Exploiting Intel TSX for Fault-Tolerant Execution in Safety-Critical Systems

Florian Haas, Sebastian Weis, Stefan Metzlaff, and Theo Ungerer
Department of Computer Science
University of Augsburg, Germany
{haas,weis,ungerer}@informatik.uni-augsburg.de

Abstract—Safety-critical systems demand increasing computational power, which requests high-performance embedded systems. While commercial-off-the-shelf (COTS) processors offer high computational performance for a low price, they do not provide hardware support for fault-tolerant execution. However, pure software-based fault-tolerance methods entail high design complexity and runtime overhead.

In this paper, we present an efficient software/hardware-based redundant execution scheme for a COTS x86 processor, which exploits the Transactional Synchronization Extensions (TSX) introduced with the Intel Haswell microarchitecture. Our approach extends a static binary instrumentation tool to insert fault-tolerant transactions and fault-detection instructions at function granularity. TSX hardware support is used for error containment and recovery. The average runtime overhead for selected SPEC2006 benchmarks was only 49% compared to a non-fault-tolerant execution.

I. INTRODUCTION

Beside the increasing demand for computational power in safety-critical systems, assurance of correct execution is still necessary to meet safety requirements (see ISO 26262). While high-performance multi-core systems provide enough performance, they usually cannot guarantee a correct and fault-free execution. In current high performance processors data is usually protected on different levels (ECC, redundant hard disks, etc.), but transient errors can still occur within the processor’s pipelines. To cope with transient errors, redundant execution schemes like lock-stepping or software-based approaches have to be used. However, lock-stepping requires doubling all hardware resources and keeping them in synchronization. In up-to-date processors with varying clock rates, multi-threading, and superscalar out-of-order execution, this is not achievable. In contrary, pure software approaches lead to a vast overhead with a big impact on the performance.

We propose a fault-tolerance enabling library (libfttm), which takes advantage of Hardware Transactional Memory (HTM) in combination with a software-based redundant execution scheme. Our approach is built on top of Restricted Transactional Memory (RTM), which is part of Intel’s Transactional Synchronization Extensions (TSX), implemented in the current Haswell microarchitecture [1]. This allows our approach to be executed on widely available COTS systems with only little overhead.

Our approach targets a Linux environment and uses the address space isolation of its processes’ virtual memory and the rollback-capability of TSX for loosely-coupled lock-stepping. The transactional execution implicitly creates checkpoints, which can be used for rollback in case an error is detected. As a consequence, our approach allows fail-operational execution. The ability of explicit rollbacks and the transactions’ atomicity allow fault-containment. Compared to software-based approaches like Software Transactional Memory (STM), restoring the state before the transaction is significantly faster with hardware support, since logging of memory operations and rollback in case of transaction aborts can be done more efficiently in hardware.

The rest of the paper is organized as follows. Section II provides a short overview of related work. The general approach, the challenges resulting from given hardware restrictions, and the implementation of fault-tolerance with TSX are described in Section III. Section IV presents the evaluation and Section IV discusses the limitations and future work. The paper is concluded in Section VI.

II. RELATED WORK

Hardware-supported loosely coupled lockstep architectures were presented for SMT processors [2, 3] and multi-core processors [4, 5]. However, they all require complex changes to the pipeline and the memory architecture, which make their use impossible in current high-performance COTS processors.

On the other side, software-based fault-tolerant mechanisms provide flexibility and do not require special hardware support. SWIFT [6], a compiler-based approach, doubles the instructions and compares their results within a single thread, and additionally offers control flow checking. PLR [7] splits a single-threaded program into two processes to detect errors. Result comparison of the redundant processes is implemented on system call granularity. By contrast to our approach, PLR cannot recover from errors without additional support for process checkpointing. Wang et al. [8] use redundant threads in their compiler-based fault tolerance approach. However, these software-only approaches request complex software-based fault-containment, which makes rollbacks to a previous error-free state costly, or requires high frequent result comparison.

The application of hardware transactional memory in dependable systems based on high-performance COTS processors is suggested in [9]. They encourage the use of the implicit rollback mechanisms for easy checkpointing and
recovery. A related approach for fault-tolerant execution based on hardware transactional memory is FaulTM [10]. It uses a custom HTM implementation, which brings the benefit of extendability for a better error-detection support. A duplicated process is executed in concurrent transactions, and before committing these transactions, their write sets are compared. A mismatch signifies a detected error and thus leads to a rollback of both transactions.

