A low-overhead profiling and visualization framework for Hybrid Transactional Memory

(Currently under review, please do not circulate)

***Abstract*—Multi-core prototyping presents a good opportunity for establishing low overhead and detailed profiling and visualization in order to study new research topics. In this paper, we design and implement a low execution, low area overhead profiling mechanism and a visualization tool for observing Transactional Memory (TM) behaviors on FPGA. To achieve this, we non-disruptively create and bring out events on the fly and process them offline on a host. There, our tool regenerates the execution from the collected events and produces traces for comprehensively inspecting the behavior of interacting multithreaded programs. With zero execution overhead for hardware TM events, single-instruction overhead for software TM events, and utilizing a low logic area of 2.3% per processor core, we run TM benchmarks to evaluate various different levels of profiling detail with an average runtime overhead of 6%. We demonstrate the usefulness of such detailed examination of SW/HW transactional behavior in two parts: (i) we port the STAMP application Intruder to Hybrid TM to speed it up by 24.1%, and (ii) we closely inspect transactions to point out pathologies such as repetitive aborts, killer transactions and starvation. The SW/HW profiling and event visualization infrastructure that we present offers possibilities of extension to many other directions.**

I. Introduction

The low performance of legacy architectural simulator software for investigating new generation Chip Multi-Processor (CMP) architectures is being addressed in a few ways: the development of new parallel simulators [1], parallelization efforts for sequential simulators [2], and prototype/emulation implementations on reconfigurable fabric [3], [4]. FPGAs were proven successful in accelerating simulations working in concert with a host computer, as well as FPGA-only multi-core MPSoC implementations [5], [6].

A proposal that has drawn considerable attention for applying parallel programming to these new shared-memory CMPs has been the use of Transactional Memory (TM). In TM, shared critical sections of a program are encapsulated inside `atomic{}` transactions, and the underlying TM mechanism automatically detects data inconsistencies and aborts and restarts transactions. This ensures deadlock-free transactional code segments to run in an all-or-none manner, saving the programmer from explicitly dealing with locks.

TM support can be provided by flexible but slower software libraries (STM), or by using fast, dedicated hardware (HTM). An HTM system is usually bound by some sort of capacity constraints, i.e. the hardware can only handle transactions with specific characteristics. Hybrid TM aims to provide the best of two worlds. Transactions first attempt to run on the dedicated TM hardware and fall back to software when it is not possible to complete the transaction in hardware (e.g. when resources are exceeded) [7]. ATLAS and later Configurable TM were the first systems to support hardware TM on FPGA[8], [9]. Hardware acceleration proposals by using Bloom filters for TM were also investigated [10], [11], as well as Hybrid TM on FPGA [12].

But another issue remains: Performance and scalability are both important for a successful adoption of TM. Profiling executions in detail is absolutely necessary to have a correct understanding of the qualities and the disadvantages of different implementations and benchmarks. A low-overhead, high-precision profiler that can handle both hardware and software TM events is required. Up to now, no comprehensive profiling environment supporting STM, HTM and Hybrid TM has been developed.

Due to its flexibility and extensibility, an FPGA is a very suitable environment for implementing profiling mechanisms and offers a unique advantage based on three main aspects. Firstly, compared to a software simulator, there are no overheads in simulation time due to the special hardware added. Moreover, FPGAs emulate real hardware interfacing real storage or communication devices and exhibit a higher degree of fidelity than software simulators. Second, the relative area overhead for implementing extra profiling circuitry can be very low, and the throughput high, as we demonstrate. Third, because of its customizability, we are free to extend the architecture with new application-specific instructions for profiling. We use this flexibility to reduce the software overheads of the profiling calls added to the programs, as we will show with the new `event` instruction.

Using these advantages in utilizing FPGAs for multicore prototyping, we address three main issues:

- STM application profiling can suffer from high overheads, especially with higher levels of detail (e.g. looking into every transactional load/store). Such behavior may influence the application runtime characteristics and can affect and alter the interactions of threads in the program execution, producing unreliable conclusions.
- Hardware extensions for a simulated HTM system and a software API was suggested by the flagship HTM-only profiling work, TAPE [13]. It is useful for pinpointing and optimizing undesired HTM behaviors, but incurs some overhead due to API calls and saving profiling

data to RAM. We argue that using an FPGA platform, hardware events can come for free.
- Visualizing executions in a threaded environment can be an efficient means to observe transactional application behavior, as was looked into in the context of an STM in C# [14]. A profiling framework facilitates capturing and visualizing the complete execution of TM applications, depicting each transactional event of interest, created either by software or by hardware.

The purpose of this work is to address these shortcomings and to develop a complete monitoring infrastructure that can accept many kinds of software and hardware transactional events with low overhead in the context of a Hybrid TM system on FPGA. This is the first study to profile and visualize a Hybrid TM scenario, with the capabilities to examine in detail how hardware and software transactions can compliment each other.

