for TSX.

To enable redundant execution, the application's process is duplicated in the library's setup code by invoking a `fork` system call. The program then executes a *leading* and a *trailing* process. Both processes execute the same program code and also have identical virtual address spaces, which are physically separated by the memory management unit. Since the instrumentation is done on the binary and the process duplication takes place at run-time, the executed code of both processes is almost identical. As a consequence, the instrumentation requires no awareness of the actual process that executes the instrumented code. The distinction of the processes happens only in the library functions by means of the value of a global variable. Comparison of both processes is done on the level of function-based blocks, or also called *dependable blocks*. For an efficient comparison, we create signatures of these dependable blocks, which are exchanged between both processes. Fig. 1 shows a schematic of the redundant execution and block-based comparison. A process is duplicated at its entry point before calling `main`. All code within the program is executed redundantly (step 1). Binary instrumentation divides this code into blocks, which are aligned at function boundaries (step 2). Signatures of these blocks are calculated in each process and transferred from the leading process to the trailing process (step 3). There, signatures are compared to detect errors. We use a FIFO queue in shared memory

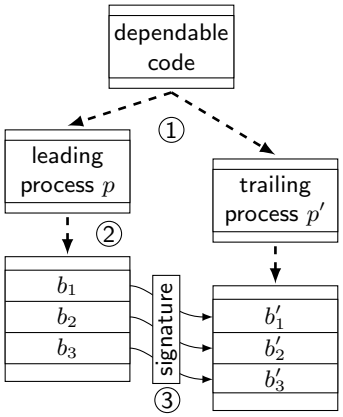

Fig. 1. Instrumentation on function-based dependable blocks b_n and redundant execution in processes p and p'.

between both processes for signature exchange, which implicitly enforces the correct ordering between the leading and the trailing process.

3.2 Signature Generation

To compare both executions of redundant blocks, we implemented a signature-based approach to integrate all modified data within a block. For signature creation we selected 32 bit CRC, which is available as the `crc32` instruction in the Haswell ISA. This allows a fast calculation with guaranteed detection of single bit flips. To create a signature, all registers of a dependable block are examined by the instrumentation tool and instructions are inserted for all used or defined registers to calculate a signature of the register content. If the same register is written multiple times inside a single block, it is sufficient to include the last value of the register in the signature. This is due to the fact that erroneous intermediate values either are propagated to other registers that are also added to the signature or the erroneous value is transitive over multiple operations (for example additions with same source and destination). Any register that is redefined inside the block would need to have the last value before the redefinition to be accumulated into the signature. A special case involves the memory write operations, since afterwards, their used registers are free to be newly assigned. In this case, data and addresses of the memory operation could be lost at the end of the block. As a consequence, memory write operations are immediately followed by their signature accumulation instructions. Further, floating-point operations are supported, and their used registers are also added to the signature. Since these instructions work on floating-point registers, an additional step is required, where we copy chunks of 64 bits into a general purpose register before calculating the CRC signature. Fig. 2 shows an example instrumentation with two memory instructions and three ALU instructions. Register `R15` is used to accumulate the CRC signature, while register `R14` holds the signature of the other process for comparison. If these registers are already in use by the program code of the current block, two other free registers are used. A block is split in case no free

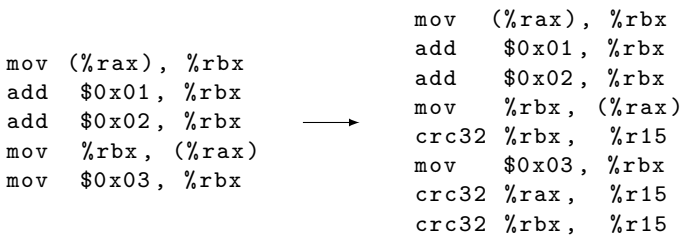

Fig. 2. Instrumentation of five instructions: used registers are accumulated by the hashing function `crc32` at the end of the block. Memory write operations require an immediate signature accumulation (see 5th line in code listing on the right).

registers can be found for storing the signatures. Splitting is repeated until at least two free registers are available.