III. APPROACH

RTM allows to define an atomic region of program code by applying the instructions `xbegin` and `xend`. All write operations within this transaction become only visible in the global memory, when the transaction has successfully committed. In case of a conflict, for example due to mutual access to shared data, the transaction is aborted and a rollback to the state before is performed. This effect of implicit checkpointing in hardware is used by our approach for recovery when a fault was detected.

The general idea behind our approach is as follows: the fault-tolerant execution model is based on the parallel execution of two instances of the same program. The program code is divided up into transactions. All memory and register write operations are calculated into a signature hash. Both hash values are compared after the transaction completes. In case they match, the write log is applied and written to memory. Afterwards, the next pair of transactions is executed. In the case of a detected mismatch of the hash values, an operand or address of any memory or register operation must be erroneous and thus leads to the differing signature. A rollback must be done to discard all write operations and restore the previous process context. The execution restarts in a correct state at the beginning of the transaction.

A. Challenges

However, this approach cannot be realized with the existing TSX implementation due to various restrictions. Before we present the actual approach, we will outline the main challenges when using RTM for fault-tolerant execution:

1) **TSX cannot be enhanced**: As TSX does not offer software handlers in addition to the abort handler, no enhancements can be made to the conflict detection. As a consequence, it is not possible to detect errors as conflicts due to differing data in both transactions. The error detection has to be done by checking signatures of the written data at the end of the transaction.

2) **Signatures cannot be compared within transactions**: The signatures created for the comparison of the memory and register operations have to be exchanged between both transactions. An exchange of data between transactions is not possible because of conflicting write and read operations. Writing to a memory address, which has been read by another transaction, leads to an immediate abort. As a consequence, the transactions have to be executed consecutively. Parallelism can be kept through overlapping execution of the successor’s transaction and the next transaction of the predecessor process. Because the predecessor transaction always has to commit and thus is irrecoverable, the transactional execution is only required for the successor process. For restoring the predecessor process in case of a detected error, its whole process is killed before restarting the transaction in the successor process. Then a new process is spawned for the predecessor. The program is then again in a correct state, since the transaction is restarted and the possibly erroneous predecessor process is killed.

3) **TSX is best-effort**: TSX is implemented for best-effort. As a consequence, no guarantee can be given for any transaction to finally commit successfully. Transactions can abort at any time, also if no conflicts have occurred. This can be for example due to instructions which modify the processor status, and also timer interrupts lead to transaction aborts. As TSX is implemented within the L1 cache, a transaction cannot exceed the cache’s capacity. If hyper-threading is enabled, two threads on the same CPU core share the L1 cache, which results in a bisection of the available L1 cache. The possibility of never-committing transactions and mutual detention due to conflicts impede deterministic behavior of TSX. However, in our approach no parallel transactions occur, since only the successor process executes transactions. As a consequence, aborts can only occur because of forbidden instructions, interrupts or cache overflows. If a transaction never commits, this specific part of the program has to be executed without rollback-capability to ensure progress of the program. In the general case, the programmer should consider such situations to allow a complete coverage of the program.

B. Principle

To get along with the challenges mentioned above, our basic approach is enhanced. To achieve parallelism for the required consecutive execution of predecessor and successor transaction, the successor transaction and the predecessor of the subsequent transaction block are executed interleaved. In the following, the predecessor transaction is named leader and the successor transaction trailer.

Fig. 1: Detecting Faults and Recovering with TSX

Because the predecessor transaction always has to commit and thus is irrecoverable, the transactional execution is only required for the successor process. For restoring the predecessor process in case of a detected error, its whole process is killed before restarting the transaction in the successor process. Then a new process is spawned for the predecessor. The program is then again in a correct state, since the transaction is restarted and the possibly erroneous predecessor process is killed.
1) Fault Detection: Fault detection requires knowledge about the differences between the write sets of the leader and the trailer. To avoid the comparison of the overall write set, a signature is calculated with a hash function, integrating the address and data of all memory accesses. This signature is written to a shared memory region after the leader transaction committed (see Fig. 1a). The signature is calculated the same way in the trailer. The signatures are compared at the end in the trailer’s transaction. If they match, the transaction commits.

2) Fault Recovery: If an error is detected due to different signatures, the trailer transaction is aborted. This restores the state before the transaction started, which is supposed to be a correct state for the trailer process (see Fig. 1b). However, the error can happen in the trailer or in the leader process, but the leader process cannot roll back. Fault containment must be assured for both processes.

