Transactional IPC in Fiasco.OC
Can we get the multicore case verified for free?

Till Smejkal, Adam Lackorzynski, Benjamin Engel and Marcus Völp

Operating Systems Group
Technische Universität Dresden, Germany
<name>.<surname>@tu-dresden.de

***Abstract*—Already announced in 2007 for Sun's Rock processor but later canceled, hardware transactional memory (HTM) finally found its way into general-purpose desktop and server systems and is soon to be expected for embedded and real-time systems. However, although current hardware implementations have their pitfalls, hindering an immediate adoption of HTM as a synchronization primitive for real-time operating-systems, we illustrate on the example of a transactional implementation of the L4/Fiasco.OC inter-process communication (IPC) how extended versions of HTM may revolutionize kernel design and, in particular, how they may reduce the verification costs of a multi-core kernel to little more than verifying a selectively preemptible uni-processor kernel. Removing L4/Fiasco.OC's half thousand lines-of-code cross-processor IPC path and making the local path transactional, we benefit from a principal performance boost for sending cross-core messages. However for the average case, we experience a 30 % overhead for local calls.**

I. Introduction

Cyber-physical systems such as autonomous cars, medical robots, and airplanes increasingly apply multi-core hardware and multi-core real-time operating systems (RTOS) to meet the performance demand of their applications. At the same time, these systems often operate with or in close proximity of humans, thus safety is a must and formal verification is most rigorous in assuring that a system is to be trusted. However, although fully verified single processor systems are at the verge (first microkernels have already been verified [1], [2]), multiprocessor verification remains a milestone to be taken.

Verification of uniprocessor kernels typically proceeds by splitting the high level verification goal into smaller properties and invariants, which are then shown to hold for arbitrary sequences of non-preemptively executing pieces of kernel code. One, if not the challenge when comparing the verification of multiprocessor kernels with uniprocessor kernels is that non-preemptive execution no longer conveys atomicity at the granularity of non-preemptive execution, but at the granularity of individual processor instructions. Instead of having to consider arbitrary interleavings of large code pieces, one must therefore establish the desired results for all possible interleavings of machine instructions, which easily pushes verification complexity beyond manageable bounds. Of course, there are several tools to assist in this tasks, for example, concurrent separation logic [3] and the multitude of approaches that followed Owicki and Gries [4], [5] seminal work on assume-guarantee reasoning. However, despite these tools, one must still specify and verify the behavior of the kernel at a fine granular and machine-dependent level.

In this paper, we argue why we believe transactions can re-establish some of the simplicity one finds when verifying uniprocessor kernels by reintroducing atomicity at a coarse granularity and, most importantly, in a machine-independent way. Our goal is not to translate uniprocessor results to the multiprocessor case, which if possible at all requires careful argumentation. Instead, we propose to re-implement the kernel as sequences of large transactions to regain the atomicity of non-preemptive execution. We evaluate on the example of L4/Fiasco.OC's IPC path using the hardware transactional memory implementation found in Intel's Haswell processors to which degree this is possible and at which costs.

We present our transactional IPC path in Section III, compare its performance against mainline Fiasco in Section IV and illustrate in a semi-formal way in Section V how lifting atomicity from individual instructions to coarse grain transactions simplifies the multicore verification task to little more than what is required when verifying uniprocessor kernels.

II. Hardware Transactional Memory

In 1993, Herlihy and Moss [6] proposed transactional memory (TM) as a mechanism to assist developers in protecting shared data structure accesses in parallel systems. Unlike lock-protected data structures, which to scale require cumbersome to design and error prone fine-grain locking schemes, transactional memory performs modifications of data structures optimistically but is prepared to discard these modifications in case of conflicts. Especially for low-contended locks, TM avoids the locking overhead at the expense of guaranteed progress in situations where transactions abort.