For gathering online profiling information, first we describe (i) profiling hardware that supports generating TM-specific hardware events with zero execution overhead, and (ii) a key extension to the Instruction Set Architecture (ISA) called the `event` instruction that enables a low, single-cycle overhead for each event generated in software. Later, a post-processing tool that generates traces for the threaded visualization environment Paraver [15] is engaged. The resulting profiling framework facilitates to visualize, identify and quantify TM bottlenecks: It allows getting insights into the interaction between the application and the TM system, and it helps to detect bottlenecks and other sub-optimal behavior in the process. This is very important for optimizing the application for the underlying system, and for designing faster and more efficient TM systems.

Running full TM benchmarks, we compare different levels of profiling and their overheads. Furthermore, we show visualization examples that can lead the TM programmer/designer to make better and more reliable choices. As an example, we demonstrate how using our profiling mechanism, the Intruder benchmark from STAMP [16] can be ported to best utilize Hybrid TM resources. Such a HW/SW event infrastructure can be easily modified to examine in detail full complex benchmarks in any research domain.

The next section presents the design objectives and the TMbox system used, a Hybrid TM implementation on FPGA. Section III explains the infrastructure that implements the profiling mechanism in order to produce meaningful HTM/STM events and to process them offline on a host. Section IV presents overhead results running TM applications and example traces illustrating the features of our profiling mechanism. Section V concludes the paper.

II. Design Objectives

To get a complete overview of TM behavior, it is vital to have a system with low impact on application runtime characteristics and behavior, otherwise the optimization hints gathered could cause a misguided attempt to ameliorate the system. Since the profiling infrastructure will be designed on actual hardware, we can not implement unrealistic behavior, and the new circuitry has to map well on the reconfigurable fabric, with minimal area and routing overheads. We made three key design choices to get low execution and low area overhead and not to disturb placing and routing on the FPGA:

- Non-intrusively gather and transfer runtime information by implementing the monitoring hardware separately. Build the monitoring infrastructure only by attaching hardware hooks to the existing pipeline.
- Make use of the flexibility of the ISA and the GCC toolchain to add new instructions to the ISA to support STM events with low profiling overhead.
- Use little area on the FPGA by adding minimal extra circuitry, without widening the buses or causing extra routing overheads. To transfer the events non-disruptively, instead of adding a new events network, utilize the idle cycles on an already-existing network. To be absolutely non-intrusive, give higher priority to real packets, buffer the event packets and send them only when there is no traffic.

A. TMbox architecture

For this work, a multicore prototype that can fit many cores on a single chip with support for STM, HTM and Hybrid TM was needed. A completely modifiable architecture would help us change the ISA and the software toolchain. The TMbox system [12] features an open-source Hybrid TM implementation of up to 16 MIPS soft processor cores interconnected with a bi-directional ring bus on a Virtex5-155t FPGA of the BEE3 prototyping platform [17].

The TMbox system features the best-effort Hybrid TM proposal ASF [18], which is used with TinySTM [19], a lightweight word-based STM library. The transactions are first started in hardware mode with a *start tx* instruction. A transactional load/store causes a cache line to be written to the special TM Cache. *Commit tx* ends the atomic block and starts committing it to memory. An invalidation of any of the lines in the TM Cache causes the current transaction to be aborted (Figure 1). Transactions are switched to software mode when (i) TM Cache capacity, which is by default 16 cache lines (256 bytes) is exceeded, (ii) the abort threshold is reached because of too much contention in the system or (iii) the application explicitly requires it, e.g. in case of a system call or I/O inside of a transaction. In software mode, the STM library is in charge of that transaction, keeping track of read and write sets and managing commits and aborts. This approach enables using the fast TM hardware whenever it is possible, but meanwhile to have an alternative way of processing transactions that are more complex or too large to make use of the TM hardware.

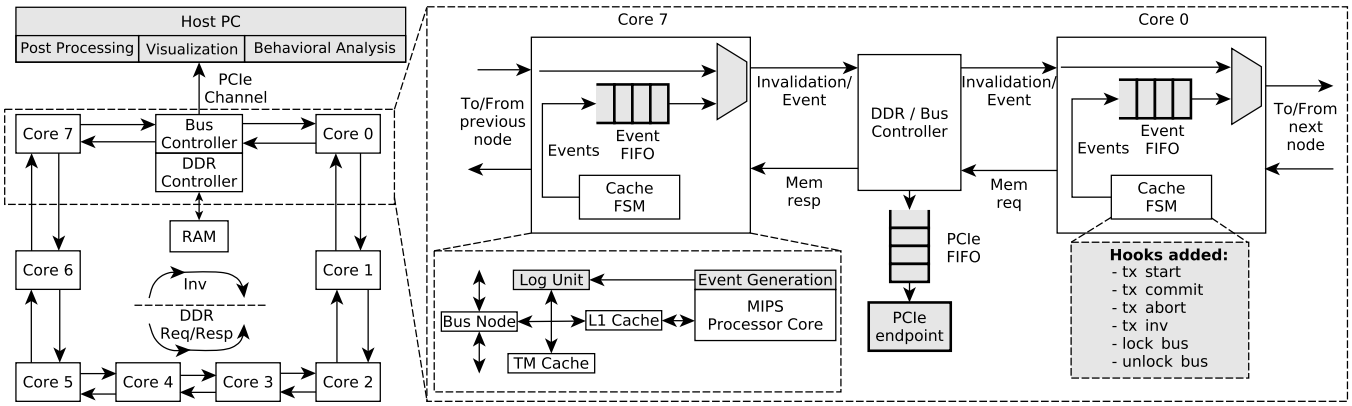


Figure 1. An 8-core TMbox system block diagram and modifications made to enable profiling (shaded).

