

# Lightweight Hardware Transactional Memory Profiling

Qingsen Wang, Pengfei Su College of William & Mary {qwang06,psu}@email.wm.edu Milind Chabbi Scalable Machines Research milind@ScalableMachines.org Xu Liu College of William & Mary xl10@cs.wm.edu

# Abstract

Programs that use hardware transactional memory (HTM) demand sophisticated performance analysis tools when they suffer from performance losses. We have developed TxSAM-PLER-a lightweight profiler for programs that use HTM. TxSAMPLER measures performance via sampling and provides a structured performance analysis to guide intuitive optimization with a novel decision-tree model. TxSAMPLER computes metrics that drive the investigation process in a systematic way. It not only pinpoints hot transactions with time quantification of transactional and fallback paths, but also identifies causes of transaction aborts such as data contention, capacity overflow, false sharing, and problematic instructions. TxSAMPLER associates metrics with full call paths that are even deeply embedded inside transactions and maps them to the program's source code. Our evaluation of more than 30 HTM benchmarks and applications shows that TxSAMPLER incurs ~4% runtime overhead and negligible memory overhead for its insightful analyses. Guided by TxSAMPLER, we are able to optimize several HTM programs and obtain nontrivial speedups.

**CCS Concepts** • General and reference  $\rightarrow$  Performance; Cross-computing tools and techniques; • Software and its engineering  $\rightarrow$  Software performance; Software libraries and repositories;

*Keywords* Hardware Transactional Memory, profiling, optimization, HTM benchmark suite

# 1 Introduction

Transactional memory (TM) [25] is a well-known technique for achieving optimistic concurrency in shared-memory parallel programs. TM provides simple interfaces to specify

PPoPP '19, February 16–20, 2019, Washington, DC, USA
 2019 Association for Computing Machinery.
 ACM ISBN 978-1-4503-6225-2/19/02.

https://doi.org/10.1145/3293883.3295728

critical sections; TM optimistically and speculatively executes critical sections, and guarantees correctness by rolling back transactions in the event of conflicts. The Draft C++ TM Specification [5] and its implementation in recent GCC compiler facilitate the usage of TM in parallel programs. The underlying TM runtime system can be implemented in either software (STM) or hardware (HTM). HTM exploits the processor's cache coherence protocol to detect access conflicts. Early HTM implementations appeared in Azul [12] and Rock [15] processors. Intel Haswell and its successors, and IBM POWER8, Blue Gene/Q and zEnterprise EC12 [26] processors now support HTM. HTM is also proposed in emerging GPGPUs [18]. In this paper, we target the programs running with Intel HTM implementation, known as transactional synchronization extensions (TSX) [31].

Transactions support speculative execution: if two transactions have no conflicting memory accesses [33], they can execute concurrently; otherwise, one is aborted. Besides conflict aborts, the memory footprint in a transaction larger than a predetermined cache capacity (e.g., L1 cache) can also cause TSX transaction aborts. TSX treats both the conflict and capacity aborts as *asynchronous* aborts. Unfriendly instructions and events, such as system calls and page faults also abort TSX transactions, known as *synchronous* aborts.

As HTM gains wider adoption, it demands a comprehensive set of tools to measure and analyze how applications behave on a given HTM architecture. Application developers need easy-to-use tools to understand and tune their applications and algorithms for target architectures.

For any measurement-based analysis, two techniques are commonly used: instrumentation and sampling. Instrumentation-based tools add extra code at either source code or binary level to collect measurements; the overhead is proportional to the frequency of instrumentation invocation. Instrumentation-based tools enjoy the benefits of microscopic observation but incur high runtime overhead (several times slowdown). Sampling-based tools sample events (e.g., CPU cycles), and the overhead is proportional to the sampling frequency or the rate of the occurrence of the sampled event. Sampling-based tools introduce low runtime overhead but often fail to offer microscopic details.