3) Fault Containment: Detecting errors and recovering are not sufficient for fault-tolerant systems. Errors also have to be contained and must not be able to spread. In our approach, errors are contained within the trailer transaction. In case of a detected error, the transaction is restarted. This discards the faulty state and the state at the beginning of the transaction is assumed to be correct, otherwise the previous transaction should have failed. Special consideration is required for the leader. As mentioned above, the signature can only be exchanged after the transaction has finished, which implies that this transaction cannot rollback. To contain errors in the leader, its whole process is killed, discarding its possibly faulty state. The abort handler of the aborted trailing transaction spawns a new process by forking. The execution continues at the beginning of the faulty transaction, where both the trailer and leader are again in a correct and error-free state.

C. Implementation

The Fault-Tolerant Transactional Memory library (libfttm) is a C library which contains the code for process splitting, signature exchange, synchronization, fault detection, and fault recovery. The process from given C or C++ source to a fault-tolerant binary executable is depicted in Fig. 2. The dependable parts of the program are wrapped, a prefix function replicates the process and a postfix function joins both processes at the end. After compilation and linking, the binary code is instrumented with transactions and the fault-detection code. The statistics gathered by our library can be used for optimization and adaption of the instrumentation. The performance counters can be analyzed on the Haswell CPU by perf, which helps to identify bottlenecks.

1) Replication: The replication is achieved by forking the process (see Fig. 3a). A new POSIX process is spawned, which executes the same payload code as the original process. This new process is almost identical to the original process because of the same virtual address space. Linux’ method of copy-on-write uses the existing memory pages and copies only modified parts of the newly created process. This allows a fast creation of the child process and thus results in acceptable overhead. The separation into two processes is helpful for fault containment, because the processes cannot modify each other’s data due to memory protection. The same virtual addresses allow the easy comparison of the data and addresses of both processes. One or more entry points where the process is forked have to be specified. A simple wrapper around a function call provides this functionality. In the simplest case, the main function itself is wrapped, which can be done by appropriate linker flags.

2) Instrumentation: The instrumentation of the program’s binary file is accomplished with PEBIL [11], a static instrumentation tool. Both leader and trailer processes are split into parts (see Fig. 3b), of which the trailer parts are executed transactionally. Our extension to PEBIL adds the required instructions for starting and ending transactions, the abort handler, and signature calculation. The comparison of the leader’s and trailer’s signatures is also added here. Splitting is based on function boundaries. Generally, one function is executed in one transaction. If another function is called within an instrumented function, the transaction is ended before the function call and a new transaction is started afterwards. The user provides a list of functions which are to be instrumented.

The signature calculation accumulates all memory addresses and the data itself in a single hash value. For storing this 128 bit hash value within a transaction, register XMM14 is used. The Haswell microarchitecture offers dedicated AES instructions, in particular aesenc, which adds a value to the AES encryption hash. This instrumentation is shown in Fig. 4. Every memory operation is instrumented and both the data value and the address are added to the hash. The XMM registers used in the instrumentation are locked by compiler flags to prevent their usage for other purposes.

(a) Process replication on source code.
(b) Instrumentation of the binary with transactions.

Fig. 2: Replication and instrumentation process on source code and program binary.

Fig. 3: Steps of Instrumentation
3) **Signature Exchange:** To exchange the signature between the leader and the trailer for comparison, communication between them is required. The fastest method for sending the signature to the trailer is via shared memory. For this, a lock-free FIFO queue is implemented, which solely requires read/write memory fences before accessing the queue. This solution happens completely in userspace, since no locks are required. The precedence order is implicitly obtained by the FIFO queue. The FIFO queue weakens the effects of varying transaction lengths, since the leader can advance until the queue is full (see Fig. 5).

4) **Abort handler:** Transactions may abort due to different reasons. In the expected case, an error was detected and the transaction is aborted explicitly with \texttt{xabort}. Aborts can also occur because of other influences. As mentioned before, TSX is prone to interrupts and exceeding cache limits. So at first, the type of the abort is distinguished and in case of non-explicit aborts the transaction is simply restarted. Since TSX does not guarantee transactions to eventually commit, a limit for maximum subsequent aborts is defined. After reaching this limit, the code of the aborting transaction is executed non-transactional. Losing the rollback-possibility on a small piece of code allows the program to proceed. The location of such parts of the program can be printed out by the library, so the programmer is able to adjust its code.

If the transaction aborts due to an explicit abort, a signature mismatch has happened. In this case, not only the trailing transaction has to be restarted, but also the leading process must be killed. This is required to obtain fault containment, and because the error may already have affected the memory of the leader, it must be killed and replaced by a new fork of the trailing process. Finally, the program execution continues at the beginning of the failed transaction in a guaranteed fault-free state.