The hardware TM implementation supports lazy commits: Modifications made to the transactional lines are not sent to memory until the whole transaction is allowed to successfully commit [12]. However, TinySTM supports switching between eager and lazy commit and locking schemes as we will look into with software transactions in Section IV-B.

A bidirectional ring bus interconnects the CPUs of the TMbox. This design decision allows for an FPGA-friendly implementation: Short wires ease the placement on the chip and relax timing and routing constraints (a property that we do not want to break). It also helps keeping the caches coherent; CPU requests and DDR responses move counter-clockwise, whereas invalidation signals that are generated by writes to the DDR move in the opposite direction (Figure 1). Whenever a write request meets an invalidation to the same address on any node of the ring bus, it gets cancelled. Meanwhile, the caches on each core also snoop and discard the lines corresponding to the invalidation address, effectively enabling system-wide cache coherency.

B. Network reuse

To cause as little area and routing overhead as possible, we discard the option of adding a dedicated network for events. Instead, we choose to piggyback on an existing network. More particularly, we utilize the idle cycles on the less-frequently-used invalidation bus. However, we do not want to disturb the execution by causing extra network congestion, so we give a lower priority to profiling events by first buffering them and transferring them only when a free cycle on the bus is detected. This way, the profiling packets do not disrupt the traversal of the already-existing invalidation packets in any way.

Although this approach might be somewhat specific to the TMbox architecture, we believe that the methodology of always first buffering the created events, and injecting them in the network only on a free slot could be applied to different network types, as well. Future work could address how to implement similar functionality on a different network type, such as a mesh or a tree.

Message Header		Message Data		
2 bits	4 bits	20 bits	4 bits	4 bits
Message Type	CPU Sender ID	Timestamp (δ-encoded)	Event Type	Event Data

Figure 2. Event format for the profiling packets.

A disadvantage of this approach is that the fixed message format of the invalidation ring bus has to be matched. Another drawback is watching out for buffer overflows. The next section explains in detail how the design decisions affected the way the TMbox system was modified to support creating, propagating, transferring and post-processing timestamped TM events.

III. The Profiling and Visualization Infrastructure

The profiling and visualization framework consists of performing three steps on the FPGA and the final step on the host computer. First, the TM behavior of interest is decomposed into a small, timestamped event packet containing information about the change of state. Second, the event is propagated on the bus to the central Bus Controller node. Third, from the node, it is transferred on the fly by PCI Express (PCIe) to a host computer. Finally, the post-processing application running on the host parses all event packets and re-composes them back to meaningful, threaded events with absolute timestamps, and creates the Paraver trace of the complete application.

A. Event specification and generation

1) HTM events: The event generation unit (Figure 1) monitors the TM states inside the cache Finite State Machine (FSM) of the processor, generating events whenever there is a state change of interest, e.g. from *tx start* to *tx commit*. Figure 2 shows the format of an event in a detailed way. The timestamp marks the time when an event occurred, and is delta-encoded: only the time difference in cycles between two consecutive events is sent. This space-efficient encoding

allows a temporal space of about a million cycles (20 ms @ 50 MHz) between two events occurring on a processor. The event data field stores additional data available for an event, for instance the cause of an abort (e.g. capacity overflow, software-induced, invalidation).

Due to the 4-bit wide event type, we can define up to 16 different event types. Some of the basic event types defined for hardware transactions include: *tx start*, *tx commit*, *tx abort*, invalidation, lock/unlock ring bus (for performing commits). These hardware events come with zero execution overhead, since the profiling machinery works in parallel to the cache FSM. Our infrastructure supports easily adding and modifying events, as long as there is a free event type encoding available in hardware.

The fact that we can only use 20 bits for the timestamp in order to match the predefined message format can cause wraparounds, so the Paraver threads can fail to be properly synchronized. To address this, we added an extra event type that is very rarely used. When it detects a timestamp counter overflow, in the next event, it also sends the number of idle timestamp overflows occurred along with the timestamp. Although the bus and the event messages could also be widened, we opted for modifying the existing hardware as little as possible to accomplish as low overhead as possible. This is also the reason why we eliminated the option of having a separate bus only for the events.

2) Extending the ISA with STM events: For generating low overhead events from software, an `event` instruction was added to the processor model by modifying the GNU Compiler Collection (GCC) and the GNU Binutils suite (GAS and objdump). The `event` instruction creates STM events with a similar encoding to the HTM events, supporting up to 16 different software events that are implemented in special event registers. Little hardware with a small area overhead of 32 LUTs/core had to be added: extending the opcode decoder, some extra logic for bypassing the data, and multiplexing it into the event FIFO. More complex processor architectures might need to be more heavily modified to add new instructions and registers. However, the ability to create such precise events from software with single-instruction overhead is a very powerful tool for closely inspecting a variety of behaviors. Software events can be modified simply by storing the wire/register/bus values of interest in event registers and by reading them from software with an `event` call.