3.3 Signature Exchange

Error detection requires the comparison of the redundant processes' signatures. Unfortunately, exchanging the signature within a transaction always results in an abort due to conflicting accesses. To overcome this, the signatures must be exchanged before the transaction is started in the trailing process. As a consequence, the process without transactions has to execute its block completely before the corresponding block of the transaction-enhanced process can be executed. In between, the signature is exchanged through a FIFO queue implemented in shared memory. The process that writes its signature to the queue is named *leader*, since its execution is at least one block ahead. The other process is named *trailer*. The trailer reads the leader's signature from the FIFO queue, stores it in a reserved register, and executes its block transactionally (see Fig. 3). Then the transaction is started, the block's code is executed and the signature is calculated. Before committing the transaction, both signatures are compared. In case of a mismatch, the transaction will be aborted. The use of a FIFO queue instead of a simple signature exchange buffer is beneficial for performance. The trailing process is slower, due to additional instructions for transactions and signature comparison. The FIFO queue now allows the leader to run ahead a fixed number of blocks, which is determined by the size of the queue. If writes from the leader process and reads from the trailer process are separated, cache line collisions vanish.

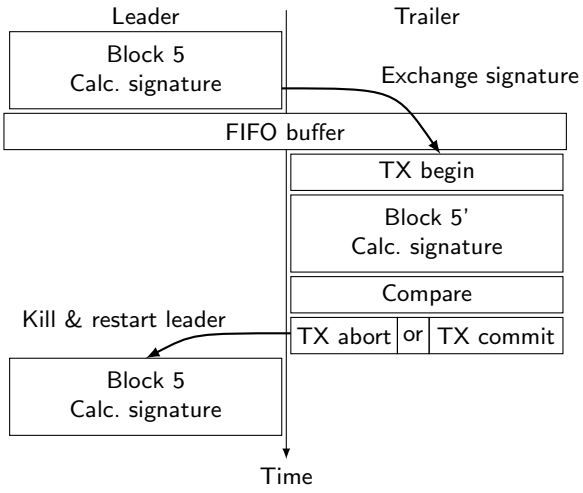

Fig. 3. Detailed view on the interlaced execution of both redundant processes. The calculated signature is exchanged through a FIFO buffer and compared with the local signature of the trailing process. Mismatches lead to transaction rollback of the trailing process and restart of the leading process.

3.4 Error Recovery and Containment

Error detection takes place at the end of the transactional block in the trailing process. Before committing the transaction, the locally calculated signature and the leading process' signature are compared. In case of a detected error, the transaction is explicitly aborted. Afterwards, the trailing process starts over at the beginning of the block, which previously was hit by the transient error. However, the leading process is at least one block ahead and not able to roll back to a point before the execution of the erroneous block. To overcome this, the leading process is killed by means of the operating system and a `fork` creates a new leading process. Also, the FIFO queue is cleared, since the signatures, which have already been produced, are probably faulty. The program execution then continues at the beginning of the previously faulty block. Fig. 3 shows the transaction abort after signature comparison, with termination and restart of the leading process, which then executes the same block again. After rollback and forking the new process, the execution is guaranteed to be error-free, since the transaction rolled back and the leading process is replaced by a copy of the correct process. Memory protection between both processes avoids error migration from one process to the other. All data that is written to memory by the trailing process within a block becomes visible only if both signatures are equal, since the signatures are compared inside of the transaction.

Transactions may not only abort explicitly due to detected errors. Overflowing cache, unfriendly instructions, or other external influences can cause a transaction to abort. If a transaction aborts, the abort handler checks the status and only if an explicit abort was forced due to a detected error, the error handler is called. Otherwise, the transaction is simply restarted.

4 Hardware Enhancements

The unmodifiable Haswell micro architecture impedes hardware enhancements, which can further improve our approach. Thus we investigated the potential impact of the following hardware-based enhancements.