To implement transactional operations in hardware, i.e., to ensure *atomicity* of updates in case the transaction completes and *isolation* in the sense that modifications remain invisible until the transaction commits, Herlihy and Moss proposed to exploit processor local caches as interim storage and cache coherence protocols (such as MESI) for conflict detection. External writes abort transactions if they are to any data loaded into the cache or accessed while executing transactionally; external reads abort a transaction if they are to cachelines that are cached in exclusive modified state (M) as a result of transactional writes. Further aborts may happen if transactional data exceeds the capacity of the cache or for other reasons that are specific to the concrete implementation of hardware transactional memory (HTM).

Many software implementations of transactional memory have been proposed over the years (see e.g. [7], [8]), including hybrid hardware-software solutions [9]. However, their applicability is limited due to significant overheads as identified by Cascaval et al. [10]. The first full fledged hardware implementation as described by Herlihy and Moss has been announced in 2011 for IBM's BlueGene-Q servers [11], followed by Intel's Transactional Synchronization Extension (TSX) [12] for standard PC hardware in 2012.

TSX offers two distinct features: *Hardware Lock Elision* and *Restricted Transactional Memory*. Hardware lock elision [13] automatically replaces locks with transactions by replacing the acquisition of the lock with a transaction begin and the release with an attempt to commit the transactional state collected while executing the critical section. In contrast, restricted transactional memory (RTM) exposes the complete transaction interface to the programmer allowing her to start, commit and abort transactions through special processor instructions. The limitations of RTM are conflict detection only at the granularity of cachelines but no finer, the bounded amount of memory that can be accessed from within a transaction, and, as far as Intel's implementation is concerned, the lack of any progress guarantee with regard to which transaction will abort. In particular, RTM aborts transactions on interrupts, system calls and in many other situations, including the execution of some privileged instructions.

Our main focus in this paper is on safety, security and correctness but not on lifeness and guaranteed completion. Nevertheless, we will argue why transactions should be considered as a mechanism to simplify the kernel and why real-time systems require future implementations of hardware transactional memory to convey progress guarantees similar to those provided by IBM in BlueGene-Q.

III. Transactional Inter-Process Communication

With *TxLinux* Ramadan et al. [14] have already shown the value of hardware transactional memory for synchronizing access to kernel data structures. However, to leverage the full potential of HTM for both simplifying in-kernel locking and verifying multi- and manycore kernels, all system calls must execute transactionally, at least to the best degree possible.

To demonstrate the feasibility (and drawbacks) of almost fully transactional system calls, we use as an example an implementation of L4/Fiasco.OC's IPC path with Intel's RTM.

A. The L4/Fiasco.OC Microkernel

L4/Fiasco.OC is a 3rd-generation capability-based microkernel designed for use in both security and real-time critical scenarios. Following Liedtke's design principle [15], the L4 family microkernel provides only those functionality in the kernel, which cannot sensibly be implemented at application level. This is the functionality required to isolate user-level subsystems (capabilities and address spaces) and inter-process communication (IPC), which provides a safe and secure means for communicating between these subsystems.

IPC messages in L4 may contain both data and capabilities, which are required to invoke kernel-implemented objects (such as IPC gates to send messages to other threads). IPC is synchronous, that is the sender blocks until the receiver is ready to receive, which removes buffer allocation from the IPC path and allows the threads' user-level control block to be used as message buffer. Through IPC operations, threads may *send* or *receive* messages or they may *call* other threads, which is an atomic send and receive operation in the sense that when the callee receives the message, the caller is already ready to receive from this thread. IPC is transparent, that is IPC uniformly works in the same way irrespective of the core on which the receiver is executing. It may be on the same core, in which case we say IPC is local or on a different processor core than the sender, in which case IPC is cross processor.

Mainline L4/Fiasco.OC [16] comes with two tightly integrated IPC paths: a fast path for core-local communication and a cross-processor IPC path, designed to preserve the performance of local IPC as much as possible. In this paper, we explore how IPC and especially cross-processor IPC can be implemented with HTM mechanisms. Besides simplifying the cross-processor IPC path (when compared to existing non-TM approaches), we show that, with a few exceptions, transactions span the same parts of the code that executes non-preemptively in the core-local case. For these exceptions, we explain why they have to execute non-transactional and sketch how one can further reduce the amount of non-transactional code.