Tools to diagnose HTM performance are still in their infancy. Early efforts, such as TSXProf [39], resort to the instrumentation and trace-replay schemes. TSXProf instruments

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PPoPP '19, February 16-20, 2019, Washington, DC, USA

functions in transactions	transactions inside a loon
Listing 1 Example of	Listing 2 Example of small
7 TM_END(); }	7 }
6 A(); B();	<pre>6 TM_END();</pre>
5 TM_BEGIN();	5 a[i]++;
<pre>4 // parallel section</pre>	<pre>4 // low contention</pre>
3 int main(void) {	<pre>3 TM_BEGIN();</pre>
2 void B(){ C(); }	2 for(int i=0; i <n; i++){<="" td=""></n;>
1 void A(){ C(); }	1// parallel loop

transactional regions, and logs them with timestamp counters. Later, in a postmortem pass, it replays transactions in an STM system. Instrumenting HTM requires intercepting each transaction's start and end. Logging each transaction instance, especially the small ones embedded in deeply nested loop incurs high overheads [32]. Moreover, instrumenting transactional instructions perturbs the transaction itself. For example, the instrumented code may increase memory footprint, leading to capacity aborts. Similarly, recording transactional details in a shared memory location can cause conflict aborts. Record-and-replay incurs non-trivial overhead ( $\sim$ 3×) in the replay stage to obtain the full contexts of transactions. Additionally, STM replay is an approximation of HTM, not an authentic reflection of execution on native hardware.

To reduce the overhead of software instrumentation, Intel processors introduce processor tracing (PT) hardware [24] that gleans a complete control flow of a program execution, including the ones in transactions, and outputs the trace in compressed binary packets. Such control flow trace provides insights inside transactions. PT incurs moderate runtime overhead, ~ 20% in our experiments for collecting the trace. However, decoding the trace packets is costly, impeding PT from online analysis [1]. Moreover, our experiments show that PT produces 30-80MB packets per thread per second, which incurs high space overhead.

To avoid fine-grained monitoring via instrumentation or Intel PT, tools such as Perf [35] and VTune [27] leverage event-based sampling of transactions via performance monitoring units (PMU). PMU sampling incurs low overhead but is fraught with problems in measuring HTM execution. First, PMU events trigger processor interrupts when performance counters overflow; interrupts abort transactions. In such a situation, these tools cannot distinguish whether a sample is triggered in the transaction or in its fallback code. Second, when a transaction aborts, it immediately rolls back to the beginning of the transaction, making these tools impossible to get the calling context of the instruction that triggers the abort, if it appears in a deep call chain inside the transaction. For example, if an abort happens inside function C in Listing 1, these tools cannot identify whether it is called from function A or B. Third, as they do not provide time decomposition of a transaction, users may miss optimization opportunities due to large lock waiting time or large overhead time. As shown in Listing 2, if the transaction shows a high commit rate, one may miss the optimization opportunity of coalescing

small transactions to reduce the overhead of creating and destroying transactions. Finally, as Intel TSX monitors conflicts at the cache line level, it is important to avoid contention to reduce unnecessary conflict aborts. Without a detailed memory address tracing, it is challenging for these tools to identify contention due to true or false sharing.

In this paper, we take up the challenge of sampling-based (and hence lightweight) HTM profiling and develop TxSAM-PLER, a lightweight and multi-scale profiler for hardware transactional memory. TxSAMPLER utilizes event-based sampling [28] techniques while overcoming the limitations of PMU HTM sampling. TxSAMPLER eliminates heavyweight instrumentation of memory loads and stores. Unlike the record-and-replay approach, TxSAMPLER is a *one-pass* profiler. As a full-fledged profiler, TxSAMPLER offers a structured approach to attacking HTM-based code; it provides a decision-tree style analysis of runtime metrics and offers rule-of-thumb suggestions for improving performance.

We make the following five contributions in this paper:

- Develop a sampling-based performance tool, TxSAMPLER, to offer detailed insights of HTM. Its multi-scale analysis apportions execution time to various transactional components and pinpoints causes of HTM aborts.
- Devise a unique solution to the challenges when using PMUs to monitor transactional execution.
- Provide a structured scheme to attack HTM codes with performance problems.
- Develop and study a rich set of HTM benchmark suites, HTMBench, which includes more than 30 programs.
- Demonstrate the effectiveness of TxSAMPLER by optimizing several HTM applications based on the insights offered by TxSAMPLER.

TxSAMPLER works on fully optimized binary executables, without the need of program recompilation or heavyweight binary instrumentation. TxSAMPLER typically incurs ~4% runtime overhead and needs less than 5MB memory per thread to collect its measurement data. Guided by TxSAMPLER, we can optimize several applications and benchmarks with problematic HTM usage and obtain up to 3.78× speedups. TxSAMPLER is available at https://github. com/ScalableMachinesResearch/TxSampler; HTMBench is available at https://github.com/ScalableMachinesResearch/ HTMBench.