Similar to the "free" hardware events discussed earlier, the events generated in software also utilize the same event FIFO. However, software events have some execution overhead: one instruction per event. In the next section, we compare execution overheads of this approach to software-only events created on a commodity machine, and demonstrate that utilizing the `event` instruction actually contributes to a smaller overhead in runtime.

B. Event propagation on the bus

A logging unit captures events sent by the event generation unit located in each core. Here, the event is timestamped using delta encoding and enqueued in the event FIFO. As soon as an idle cycle is detected on the invalidation bus, the event is dequeued and transferred towards the bus controller.

To prevent a disturbance of program runtime behavior, the profiling events are classified as low-priority traffic on the invalidation ring bus. So, invalidation packets always have higher priority. Consequently, when the ring bus is busy transferring invalidation messages, it is necessary to buffer the generated events. To keep the events until a free slot is found, event FIFOs (one BRAM each) were added to each core, as shown in Figure 1.

The maximum rate at which an invalidation can be generated on the TMbox is once every three cycles. The DDR controller can issue a write every three cycles, which translates into an invalidation message that has to traverse the whole bus. Therefore, for an 8-core ring setup, the theoretical limit of starting to overflow into the event FIFOs is when one event is created by all cores every 12 cycles. Using a highly contended shared counter written in MIPS assembly, we observed that the FIFOs never needed to have more than 4 elements. This is the worst case behavior: TM programs written in high level languages incur further overhead through the use of HTM/STM abstraction frameworks and thus would actually exhibit a smaller pressure on the buffers of the monitoring infrastructure.

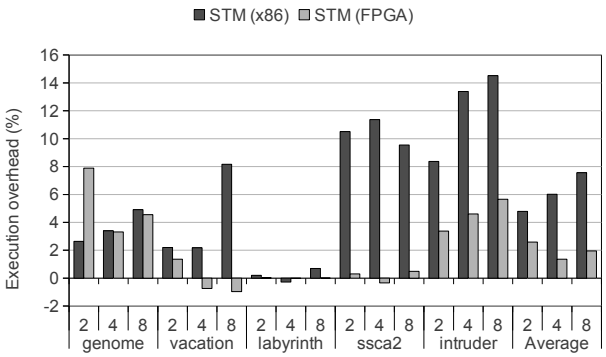
Changing the network type for the system would imply the need to modify the infrastructure to look for and to use empty cycles or to add another data network for events which would come with routing issues and area overhead. While with a dedicated event bus this step would have been trivial, better mapping on the FPGA requires a lower cost approach. Therefore, we reuse the already-available hardware and only incurring area overhead by placing FIFOs to compensate for traffic congestion.

C. Transfer of events to the host

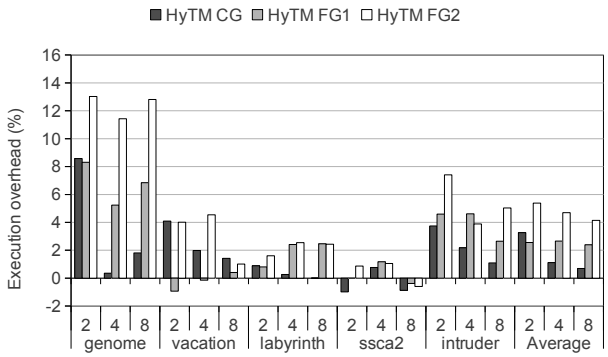
To transfer the profiling packets, we use a PCIe connection that outputs data at 8 MB/sec, coupled with a large PCIe output FIFO placed to sustain temporary peaks in profiling data bandwidth. In our executions, we did not experience overflows and lost packets, although the throughput of the PCIe implementation is obviously limited. A suitable alternative for when a much greater amount of events are created (e.g. at each cache miss/hit), might be to save to some large on-chip DDR memory instead of transferring the events immediately. However this memory should preferably be apart from the shared DDR memory of the multicore prototype, for reasons of non-disruptiveness. The profiling data might reach sizes of many MB, so saving the profiling data on on-chip BRAMs is not a viable option.

Table I
AREA OVERHEAD PER PROCESSOR CORE AND THE TRACKED EVENTS IN DIFFERENT PROFILING OPTIONS

Profiling Type	Area Overhead (per CPU core)	Actions Tracked
STM-only (x86 host)	NONE	SW start tx, SW commit tx, SW abort tx
STM-only	32 5-LUTs + 1 BRAM	SW start tx, SW commit tx, SW abort tx
HTM-only	129 5-LUTs + 1 BRAM	HW start tx, HW commit tx, HW abort tx, lock bus, unlock bus, HW inv, HW tx r/w, HW PC
Hybrid TM (CG)	129 5-LUTs + 1 BRAM	HTM-only + STM-only
Hybrid TM (FG1)	129 5-LUTs + 1 BRAM	Hybrid TM (CG) + SW tx r/w + tx ID
Hybrid TM (FG2)	129 5-LUTs + 1 BRAM	Hybrid TM (FG1) + SW inv + SW PC