B. IPC with RTM

Ideally, from the viewpoint of verifying the kernel and to minimize transaction overhead, the entire IPC operation should be a single transaction. However, there are two general obstacles, which prevent us from turning IPC and, more generally, system calls into a single transaction each: (i) privileged instructions and device accesses abort transactions unconditionally; and (ii) transactional state may become too large to fit the L1 cache, which also leads to aborts. In the L4/Fiasco.OC IPC path, the transaction-aborting operations are the programming of timeouts, which involves setting the hardware-timer to the earliest pending timeout, and the reloading of the page-table base register, when IPC switches to a thread in another address space. In addition, on architectures such as ARM, the transfer of memory capabilities causes aborts when TLB entries have to be flushed as a result of upgrading page-table entries.

Fig. 1 shows a schematic of the L4/Fiasco.OC IPC path and the steps involved when the left-hand thread calls either one of the two right-hand side threads (in the same or in a different address space). Immediately after entering the kernel (e.g., with `sysenter` on x86-systems), execution may proceed transactionally (with `xbegin`) after setting the address of the abort handler (black dot #1 in Fig. 1). IPC proceeds by checking whether the receiver is waiting for the sender (i.e., it has already executed a receive operation) or whether the receiver is still involved in other operations (e.g., it may be running). In the first case, the sender and receiver rendezvous and the kernel starts the message transfer. After the transfer completes, which in case of a transfer of memory capabilities may require additional preemption points and hence transactions, the caller prepares its receive phase to ensure that it is ready to receive when the receiver replies. In case of capability transfers, the TLB shootdown can either be deferred to after the IPC operation or handled immediately after the capability transferring transaction commits. Switching to the receiver involves storing and reloading the register state and stack pointers of the IPC partners. These operations can be

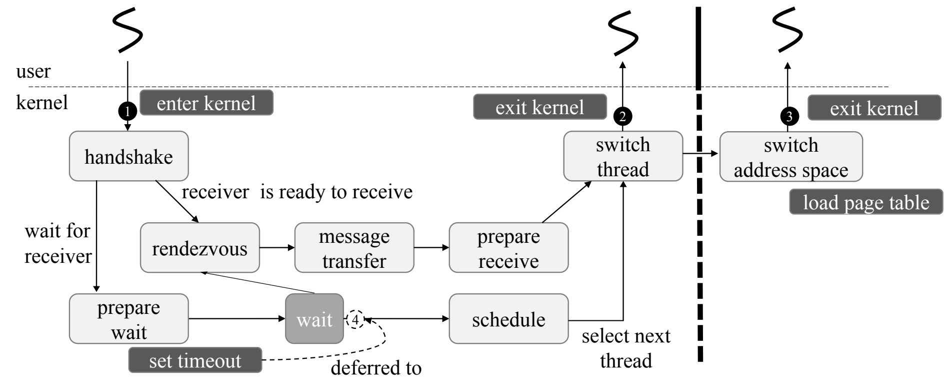

Fig. 1. Schematics of the L4/Fiasco.OC IPC path for an IPC call operation to a peer thread within the same (middle thread denoted by the wiggling line) or another (right thread) address space, that is intra vs. inter address space communication. Black (and dashed white) dots mark the begin and end of transactions. The path either directly proceeds to the receiver or stops at the preemption point *wait*. As part of waiting, the scheduler is invoked to select the next ready-to-run thread to which it then switches. Dark gray operations are privileged operations, which cannot be executed transactionally. They are deferred until after the end of the transaction.

executed transactionally without risk of abort. Therefore, when sending to a thread in the same address space, only one transaction is required, unless capability transfers needs to be preemptible. The transaction starts at the black dot #1 and commits immediately before returning to user-level (e.g., with `sysexit`) at #2.