4.1 Signature Generation

Since the most relevant bottleneck is the signature generation, our approach benefits from hardware signature generation. A possible hardware extension may calculate the signature implicitly, by issuing an accumulation on every read from a register and on every write to a register or to memory. The accumulation can happen in a dedicated register, which can be reset and read by the software. The instrumentation tool is then only required to reset the signature register at the beginning of a block. The comparison at the end of the block reads from this signature register instead of the reserved general purpose register.

4.2 Hardware Queue

Further improvement is possible by supporting signature exchange in hardware. This requires a mechanism to send data uni-directionally between individual cores, additionally with buffering to form a FIFO queue. It is sufficient to connect pairs of cores, a queue between every single core is not required. The assignment of processes to cores is handled by the software library.

4.3 Transactional Memory

Enhancements to TSX can also increase the performance and the versatility of our approach. Transactions should not abort in the error-free case, since conflicts do not occur. However, transactions are limited in their size, which depends on the cache, and are sensitive to interrupts and other system-related events. Also various instructions are not allowed to be executed inside of a transaction. Robust transactions surviving such events would allow a guaranteed transactional execution, resulting in a better coverage and less overhead for instrumented programs. Escape actions allow to read or write data in or out of a transaction without triggering a conflict. This enables parallel transactional execution of both redundant processes. In this case, signatures can be exchanged during the commit phase, and the transaction will only be allowed to commit if the signatures match. Otherwise, both transactions are rolled back and restarted.

5 Evaluation

The implementation of the concept described in the previous sections was evaluated by enhancing real-world programs. Execution of these programs is possible on existing hardware that supports the TSX instructions. In our setup, we used an Intel Xeon E5-2697 v4 "Broadwell-EP" with Turbo-Boost disabled. Further, hyper-threading was disabled to avoid transaction capacity problems due to shared L1 caches. A subset of the SPEC CPU 2006 benchmark suite [6] was used to compare the execution times of software with fault-tolerance disabled and enabled. Due to the restrictions of TSX on the length of transactions, and instructions that are not allowed in transactions, some benchmarks must have been elided from the evaluation. In this section, we break down the run time overhead and discuss its main sources. The effect of the possible hardware improvements proposed in Section 4 on the performance is shown afterwards.

5.1 Performance Overhead

On the Xeon E5 v4, the redundant execution on two cores takes more than twice the time for some benchmarks (see Fig. 5). To determine the sources of the overhead, individual parts of the instrumentation have been disabled. Results are shown in Fig. 4 and described in the following paragraphs.

Transactional Overhead One source of overhead is the transaction instructions, which wrap the dependable blocks in the trailing process. Since TSX does not guarantee a non-conflicting transaction to commit eventually, aborts due to other reasons may occur. These reasons are for example cache overflows, exceptions, and interrupts. Such aborts are transparent to the instrumented program, since the affected transaction is executed again. The chances to commit successfully are high for the second try of a transaction, depending on the abort reason. The resulting overhead of aborted and restarted transactions consists of the time needed to rollback the transaction plus the time needed to re-execute the code inside of this transaction. Cache overflows and exceptions may lead to never-committing transactions. In this case, the affected block is executed non-transactional by setting a flag for a conditional jump placed before the transaction start and end instructions. Within Fig. 4, the fraction of the performance overhead related to transactional memory instructions is shown in the black bars at the bottom ("Transaction"). Table 1 lists the executed transactions per benchmark, together with the average percentage of transaction aborts. Aborts due to cache overflow relative to the total number of transaction aborts and the number of non-transactionally executed blocks relative to the overall number of transactions are shown in the middle of Table 1.

Signature Generation Signatures are calculated at the end of every block, except for memory write instructions. For these, the signature is accumulated directly after the write instruction. The overhead of signature generation depends on the size of the blocks and on the number of blocks. Larger blocks decrease the overall overhead of signature generation, since less signature calculations are required. The percentage of overhead due to signature generation is shown in the bars labeled "Signature" in Fig. 4.