(a) Runtime overhead (%) for STM (x86) vs. STM (FPGA), in different core counts and applications. (avg. 20 runs)



(b) Runtime overhead (%) for different Hybrid TM profiling levels, core counts and applications. (avg. 20 runs)

Figure 3. STAMP-Eigenbench Benchmark Overheads

D. Post-processing and execution regeneration

After the supervised application has terminated, and all events have been transferred to the host machine, they are fed in to the Bus Event Converter. This program, which we implemented in Java, (i) parses the event stream, (ii) rebuilds TM and application states, and (iii) generates statistics that are compatible for visualizing with Paraver [15]. The mature and scalable program Paraver was originally designed for the processing of Message Passing Interface (MPI) traces, which we adapted to visualize and analyze TM events and behavior. Our post-processing program converts the relative timestamps to absolute timestamp values and re-composes the event stream into meaningful TM states. At this point, additional states can also be created, depending on the information acquired through the analysis of the whole application runtime. This removes the need to modify the hardware components to add and calculate new states and events during the execution, and allows for a more expressive analysis and visualization.

IV. EXPERIMENTAL EVALUATION

In order to demonstrate the low overhead benefits of the monitoring framework proposed, we ran STAMP [16] applications using Eigenbench [20], a synthetic benchmark for TM mimicry. STAMP is a well-known TM benchmark suite with a wide range of workloads, and Eigenbench imitates its behavior in terms of number and size of transactions, read/write sets and many other orthogonal characteristics. We used the parameters for five STAMP benchmarks provided by the authors of Eigenbench. We compare runtime overheads of the profiling hardware to an STM-only implementation which generates runtime event traces in a way comparable to our FPGA framework. This version called STM (x86) tracks each transactional start, commit, and abort events in TinySTM running on a Westmere[1] server. The events are timestamped and placed in a buffer, which is written to a thread-local file handle.

Along with STM and HTM profiling, we engage three levels of Hybrid TM profiling to enable both light and detailed profiling options. The coarse-grained (CG) version features the typical HTM and STM events (Table I). Besides the most common *start tx*, *commit tx* and *abort tx* events, we also look at invalidation events and the overheads of locking/unlocking the bus for commits (part of the HTM commit behavior of TMbox). Additionally, there are two fine-grained profiling options that are implemented through the `event` mechanism in software. FG1 includes tracking all transactional reads and writes, also useful for monitoring readset and writesets. It also keeps transaction IDs, which are needed for dynamically identifying atomic blocks and associating each transactional operation with them.

[1]OS is Linux version 2.6.32-29-server (Ubuntu 10.04 x86_64).