If the receiver resides in a different address space, the page-table base register must be reloaded, which causes an unconditional abort when executed transactionally. Therefore, we defer the actual address space switch to the point in time after the transaction commits at #3 and execute it immediately before returning to the user. The instructions that remain non-transactionally are the check whether an address space switch is pending and the `mov %1, cr3` instruction, which performs the switch of the page table and hence of the address space. For a verification, these arbitrary interleavings of these two instructions of the transactions and other non-transactional code must be considered. However, because only few operations must be deferred. We expect these interleaving to remain within manageable complexity.

So far we have only considered the case where the receiver is ready to receive from the sender and not involved in some other operation. If this is not the case, the sender blocks waiting for the receiver to execute the receive and message transfer[1].

L4/Fiasco.OC limits the time senders have to wait for receivers to participate in the IPC with timeouts. As we have already explained. Timeouts require programming the hardware timer, which is not possible from within a transaction. However, the actual programming of the timer can be deferred to the *wait* preemption point (white-dashed dot #4) and with some additional restructuring of the implementation also to the point in time when the scheduler switches to the next thread to run. Notice, interrupts remain disabled and the timer will be programmed before any user-level code is executed on this core. The programming of the hardware timer is a second case where code must be executed non-transactionally and interleavings must be considered at the instruction level. To become ready to receive, a send timeout can be specified, which the kernel programs by writing to the hardware timer register. Like with the page-table load, we defer this programming of the hardware timer register to the point in time when the transaction is committed.

When enlarging the transaction in the prescribed way, we must of course validate that the transactional state stays small enough to fit in the transaction-storing cache (i.e., L1 in case of Intel Haswell). In addition, we have to ensure that transactions remain small enough to avoid frequent aborts due to conflicts. With the additional preemption point at #4, no capacity aborts occurred and, as we shall see in Sect. IV, the probability of other IPC operations causing retries is little more than $10^{-7}\%$.

C. *Manipulating Page-Table Entries Transactionally*

One uncertainty that remained from the documentation [17] was whether page-table manipulations in Intel Haswell adhere to the transaction semantics, that is, whether page-table walks by one processor causes aborts of transactions that modify the walked page table. We therefore performed a small test, which transactionally updates the page table on one core while accessing the mapped memory on another core, to confirm that the page-table walker actually triggers aborts. As long as this implementation is maintained, only possibly required TLB shootdowns must remain outside the transaction. Otherwise, if the page-table walker bypasses the transaction mechanism and evaluates transactional state, large parts of the kernel's address space implementation would have to be moved out of the transaction because intermediate state would become visible that gets discarded if the transaction aborts.

IV. Evaluation

Similar to other research in the area of hardware transactional memory, there are two main aspects to consider when evaluating transactions in L4/Fiasco.OC: First, whether it is possible to reduce the complexity of the kernel code, and second, whether the performance of the kernel can be improved or not.

A. *Reducing Kernel Complexity*

With HTM, writing synchronized code is easier than with traditional locking mechanisms. This characteristic is

[1] For simplicity, Fig. 1 illustrates only the sender-driven part of the IPC path.

mainly related to two aspects: First, difficult problems such as deadlocks, priority inversion, and convoying do not exist with HTM because transactions do not block during their execution. Instead, they execute optimistically and roll back in case of conflict. Second, the programmer needs not to decide which portions of its code can run in parallel and hence which fine-grain lock to use where. Transaction detect automatically and at the granularity of cache lines, whether data accesses conflict. Hence, it is possible to use transactions also for larger critical sections because the conflict sets are determined dynamically. Still short transaction reduce the likelihood of conflict and the aborts they entail.

The IPC mechanism of the L4/Fiasco.OC microkernel already distinguishes core-local from cross-core communication in its locking mechanisms, by requiring that changes of critical process information is performed on the home core of the modified thread. Hence, if two threads perform an IPC operation while on the same core, no synchronization is needed. To protect locally unsynchronized critical IPC state from inconsistent modification during cross-core IPC, the cross-processor IPC path temporarily stops the partner's core to perform the modification there. This operation requires a comprehensive and time intensive synchronization via *inter processor interrupts* (IPI).