The remaining paper is organized as follows. Section 2 introduces the background of Intel TSX. Section 3 overviews the methodology of TxSAMPLER for addressing the profiling challenges. Section 4 and 5 describe TxSAMPLER's time and abort analyses, respectively. Section 6 discusses some implementation details. Section 7 evaluates TxSAMPLER. Section 8 applies TxSAMPLER on several case studies. Section 9 surveys existing work and distinguishes our approach. Section 10 shows some limitations of TxSAMPLER and our future work. Finally, Section 11 presents the conclusions. TxSampler

# 2 Background

In this section, we briefly review the programming model of Intel TSX. Intel TSX provides two mechanisms to utilize HTM: restricted transactional memory (RTM) and hardware lock elision (HLE). RTM focuses on the flexible support for HTM, while HLE aims to make existing lock-based programs compatible with HTM. This paper focusses on RTM, but all the techniques can be applied to HLE with trivial extension.

When starting a hardware transaction, the software adds the memory word associated with the lock to its read set by reading the lock word, and continues the transaction if the lock is not held. This transaction may succeed if 1) the lock is not acquired by another thread and 2) no conflicting memory accesses happen. Upon an abort, the transaction is rolled back and TSX hardware reports a status code indicating the abort reason. The software then determines whether to retry the transaction. If not to retry, the software falls back to the slow path, which involves acquiring the global lock, executing the slow path code, and releasing the lock.

For code maintenance, the transaction (fast) path and the fallback (slow) path usually share the same user code. Executing the slow path needs to acquire the lock for correctness, whereas the fast path waits for the lock to be available before starting the speculative execution. The RTM runtime system is typically coded in a library and linked to application executables. In the remaining paper, we refer to Intel HTM as RTM specifically. Moreover, we call the code executed in the fast HTM path as a *transaction*, and both an HTM path and its corresponding fallback path as a *critical section*.

# 3 **TxSAMPLER:** Methodology

Since inexperienced users may be overwhelmed by the abundant information (related to time, transaction states and memory accesses) provided by TxSAMPLER, it employs a decision tree as shown in Figure 1 to pinpoint bottlenecks. The decision tree can help users take the most advantage of TxSAMPLER to find possible optimization opportunities. For instance, assuming the low abort ratio is benign, one may lose the optimization opportunities of transaction overhead in TSX applications Histo (Section 8.3).

TxSAMPLER first quantifies and attributes execution time to code that belongs to critical sections. If the time spent inside critical sections is significant, it will be classified into different components: transactional path, fallback path, lock waiting and overhead for further analysis (elaborated in Section 4). If there are numerous HTM aborts, TxSAMPLER needs to quantify the penalty of aborts and identify potential causes of aborts. According to different abort reasons, the decision tree offers different optimization suggestions. Section 5 details this abort analysis. TxSAMPLER profiles associate its metrics with calling contexts even deep in transactions *including speculatively executed source code regions*. Section 6 provides these implementation details.



Figure 1. The decision tree in TXSAMPLER that offers optimization guidance. The red dotted lines and the numbers show an example usage in our case study, as described in Section 8.1.

In this section, we highlight the challenges that TxSAM-PLER addresses in leveraging PMU sampling to profile parallel programs with HTM.

## 3.1 Challenge I: Handle Aborts due to Interrupts

If a PMU sample occurs when the code is executing a transaction, it results in aborting the transaction and the CPU state looks as if it was about to execute the first instruction of the fallback path. A PMU sample will be seen in the fallback path on two occasions:

- 1. The application caused an HTM abort and the PMU sample happened in the fallback path.
- 2. The PMU counter overflowed in the speculative path, which resulted in the transactional abort, and hence the execution was directed into the fallback path.

Distinguishing these two cases is crucial. If the PMU triggers the transaction abort when sampling a precise event [29] (e.g., CPU cycles, memory loads/stores), PMU records the precise instruction pointer (IP) at the sample point. However, developers often share the same code in both transaction and fallback paths. Thus, only knowing the IP does not distinguish whether the sample happened inside the transaction path or the fallback path.