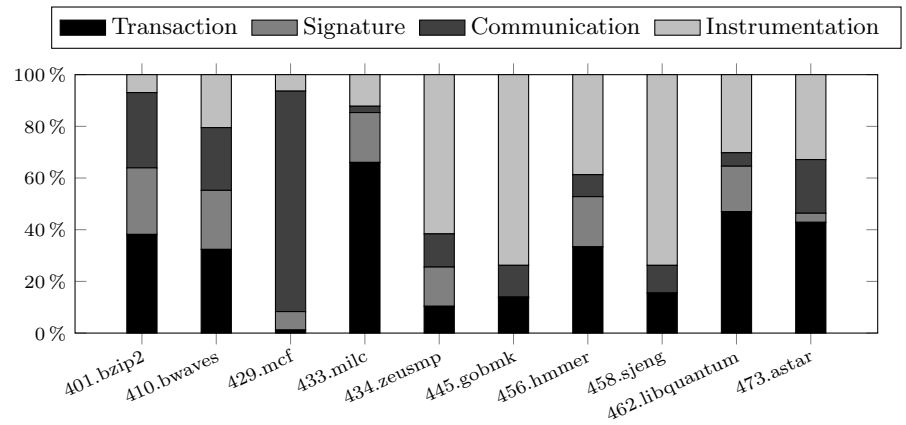

Fig. 4. Performance overhead arising from transaction instructions, signature generation and exchange, and binary instrumentation.

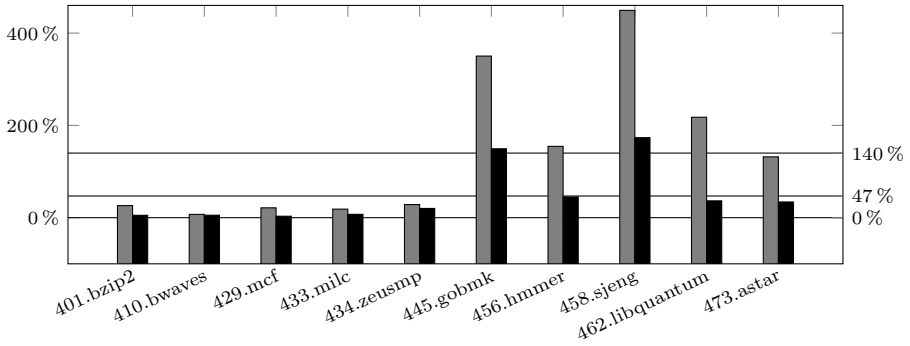

Fig. 5. Relative execution time overhead on an Intel Xeon E5 v4 (gray bars). Average performance overhead is 140 %. With emulation of hardware support for signature creation and exchange (black bars), average overhead is reduced to 47 %.

Core-to-Core Communication The performance of the communication between redundant processes is crucial, since signature reception is on the critical path of the trailing process. The FIFO queue decouples writing to and reading from the signature exchange buffer. Transmitting signatures from one core to another is still costly, since reading a cache line full of signatures always entails a cache miss. The execution time overhead related to signature exchange is shown in Fig. 4, labeled "Communication".

Instrumentation Overhead Additionally, we evaluated the performance overhead induced by the duplicated execution itself. In the instrumentation phase, signature generation, exchange, and comparison, as well as the transaction instructions were omitted. Only code that is required for duplicated process execution, consisting of process-identifying code and register-saving code, is inserted. Also, the redundant process is created. The gray bars on top in Fig. 4 ("Instrumentation") show the instrumentation overhead. On average, the instrumentation overhead is 36 %.

5.2 Impact of Proposed Hardware Enhancements

To estimate the maximum possible performance improvements with the previously proposed hardware enhancements, we modified the instrumentation to emulate the availability of signature generation and exchange in hardware. This emulation requires the modification of the signature comparison, since it is no longer possible to use real and correct values for the signatures. Signature comparison can be replaced in a way where the number and type of instructions can be the same, but the comparison results are always equal. By removing the `crc32` instructions in the instrumentation, we get an emulation of the hardware-integrated signature creation and a timing behavior close to what can be expected. To estimate the maximum performance improvement in case of a hardware-assisted core-to-core communication, we modified the instrumentation to assume zero cost for writing