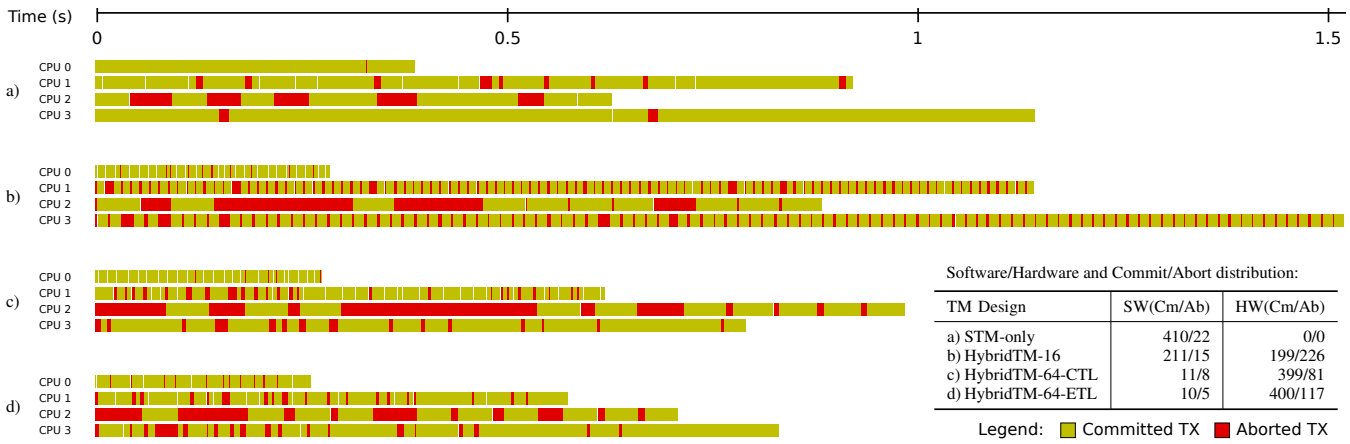


Figure 4. Improving Intruder-Eigenbench step by step from an STM-only version to utilize Hybrid TM appropriately.
STM-only: Here, the TM application is profiled with FG1 level profiling, where a count of the number of read/write events for each transaction is kept. By analyzing the profiled data, we discover a certain repetition of small transactions which can benefit from HTM acceleration. However, there also exist very large transactions, suggesting that an HTM-only approach is not feasible.
Hybrid-TM-16: Introduces HTM with a 16-entry TM Cache per processor core, so this trace depicts both STM and HTM events. CPU 0 seems to have benefited from using HTM (as shown in the table in Figure 4), although there are still some small transactions on other CPUs that might fit if the hardware buffers are increased in size. Please note that a poorly-configured Hybrid TM can end up showing worse performance than an STM.
Hybrid TM-64-CTL: Uses a larger, 64-entry TM Cache. Here, CPUs 1 and 3 also start utilizing HTM efficiently, causing the software transactions to reduce in number. However, CPU 2 suffers from long aborting transactions and a large wasted work. Looking at the available software TM options, we switch from Commit Time Locking (CTL) to Encounter Time Locking (ETL) to discover some conflicts early and to decrease the abort overheads.
Hybrid TM-64-ETL: On overall, a 24.1% speedup in execution time when moving from STM-only to Hybrid TM-64-ETL was observed. Although there are only 3 fewer aborts in software now, they cause much less wasted work, helping the application to run faster to completion.

In addition, the maximum profiling level FG2 that we implemented features source code identification, a mechanism for monitoring conflicting addresses and their locations in the code. For enabling this, a `JALL` (Jump And Link and Link) instruction was added to the MIPS ISA. This extends the standard `JAL` instruction by storing an additional copy of the return address, which is kept as a reference to be able to identify the Program Counter (PC) of the instruction that is responsible of the subsequent transactional read/write operations in TinySTM. This way, a specific event with that unique PC is generated by the transactional operations in these subroutines, effectively enabling us to identify the source code lines with low overhead.

A. *Runtime and area overhead*

In Figure 3a, STM (x86) profiling overhead was compared to our FPGA framework with the same level of profiling detail (STM-only). The overhead introduced by the FPGA implementation is less than half of the STM (x86) overhead, on average. This is largely due to adding the `event` instruction to the ISA to accomplish a single instruction overhead per software event. Please note that if the transactional read and write events were tracked additionally, we would expect a larger slowdown for STM (x86).

Figure 3b shows the extra overhead that our FPGA profiler causes by turning on all kinds of TM profiling capabilities. Almost half a million events were produced for some benchmarks. With the highest level of detail, the average profiling overhead was less than 6% and the maximum 14%.

When the transactions can be run on the dedicated hardware, as in the case of SSCA2, the overall profiling overhead is lower. This is because hardware events come “for free” and less work has to be done in software, where there is some overhead. Therefore, the success of the Hybrid TM drives that of the TM profiling machinery. The higher the percentage of transactions that can complete in hardware, the faster and more efficient the execution of the TM program is, and the more lightweight is the profiling. Conversely, Genome has many transactions that never fit the dedicated TM hardware and exhibit higher overheads. For future work, we want to investigate new techniques that could allow zero runtime overhead in STM profiling. This would also avoid the interferences caused by profiling in the normal program behavior, which we observed in the case of Vacation.

Interesting cases of low Hybrid TM CG overheads appear when running Genome with 4 threads and SSCA2 with 2 and 8 threads. This behavior is due to the specific Eigenbench parameters for STAMP benchmarks, eg. Genome’s parameter values for CPU 3 are huge and cause the application to behave much differently for 4 threads than for 2 threads.

The inclusion of profiling hardware to TMbox results in a 2.3% increase in logic area, plus the memory needed to implement the event FIFO (1 BRAM) for each processor core. The fixed area overhead of the PCIe endpoint plus the PCI_FIFO occupies 3978 LUTs and 30 BRAMs.