5) **Signal handler:** Signals sent to the program have to be handled, especially segmentation faults, illegal instructions, and floating-point exceptions. The behavior of the program is unexpected at first glance, when receiving those signals within a transaction. For example, an erroneous pointer due to a bit flip leads to a segmentation fault, since the address is not within the program’s memory. The CPU misses the address in the TLB and raises a page fault exception to be handled by the operating system. However, the exception forces the transaction to abort immediately, entailing a rollback, and thus the exception never gets visible. When retrying, the operating system had no chance to map the page and the exception is raised again. In the trailing process, this behavior is beneficial, since in the next try of the transaction, the pointer is possibly correct and the program can proceed. In the leader, however, the transaction does not abort and a segmentation fault is signaled by the operating system. Since the termination of the program has to be avoided, the same rollback mechanism as in case of a detected error must be applied. Therefore, a signal handler catches all signals and induces the rollback.

**IV. Evaluation**

To show the applicability of our approach, we measured the performance overhead in the error-free case. For testing the fault-detection and recovery capability, errors were injected and the instructions required for recovery were measured.

**A. Performance Overhead**

The performance overhead was measured on a desktop computer with a current Intel i7-4770 CPU. The general overhead of transactions on the Haswell CPU was evaluated in [12]. Begin and commit of a transaction comes with relatively low cost, the total overhead is 43 cycles. The wrapper code of our library around the instrumented parts of the program increases the amount of executed instructions by around 20 million. This includes the fork system call, preparations in both processes and additional instructions for statistics. Every instrumented transaction comes with an overhead of 261 instructions, which contain the code for transaction preparation and abort handling, plus 105 instructions for signature exchange via the lock-free FIFO. Each single instrumented memory operation adds 4 extra instructions for signature calculation.

The fault-tolerance run-time library was evaluated by a subset of the SPEC2006 benchmarks [13]. The number of benchmarks that can be used for our approach is limited by different factors: the benchmarks have to be written in C or C++ and no indeterminism is allowed, otherwise the execution in the leader and trailer processes would be different. Also, the signature calculation restricts the usage of the floating point registers, thus we evaluated only the integer benchmarks. The main computational part of the benchmarks were wrapped and instrumented. Only a small amount of functions which cannot be executed in a transaction were excluded from instrumentation. The execution time of the original benchmark without

---

Fig. 4: Instrumentation of a single instruction with hash instructions for signature generation.

Fig. 5: Signature exchange through a lockfree FIFO queue decreases the influence of varying transaction lengths.
TABLE I: SPEC2006 C Integer Benchmarks Results

<table>
<thead>
<tr>
<th>benchmark</th>
<th># TX</th>
<th># in./TX</th>
<th>ex. instr.</th>
<th>ex. time</th>
<th># aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>401.bzip2</td>
<td>9.1</td>
<td>9.2</td>
<td>252 %</td>
<td>147 %</td>
<td>4,329</td>
</tr>
<tr>
<td>429.mcf</td>
<td>11.7</td>
<td>10.8</td>
<td>302 %</td>
<td>155 %</td>
<td>13,304</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>8.3</td>
<td>6.2</td>
<td>399 %</td>
<td>197 %</td>
<td>881</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>15.9</td>
<td>10.7</td>
<td>257 %</td>
<td>157 %</td>
<td>682</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>17.1</td>
<td>16.9</td>
<td>268 %</td>
<td>152 %</td>
<td>759</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>9.3</td>
<td>11.4</td>
<td>214 %</td>
<td>128 %</td>
<td>7,018</td>
</tr>
<tr>
<td>473.astar</td>
<td>5.7</td>
<td>10.7</td>
<td>223 %</td>
<td>112 %</td>
<td>1,166</td>
</tr>
</tbody>
</table>

Fig. 6 shows the overhead in percentage of the original execution time. The number of executed transactions (# TX) and the resulting overhead is depicted in Table I. Additionally, the average number of instructions per transaction (# in./TX), and the amount of aborted transactions is shown. Column ex. instr. shows the amount of additionally executed instructions, relative to the original program. The execution time overhead relative to the original program is shown in column ex. time. It can be seen that the execution of the duplicated program leads to more than twice of the executed instructions (214 % to 399 %), but the total execution time is less than twice (112 % to 197 %). The exchange of signatures between the processes requires fences for a consistent state of the FIFO queue and thus induces overhead. However, signature creation itself is less costly, since the additional aesenc instruction and register operations can be executed in parallel to the original memory instructions, due to superscalar out-of-order execution.