With the introduction of RTM in IPC, we removed the restriction that process information can only be changed on the process' home core. Instead, all modifications are executed transactionally. This way, the flow of executing local and cross-processor IPC have become identical and can be handled in one routine. The only remaining difference is in the way how the transitions from the sender context to the receiver context are realized. While the local IPC case requires only a scheduler activation, the cross-processor IPC case still requires an IPI to trigger scheduling on the remote core.

Hence, we were able to remove most of the complexity of cross-processor IPC path and replaced it with a simpler local IPC path. We expect to be able to make similar changes to other kernel routines and thereby further reduce the complexity of the kernel.

Unfortunately, since Intel®'s RTM extension does not provide any progress guarantee for the transactions, our implementation has a significant drawback. We always have to have a fallback mechanism to guarantee completion of all system calls in case of transaction aborts. Our current approach is to first retry the transaction for a couple of times and then to revert to the traditional cross-processor IPC path, which we could have removed otherwise to safe about 400 lines of code. In general, there is no need to abort all transactions, as demonstrated in IBM's HTM implementation [18], where later transactions cannot abort earlier transactions.

In situations where probabilistic completion and progress guarantees suffice, fall-back mechanisms are not required and the IPC operation could simply be aborted if it did not succeed within a limited number of retries. In Table I, we have collected a statistics to determine the number of retries required. During our performance benchmark (see Section IV-B), $8.64 \cdot 10^{-}5\%$ of the IPC operations failed to commit directly and only $1.05 \cdot 10^{-}7\%$ failed after a second attempt. We did not observe an IPC operation that did not complete after two retries.

TABLE I. STATISTICS ABOUT THE ABORT AND RETRY BEHAVIOR OF THE TRANSACTIONS USED IN THE L4/FIASCO.OC KERNEL

Total	Direct Commit	1 Retry	2 Retries	> 2	Fallback
10,446,981,951	10,446,972,918	9022	11	0	0

B. Performance

Yoo and Leis [19], [20] observed for their benchmarks a general performance advantage of using HTM, except in those that required no synchronization in the first place. To see whether, besides the above reduction in code complexity, IPC benefits from similar advantages when the number of parallel operations increases, we have performed the following experiments on an Intel Haswell i4770 running at 3.4 GHz. We expected significant improvements in the multiprocessor case and low overheads for local IPC.

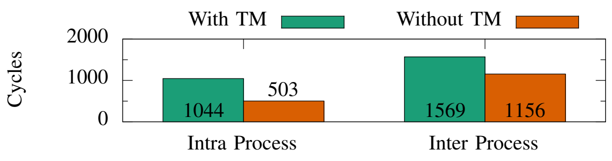

Fig. 2. Minimum number of processor cycles needed to perform a processor local IPC send-receive operation a) within one process (intra process) and b) between two processes (inter process) with the usage of TM and without it. (Deviation in the result is negligible.)

To determine the performance characteristics of local IPC, we measured the costs of transferring an empty message between two threads using an IPC send-receive operation. We compared intra process and inter process communication. As shown in Fig. 2, our implementation introduces a significant overhead of about 107 % for IPC between threads of the same process and of about 35 % for IPC between two processes. Our analysis indicates that the house keeping for the four transactions we need for one IPC send-receive operation introduces this performance decrease. Each transaction costs about 100 cycles.