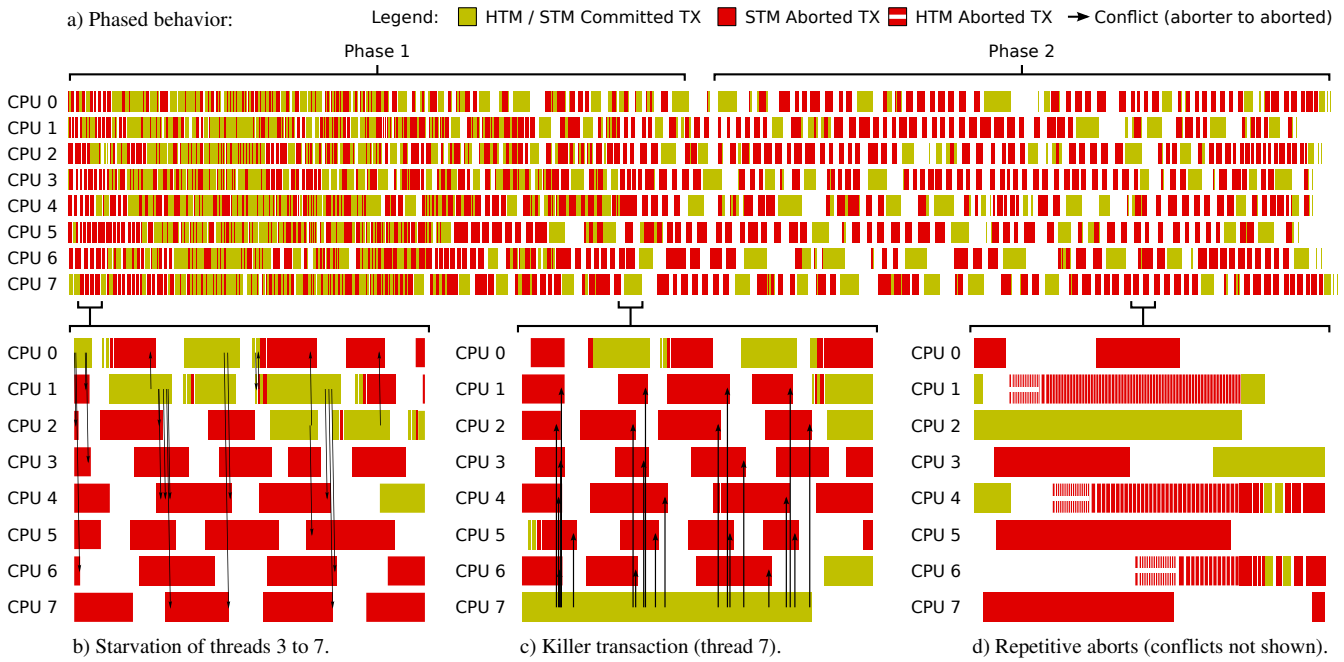


Figure 5. Example traces showing phased behavior and transactional pathologies in Intruder, a network intrusion detection algorithm that scans network packets for matches against a known set of intrusion signatures.
The benchmark consists of three steps: capture, reassembly, and detection. The main data structure in the capture phase is a simple queue, and the reassembly phase uses a dictionary (implemented by a self-balancing tree) that contains lists of packets that belong to the same session.
a) Phased behavior: A program does not always exhibit the same behavior throughout its execution, and in terms of TM might show different phases of high aborts, shorter transactions, or serialization. In the first half of this trace, there are more transactions and parallelism. Here, the packets are being constructed by all the threads and there is not enough complete data to process for detection. In the second half of the execution, complete packets are ready for the detector function, which generates less (but larger) transactions in number, which results in more conflicts among them. Dynamic switching mechanisms would be suitable for treating adequately phased behavior in TM applications.
b) Starvation: A clear example of starvation on CPUs 3, 5, 6 and 7 (towards the beginning of the benchmark) is shown.
c) Killer transaction: Illustrates a single transaction (CPU 7) aborting six others. After it commits, other CPUs can finally take the necessary locks and start committing successfully.
d) Repetitive aborts: Demonstrates the pathology of repetitive aborts and its effect on the execution, as in [21]. Finding the optimal abort threshold (to switch to STM mode) could be important in such cases.

B. Improvement Opportunities

In this section, we present sample Paraver traces for the Intruder benchmark to demonstrate how our low overhead profiling infrastructure can be useful in analyzing TM benchmarks and systems. First, we run the Eigenbench-emulated Intruder and suggest a simple methodology to improve the application's execution for the appropriate usage of TM resources. Next, to visualize TM behavior and pathologies from real application characteristics, we depict some example traces running the actual, non-emulated Intruder benchmark from the STAMP suite.

1) Intruder-Eigenbench: Figure 4 shows the four traces of a simple refinement process using our profiling mechanism. By running the STAMP application Intruder with 4 CPUs, we attempt to derive the best settings both in hardware and in software for running this application in Hybrid TM mode. The program has to complete a total of 410 transactions on four threads. Around 300,000 events are generated in the highest profiling mode for this benchmark. On overall, a 24.1% improvement in execution time was observed when moving from STM-only to Hybrid TM-64-ETL. This final version of Intruder is able to utilize both the TM hardware and the software TM options better.

2) Intruder-STAMP: To pinpoint real application behavior, the actual non-emulated Intruder from STAMP was ran with 128 attacks [16]. Intruder is an interesting benchmark in the sense that (i) it contains a mix of short and long transactions that can sometimes fit in the dedicated transactional hardware, and other times overflow, (ii) typically has a high abort rate which is interesting for TM research, (iii) exhibits real transactional behavior, such as I/O operations inside transactions, and (iv) demonstrates phased behavior, which shows an inherent advantage of our visualization infrastructure.

Figure 5 describes and depicts phased behavior and some examples of different pathologies that can be discovered thanks to the profiling framework. Some solutions to these problems include rewriting the code, serialization, taking pessimistic locks or guiding a contention manager that can take appropriate decisions.

Additionally, our infrastructure could also perform the following actions automatically:

Suggest HW/SW partitioning for transactions: Some transactions are more suitable to run in software and others in hardware. This way, we can avoid the wasted work caused by the transactions that are sure to abort in HTM mode (because of overflow, I/O, etc.).

Propose which locking/versioning strategy to use: CTL vs. ETL, or lazy versioning vs. eager versioning could be dynamically switched in flexible STM schemes. The application or the TM infrastructure can choose to use one mechanism over another, either statically or dynamically, by looking at how early the aborts happen, transaction sizes and other relevant data.