If a transaction aborts subsequently, the program advance is guaranteed by omitting the transactional instructions for this specific part. Aborts are mostly due to cache capacity overflows, missing memory pages, or processor internal events. The evaluation also showed that the overhead does not only depend on the number of transactions, but also on their length. 445.gobmk had an higher overhead than 462.libquantum, although both benchmarks executed a similar amount of transactions. However, the average size of a transaction was smaller for 445.gobmk, leading to more overhead relative to the execution time. The reasons for small transaction sizes are either short functions or many function calls with only a small amount of instructions in between, because functions boundaries determine the transaction sizes. Transaction aborts also increase the overhead, but they happen rarely and thus don’t have high impact on the execution time.

B. Fault Injection and Recovery Overhead

For fault-injection, we used the MARSS x86 simulator [14]. An implementation of HTM with compatible TSX instructions as on the Haswell CPU was newly available. Although the implementation is different, with a dedicated memory for the transactional memory write set, it does not influence the evaluation. In MARSS, error injection was implemented in the commit stage, where single bits are flipped in random reorder buffer entries before instruction commit. As we were only interested in faults which manifest and actually lead to errors in addresses or data, only reorder buffer entries, which are currently in use, were injected with errors. This leads to erroneous data in the registers, which are either treated as data or address. As a consequence, the signatures calculated in both processes do no longer match. A fixed error rate is specified and in the next possible cycle a single bit in one reorder buffer entry is flipped. Only programs instrumented with our approach are fault-tolerant, all other programs and especially the kernel are not fault-tolerant. As this was not subject to our research, we excluded all non-fault-tolerant programs from error injection. Also the code of the libfttm is prone to errors, so faults are only injected if the thread is currently within a transaction. Erroneous addresses may lead to segmentation faults which lead to program termination or kernel panics outside the fault-tolerant code and thus prohibit an evaluation.

Our approach was able to detect and recover from all injected errors which manifest either in addresses or data within the instrumented code. Interesting in the case of a detected error is the fault-recovery time, e.g. the time required for returning to a previous error-free state (mean time to recover, MTTR). In case an error is detected, not only the transaction in the trailer process is restarted, but also the whole leader process has to be killed and restarted by forking. This entails a lot of executed instructions, as system calls and kernel code is involved. The fault recovery is the only case in which a huge overhead is involved, while the normal operations entail only a slight overhead. However, the mean time between failure (MTBF) is usually relatively high, compared to the time required for recovery, which is in the order of milliseconds. In total, around 14.4 million instructions were executed on every rollback. This does not contain the instructions executed on the retry of the transaction. Also, the rollback itself does not require additional instructions, since this happens completely in hardware. Most instructions in the rollback case are mainly due to the killing of the leading process and the fork.
V. Limitations & Future Work

Although Intel’s TSX offers promising opportunities to exploit for fault-tolerant execution, its current implementation has some limiting factors for a high-performance execution with high fault coverage. In the TSX’ nature of providing a best-effort synchronization mechanism, the successful execution of transactions is not guaranteed. As a consequence, programs may exhibit regions of code which are not possible to be executed within a transaction. Additionally, the implementation within the L1 cache requires the transaction’s write-set to fit in the eight-way associative cache.

Input/Output and system calls are currently executed individually in both processes. PLR [7] for example offers input replication and system call emulation in its sphere of replication. In future versions of our approach, input data has to be replicated, and output data compared before actually writing it. This would allow the access to a shared file handle for reading or writing. System calls should be executed only once after comparing their input parameters. The diverging parallel execution of both processes requires synchronization before such operations, which could influence the overhead negatively.

The next step, especially with the four-core SMT Core i7 in mind, is to support parallel applications. This requires special implementation of locks and synchronization mechanisms. Further, we expect to achieve better instrumentation results by developing compiler support to replace the static instrumentation of the binary. The additionally available information in the compiler could allow for longer transactions and thus less overhead.

VI. Conclusion

This paper has shown the applicability of TSX for fault-tolerant execution on the Intel Core i7. Our software-based approach duplicates the process of a single-threaded program and the binary instrumentation encapsulates the code in transactions. Fault detection is based on the comparison of signatures, which are calculated in both processes. To achieve fault containment, the fault-tolerant transaction of the trailer thread is rolled back and the leader process is replaced with a new duplicate. The performance overhead of 49% in average is shown to be acceptable, since both processes can be executed in parallel. The promising results motivate to extend our approach towards support of multi-threaded applications.

ACKNOWLEDGMENT

This work is partly funded by an Intel Research Grant on high-performance computing in safety critical systems.

REFERENCES