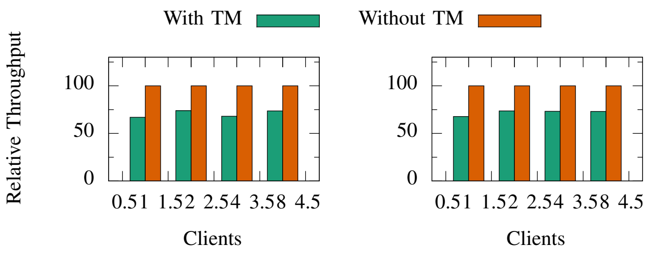

Fig. 3. Average number of full IPC round trips achieved in a second by 1, 2, 4, or 8 clients communicating with one server (left) or equally many servers (right) with the usage of TM relative to the same number without the usage of TM. Deviations in the results are negligible.

While IPC cycle counts reveal raw kernel performance, they generally reveal little insight on application performance. We have therefore also measured two benchmarks, which simulate client-server communication, a scenario common in microkernel-based systems. We measured the relative throughput in IPC send-receive operations between (a) an increasing number of client threads communicating with one server thread and (b) an increasing number of client threads communicating with dedicated server threads. Fig. 3 shows that the transactional implementation introduces a performance degradation between 28 % and 35 % in all scenarios. This overhead is consistent with the results of the raw IPC performance benchmark. In total, the original local IPC implementation, which requires no further synchronization, performs significantly better than transactional IPC.

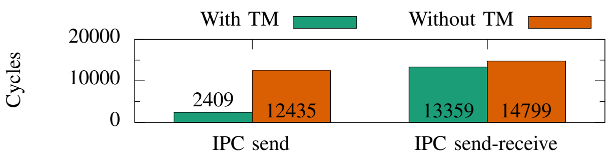

Fig. 4. Minimum number of processor cycles needed to perform a cross-processor IPC operation within one process with a) only sending and b) sending and receiving with the usage of Transactional Memory and without it. Deviations in the results are negligible.

For cross-processor IPC we tried to run a similar benchmark as described above. Unfortunately, this was not possible because this test triggered the RTM implementation bug in Haswell [21]. Our system failed silently. To still provide performance characteristics, we therefore measured the number of processor cycles required for a cross-processor IPC send operation as well as a cross-processor IPC send-receive operation. However, in contrast to the local IPC benchmark, every IPC operation had to wait for a constant time to avoid the above bug. Consequently, the measured values do not present the full potential of our system, but just an indication how future systems behave. As it can be seen in Fig. 4, our implementation performs better than the original code. Especially, the IPC send operation runs up to five times faster than its traditional counterpart. This large difference between the two implementations is mainly because we were able to remove the time expensive IPI from the critical path as we only need it to trigger the rescheduling on the remote core in a *fire-and-forget* fashion. The sender could proceed immediately. For the IPC send-receive operation, we have to wait for one IPI to trigger the scheduling of the receiver and for a second during the reply. As IPI costs dominate IPC send-receive costs, our transactional implementation performs as well but no better than the traditional path. For a saturated server, we expect these costs to be hidden because the server will then find the next request pending when it replies to the current one.

V. SIMPLIFYING THE MULTICORE VERIFICATION TASK

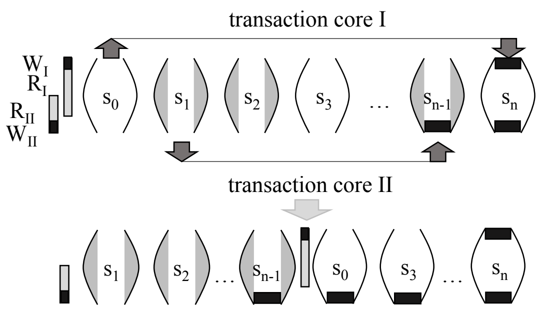

Fig. 5. Parallel interleaved execution of transactional operations exhibits the same visible states as a corresponding sequential execution.

Before we proceed with our argument why we believe that a consequent application of transactions will simplify the multiprocessor verification task to little more than what is required for a uniprocessor kernel, let us clarify our assumptions and goals. Our focus is on verifying multiprocessor kernels, not on translating uniprocessor verification results to a multiprocessor setting, which if at all possible requires additional arguments. We assume transactions to be correct and complete with regard to device side effects. That is, accesses to memory used by the kernel that origin from a device must adhere to the cache protocol and cause aborts if they conflict with transactional kernel state.