Table 1. Performance of instrumented benchmarks. The number of transactionally executed blocks is listed with the relative amount of aborted transactions. Cache-related aborts are relative to aborted transactions and the number of non-transactionally executed blocks is relative to the total number of blocks. The execution time is relative to the execution time of the original benchmark. In the rightmost column, the relative execution time of instrumented benchmarks with emulation of hardware support is listed.

benchmark	# blocks	aborts	cache	no TX	rel. ex. time	HW opt.
401.bzip2	10.9 M	0.64 %	78.7 %	0.25 %	1.26	1.05
410.bwaves	3.6 M	3.52 %	22.6 %	0.33 %	1.07	1.05
429.mcf	2.6 M	8.91 %	95.6 %	4.30 %	1.21	1.03
433.milc	9.6 M	19.13 %	5.8 %	2.19 %	1.19	1.07
434.zeusmp	27.7 M	0.15 %	83.3 %	0.03 %	1.28	1.20
445.gobmk	653.6 M	0.03 %	59.5 %	0.01 %	4.50	2.49
456.hmmer	47.8 M	0.22 %	83.6 %	0.09 %	2.55	1.45
458.sjeng	265.6 M	0.06 %	10.1 %	0.00 %	5.49	2.73
462.libquantum	0.7 M	2.48 %	99.6 %	1.24 %	3.18	1.36
473.astar	127.3 M	2.27 %	99.0 %	1.03 %	2.32	1.34

and reading signatures. Calls to the appropriate functions are replaced with `nop`, and the signature comparison was disabled as described above. This is required to execute the program, otherwise the error detection would always roll back due to mismatching signatures. The emulation of hardware support for signature generation and exchange leads to significantly improved performance. The relative performance overhead of our approach decreases to 47 % on average and below 10 % for some benchmarks (see Fig. 5 and the rightmost column in Table 1).

6 Limitations and Future Work

The efficient implementation of transactional memory in Intel TSX allows fast checkpoint creation and rollback. Since it is a fixed hardware implementation, some limitations appear when leveraging TSX for fault-tolerant execution. As an optimistic and best-effort synchronization mechanism, the eventual successful execution of a transaction cannot be guaranteed. This can lead to regions in the application code containing unfriendly instructions, which cannot be executed in a transaction. Additionally, transactions may overflow easily if the L1 cache exceeds due to its limited associativity. The fixed hardware implementation of TSX causes some limitations, which must be handled by the instrumentation. Since TSX uses the L1 cache to hold back data which is not yet committed, a transaction must be aborted in case of a full cache or cache-set. This can happen frequently, since the L1 cache is eight-way set-associative.

A transaction may abort not only due to conflicting memory accesses, but also due to unfriendly instructions or other system events, like interrupts. Most unfriendly instructions modify processor flags or control registers, and thus rarely occur in user space code. In case of an abort due to exceptions or interrupts, the

transaction usually commits after a few tries. However, TSX does not guarantee that a transaction commits eventually, and some instructions always lead to a transaction abort. If our library detects repeated aborts of the same transaction, the affected block must be executed without a transaction. In fact, the execution of this specific block is then vulnerable. Error detection is still possible, but the non-transactional execution prevents a rollback. This reduces the fault-tolerant coverage of the application, but the instrumentation of the program can be optimized to lower the number of always aborting transactions, e. g. by splitting blocks to shorten transactions. Further, TSX transactions cannot be paused in a way that would allow to escape the transaction. Escape actions would allow to use transactions in both processes and to compare the signatures within the same transaction. In case of a rollback, the leading process would not be required to be killed, since it still would be possible to rollback the faulty block. As a workaround, the interleaved execution with signature exchange through a FIFO queue was implemented. However, this makes rollback complicated, since the leading process has to be killed and newly forked.

The presented approach is currently not capable of instrumenting multi-threaded programs, due to the difficulty of organizing the program execution of multiple redundant threads, synchronization in between, and assurance of correct execution after error recovery. This challenging topic will be part of our future work.