Debugging: By using software programmable `events`, the register values in hardware can quickly be brought up to the software layer and analyzed there.

Additionally, by modifying the application software and the post-processing application, and adding events of interest, various advanced profiling information can be reached by analysis. Some examples are to draw sets/tables of conflicting `atomic{}` blocks, or read/write sets. Profiling the reasons of the aborts gives a better idea of which transactions are frequent aborters of which other transactions, which could help contention management schemes. Up until recently, such advanced profiling was studied in the context of STMs in Java [22] and Haskell [23].

V. Conclusions

An FPGA, for its flexibility in programming and its speed, is a convenient tool for the customization of hardware and application-specificity. Based on this, we have built the first profiling environment capable of precise visualization of HTM, STM and Hybrid TM executions in a multi-core FPGA prototype. We have used a post-processing tool for events and Paraver for their interactive visualizations. Taking into consideration non-intrusiveness and low overhead, the extra hardware added was small but efficient. It was possible to run STAMP TM benchmarks with maximum profiling detail inside the 14% overhead limits. On average, we incurred half the overhead of an STM-only software profiler. Our infrastructure also proved successful to port the Intruder benchmark to use Hybrid TM and get a speedup of 24.1%, and to detect bottlenecks and transactional pathologies.

The profiling framework could be easily adapted to work for any kind of multicore profiling and visualization, and with other state-of-the-art shared memory hardware proposals such as speculative lock elision, runahead execution or speculative multithreading. The event-based framework created in this project can be easily extended to enable the analysis of various processor core functionalities such as ALU, TLB and cache operations, locking behavior or memory access patterns, which can be useful for the construction of adaptive and self-optimizing systems.

References

[1] J. E. Miller, H. Kasture, G. Kurian, C. G. III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," *HPCA '10*, pp. 1–12, 2010.

[2] "Wind river simics," http://www.windriver.com/products/product-notes/PN_Simics_0410.pdf.

[3] N. Dave, M. Pellauer, and J. Emer, "Implementing a functional/timing partitioned microprocessor simulator with an FPGA," *WARFP*, 2006.

[4] E. S. Chung, E. Nurvitadhi, J. Hoe, B. Falsafi, and K. Mai, "A complexity-effective architecture for accelerating full-system multiprocessor simulations using fpgas," in *FPGA '08*.

[5] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators," in *MICRO 40*, 2007, pp. 249–261.

[6] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanović, "RAMP gold: An FPGA-based architecture simulator for multiprocessors," in *DAC '10*, 2010.

[7] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory," *ASPLOS'06*.

[8] N. Njoroge *et al.*, "ATLAS: a chip-multiprocessor with transactional memory support," in *DATE '07*, pp. 3–8.

[9] C. Kachris and C. Kulkarni, "Configurable transactional memory," in *FCCM '07*, April 2007, pp. 65–72.

[10] M. Labrecque, M. Jeffrey, and J. Steffan, "Application-specific signatures for transactional memory in soft processors," in *ARC 2010*, pp. 42–54.

[11] C. Thacker, "Hardware Transactional Memory for Beehive." MSR Silicon Valley, 2010.

[12] N. Sonmez, O. Arcas, O. Pflucker, O. S. Unsal, A. Cristal, I. Hur, S. Singh, and M. Valero, "TMbox: A flexible and reconfigurable 16-core hybrid transactional memory system," in *Proc. FCCM '11*, pp. 146–153.

[13] H. Chafi, C. C. Minh, A. McDonald, B. D. Carlstrom, J. Chung, L. Hammond, C. Kozyrakis, and K. Olukotun, "TAPE: a transactional application profiling environment," in *ICS*, 2005, pp. 199–208.

[14] F. Zyulkyarov, S. Stipic, T. Harris, O. S. Unsal, A. Cristal, I. Hur, and M. Valero, "Discovering and understanding performance bottlenecks in transactional applications," in *PACT'10*.

[15] "Paraver website," http://www.bsc.es/paraver.

[16] C. C. Minh, J. W. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *IISWC*, 2008.

[17] J. Davis, C. Thacker, and C. Chang, "BEE3: Revitalizing computer architecture research," *Microsoft Research*, 2009.

[18] D. Christie *et al.*, "Evaluation of AMD's advanced synchronization facility in a complete transactional memory stack," in *EuroSys'10*, pp. 27–40.

[19] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *PPoPP '08*, pp. 237–246.

[20] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun, "EigenBench: A simple exploration tool for orthogonal TM characteristics," in *IISWC'10*, 2010.

[21] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood, "Performance pathologies in hardware transactional memory," *ACM SIGARCH CA. News*, vol. 35, pp. 81–91, June 2007.

[22] M. Ansari, K. Jarvis, C. Kotselidis, M. Lujan, C. Kirkham, and I. Watson, "Profiling transactional memory applications," in *Euromicro'09*, pp. 11–20.

[23] C. Perfumo *et al.*, "The limits of software transactional memory (STM): dissecting Haskell STM applications on a many-core environment," in *Computing Frontiers '08*.