Our confidence is based on the following observation. If kernel code executes transactionally, interaction with this code is limited to points in time equivalent to the beginning of the transaction respectively to the time it commits. Any other interaction (by devices or remote cores) will cause an abort and unrolling of transactional state. By “equivalent times” we mean that all interacting write must happen before the transaction reads this data. Our argument, which we are currently transforming into a machine-checked proof, goes as follows. If the majority of the kernel executes transactionally, the trace positions, which characterize the execution of atomic machine instructions, can be rearranged to obtain a trace, which matches the execution behavior of a uniprocessor kernel with selectively preemptible system calls. Instead of considering all possible interleavings at the granularity of atomic machine instructions, it therefore suffices to consider only those interleavings where the instructions inside a transaction execute one after another and without other instructions interleaving. Fig. 5 shows this interleaving and the rearrangement into blocks. It suffices to consider only traces such as the one below, where transactions execute as blocks. Positions of transaction I (white) are combined to a single block and executed after the positions of transaction II (gray).

The rearrangement is possible because we know from the correctness of transactions that cached state becomes only visible if external writes went to a different set of physical addresses than transactional reads or writes. Let R_I, W_I, R_{II} and W_{II} denote these read and write sets for the two transactions. We conclude that for the interleaving of core I and core II, cached state of core II becomes visible only if $W_I \cap (R_{II} \cup W_{II}) = \emptyset$ and likewise for core I if $W_{II} \cap (R_I \cup W_I) = \emptyset$. But then we can shuffle the trace positions such that preserving the order of transactions, all positions of core II (who committed first in this trace) occur before those of core I (who committed last) follow. We realize that these traces are identical with regard to the visible memory updates. Notice in particular that the above address disjointness rules out that core I may depend on the state written by core II (black part in s_0, s_3, s_n) since otherwise core I's transactions would have aborted. But now the trace is identical to a sequential execution of the system calls in a non-preemptive manner while restricting the observation of state to the preemption points.

Notice, for deferred operations, we still require the machinery to verify kernel code at machine granularity. For these, we have to consider all possible interleavings of these instructions and of the transactions at their boundaries. The latter is because we require devices to abort transactions in case of conflict.

VI. Related Work

Ramadan et al. [14] were first to demonstrate the benefit of HTM for synchronizing operating-system code. However, unlike TxLinux, we take a more holistic approach trying to turn every system call into a sequence of transactions to benefit from the simplified interleaving in the verification task. In this regards, our work is more closely related to TxOS by Porter et al. [22] and their attempt to provide transactional kernel behavior for certain mechanisms such as I/O. Of course, there is a large body of work beyond the operating-system kernel. For example, Karnagel et al. [23] and Leis et al. [20] use RTM to improve the performance of in-memory database systems, Kleen [24] extends the GNU C pthreads library to use HTM for transactional synchronization. Ariel et al. [25] formally specify HTM for the purpose of verifying correctness of their implementation, a task which Gupta et al. [26] extend to HTM implementations with non-transactional writes as for example supported in AMD's ASF proposal [27]. To the best of our knowledge this is the first work to realize how a consequent application of HTM can simplify the verification task.

VII. Conclusion and Future Work

In this paper, we have shown how L4/Fiasco.OC's CPU-local IPC path can be converted into an almost completely transactional multiprocessor path. For scenarios where stochastic completion guarantees suffice, we observe a performance improvement in the cross processor case at the costs of significantly increasing uniprocessor costs by almost a factor of 2, and requiring retries in $1.05 * 10^{-7}$ % of all cases. In addition, we have shown how a consequent re-implementation using transactions may simplify the multiprocessor verification task by allowing similar reasoning for the transactions as in the uniprocessor case. Obvious directions for future work include a re-evaluation on newer-generation hardware, progress guarantees for HTM and the lifeness guarantees they entail, and an extension of the described approach to applications and user-level servers.