For a complete sphere of replication, calls to external functions and system calls require interception. Currently, external functions are executed without instrumentation in both processes to ensure the same functionality as the original, non-instrumented application. However, this prevents I/O on the same file handle. With an enhanced sphere of replication, redundant processes become transparent for their environment. External functions then are executed only once, with their result being provided to both redundant process. This is also required to support synchronization in redundant parallel applications.

7 Conclusion

In this paper, we proposed a software/hardware hybrid fault-tolerant execution on an Intel Xeon "Broadwell-EP" multi-core CPU. Transient errors can be detected by redundant execution and signature-based process comparison. By leveraging the checkpointing and rollback mechanisms of Intel's hardware transactional memory, an efficient method for error recovery and containment was presented.

Binary instrumentation enhances already compiled programs without requiring modifications of the source code. This allows a wide range of applications for fault-tolerant execution with full coverage of the program code. The resulting execution time overhead of redundant and fault-tolerant execution was shown to be 140 % on average on existing hardware. Signature generation and transactional execution are the main contributors to the increased execution time. However, hardware enhancements and the integration of advanced techniques for signature creation and exchange reduce the average run time overhead to 47 %.

References

1. Bernick, D., Bruckert, B., Vigna, P.D., Garcia, D., Jardine, R., Klecka, J., Smullen, J.: NonStop® Advanced Architecture. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN). pp. 12–21 (2005)
2. Fetzer, C., Felber, P.: Transactional Memory for Dependable Embedded Systems. In: Proceedings of the International Conference on Dependable Systems and Networks Workshops (DSN-W). pp. 223–227 (2011)
3. Haas, F., Weis, S., Metzlaff, S., Ungerer, T.: Exploiting Intel TSX for Fault-Tolerant Execution in Safety-Critical Systems. In: Proceedings of the International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT). pp. 197–202 (2014)
4. Haas, F., Weis, S., Ungerer, T., Pokam, G., Wu, Y.: POSTER: Fault-tolerant Execution on COTS Multi-core Processors with Hardware Transactional Memory Support. In: Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT). pp. 421–422 (9 2016)
5. 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)
6. Henning, J.L.: SPEC CPU2006 Benchmark Descriptions. ACM SIGARCH Computer Architecture News 34(4), 1–17 (2006)
7. Herlihy, M., Moss, J.E.B.: Transactional Memory: Architectural Support for Lock-Free Data Structures. In: Proceedings of the International Symposium on Computer Architecture (ISCA). pp. 289–300 (1993)
8. Kuvaiskii, D., Faqeh, R., Bhatotia, P., Felber, P., Fetzer, C.: HAFT: Hardware-assisted Fault Tolerance. In: Proceedings of the European Conference on Computer Systems (EuroSys). pp. 25:1–25:17 (2016), http://doi.acm.org/10.1145/2901318.2901339
9. LaFrieda, C., Ipek, E., Martinez, J.F., Manohar, R.: Utilizing Dynamically Coupled Cores to Form a Resilient Chip Multiprocessor. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN). pp. 317–326 (2007)
10. Laurenzano, M.A., Tikir, M.M., Carrington, L., Snavely, A.: PEBIL: Efficient Static Binary Instrumentation for Linux. In: Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 175–183 (2010)
11. Mukherjee, S.: Architecture Design for Soft Errors. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2008)
12. Reinhardt, S.K., Mukherjee, S.S.: Transient Fault Detection via Simultaneous Multithreading. In: Proceedings of the International Symposium on Computer Architecture (ISCA). pp. 25–36 (2000)
13. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I.: SWIFT: Software Implemented Fault Tolerance. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO). pp. 243–254 (2005)
14. Shye, A., Blomstedt, J., Moseley, T., Reddi, V.J., Connors, D.A.: PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures. IEEE Transactions on Dependable and Secure Computing (TDSC) 6(2), 135–148 (2009)
15. Yalcin, G., Unsal, O.S., Cristal, A.: Fault Tolerance for Multi-Threaded Applications by Leveraging Hardware Transactional Memory. In: Proceedings of the International Conference on Computing Frontiers (CF). pp. 4:1–4:9 (2013)