Acknowledgment

This work is in part funded by the German research council DFG through the cluster of excellence "Center for Advancing Electronics Dresden" cfaed and DFG-SPPEXA's project FFMK.

References

[1] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. ACM, 2009, pp. 207–220.

[2] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta, "Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework," in *IEEE Security and Privacy*, Oakland, 2013.

[3] S. Brookes, "A semantics for concurrent separation logic," *Theor. Comput. Sci.*, vol. 375, no. 1-3, pp. 227–270, Apr. 2007.

[4] S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs," *Acta Informatica*, vol. 6, pp. 319–340, 1976.

[5] ——, "Verifying properties of parallel programs: an axiomatic approach," *Communications of the ACM*, vol. 19, pp. 279–285, 1976.

[6] M. Herlihy and J. E. B. Moss, *Transactional memory: Architectural support for lock-free data structures*. ACM, 1993, vol. 21, no. 2.

[7] N. Shavit and D. Touitou, "Software transactional memory," *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.

[8] T. Harris and K. Fraser, "Language support for lightweight transactions," in *ACM SIGPLAN Notices*, vol. 38, no. 11. ACM, 2003, pp. 388–402.

[9] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson, "Architectural support for software transactional memory," in *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, 2006.

[10] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee, "Software transactional memory: Why is it only a research toy?" *Queue*, vol. 6, no. 5, p. 40, 2008.

[11] R. Haring and B. Team, "The blue gene/q compute chip," in *The 23rd Symposium on High Performance Chips (Hot Chips)*, vol. 4, 2011, pp. 125–180.

[12] Intel®, "Intel® Architecture Instruction Set Extensions Programming Reference," https://software.intel.com/sites/default/files/m/9/2/3/41604, 2012.

[13] R. Rajwar and J. R. Goodman, "Speculative lock elision: Enabling highly concurrent multithreaded execution," in *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 2001, pp. 294–305.

[14] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel, "MetaTM/TxLinux: transactional memory for an operating system," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 92–103, 2007.

[15] J. Liedtke, "On μ-kernel construction," in *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, Copper Mountain Resort, CO, Dec. 1995, pp. 237–250.

[16] "The Fiasco.OC Microkernel," http://os.inf.tu-dresden.de/fiasco/, 2014, [Online, accessed 27-Nov-2014].

[17] Intel®, "Intel® 64 and IA-32 Architectures Optimization Reference Manual," http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, 2014.

[18] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael, "Evaluation of Blue Gene/Q hardware support for transactional memories," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 127–136.

[19] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of intel® transactional synchronization extensions for high-performance computing," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 19.

[20] V. Leis, A. Kemper, and T. Neumann, "Exploiting hardware transactional memory in main-memory databases," in *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*. IEEE, 2014, pp. 580–591.

[21] Intel®, "Haswell Specification Update," http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e3-1200v3-spec-update.pdf, 2014.

[22] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel, "Operating system transactions," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009.

[23] T. Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Legler, B. Schlegel, and W. Lehner, "Improving in-memory database index performance with intel transactional synchronization extensions," in *in Proc. 20th Int'l Symp. High-Performance Computer Architecture*, 2014.

[24] A. Kleen, "Lock elision in the gnu c library," http://lwn.net/Articles/534758/, 2013, [Online, accessed 29-Now-2014].

[25] A. Cohen, J. O'Leary, A. Pnueli, M. R. Tuttle, and L. D. Zuck, "Verifying correctness of transactional memories," in *Formal Methods in Computer Aided Design, 2007. FMCAD '07*, Nov 2007, pp. 37–44.

[26] A. Cohen, A. Pnueli, and L. Zuck, "Mechanical verification of transactional memories with non-transactional memory accesses," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, A. Gupta and S. Malik, Eds. Springer Berlin Heidelberg, 2008, vol. 5123.

[27] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier *et al.*, "Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 27–40.