Solution: We exploit the Last Branch Records (LBR) [29] feature available in modern Intel CPUs to assist in determining whether a sample triggers an abort. The LBR is a circular buffer that contains the most recent N (N is 16 for Haswell/Broadwell, and 32 for Skylake and successors) history branches (e.g., function calls, conditional and unconditional jumps, etc.) of the processor. Each entry contains (1) a  $\langle from, to \rangle$  pair of instruction pointers, (2) an abort bit indicating whether this branch is caused by a transaction abort and (3) another "in-tsx" bit showing whether the branch is in HTM or not. Upon each PMU interrupt, we check the

Execute one Critical Section (CS)		inHTM	inFB	inWait	iпОН
<pre>//prepare for TX If (try_htm) {</pre>	V				V
<pre>wait_until_lock_free();</pre>	V			V	
//execute user code in HTM }	V	V			
else {	V				V
acquireLock();	V			V	
<pre>//execute user code in fallback</pre>	V		V		
<pre>releaseLock();}</pre>	V				V

Figure 2. An example of a critical section with instructions in different states.

most recent LBR entry, which always records the triggering interrupt (as shown in Figure 3(b)). If the abort bit of this branch is set, it indicates this PMU sample occurred while the CPU was executing inside a transaction. Otherwise, this sample was taken in the fallback path and did not abort the transaction.

## 3.2 Challenge II: Attribute to Components

A PMU sample may happen at any time, either in critical sections or not. To perform effective time analysis, we classify PMU samples triggered inside critical sections into four categories: inside a transaction, inside a fallback path, in a lock-waiting part, and in a transaction overhead section. Figure 2 shows an example of a critical section with code in different states.

Our solution is to expose the RTM runtime library states to the profiler. The profiler queries these states in each PMU sample. These states are thread-private, which are elaborated as follows:

inCS: executing in a critical section.

**inHTM:** executing in a transaction path.

inFallback: executing in a fallback path.

- inLockWaiting: waiting for a global lock to be available
   (to start a transaction).
- inOverhead: initiating or cleaning up a transaction.

Because these states are thread-private, this bookkeeping mechanism incurs negligible synchronization overhead to the RTM library. We encode these state flags into different bits of a single word for retrieval to minimize the memory overhead. To extract these state flags, we extend RTM library to provide a state query function, which can be called at any time during the program execution. This query function does not incur extra overhead to the RTM library because it is only called from profilers, not HTM applications. This simple extension to the RTM library adds ~20 lines of code and incurs less than 1% runtime overhead according to our experiments. Given the simple implementation and low overhead, we hope to standardize this extension to HTM runtime libraries for the support of efficient performance measurement analogous to OpenMP tools interface specifications [16].



Figure 3. Example of constructing a calling context where a sample occurs at POINT X. Some LBR entries related to RTM library are omitted.

#### 3.3 Challenge III: Analyze Contention

Memory contention analysis is critical to understand the root causes of conflict aborts, and it requires a profiler to identify true or false sharing among threads.

Solution: We utilize th PMU to analyze contention with negligible overhead by sampling precise events—memory loads and stores. With these events, PMUs can capture the effective address of the sampled memory access [30].

Upon each sample, we create a *per-byte* shadow memory according to the effective address captured by the PMU to record the information of the sampled access including (1) a read-or-write flag, (2) the ID of the thread initiating the access and (3) the current timestamp obtained from rdtsc instruction [3]. Similarly, we also create a *per-cacheline* shadow memory derived from the effective address of the sample.

We detect memory contention upon each sample, when (1) the thread ID of the current sample differs from the ones retrieved from the *per-cacheline* shadow memory, (2) at least one of these recorded flags is a store and (3) the time difference of the two memory accesses is less than a threshold *P*. After identifying the current sample involved in contention, we further check the *per-byte* shadow memory related to the address accessed by the current sample; if the thread ID of the current sample is different from the one recorded in the per-byte shadow memory, the contention is due to true sharing; otherwise, the contention is due to false sharing. The threshold *P* is set to 100ms based on empirical observations.

#### 3.4 Challenge IV: Attribute to Call Path

The calling context (i.e. call chain) inside a transaction is unavailable since PMU samples abort transactions and the signal handler always sees the program context to be at the beginning of the transaction. We work around this problem by extracting call and return instructions from the LBR records and reconstruct the partial call path inside transactions. We configure LBR to collect only recent function calls and returns at each sample and pair them to extract function call relationship in all LBR entries. The call path in a transaction may only be partially constructed depending on the number of function invocations inside the transaction and the size of the LBR stack. Additionally, we unwind the call stack starting from the signal context to determine the call path leading to the beginning of the transaction. We concatenate these two call paths to provide more contextual insights into transactions.

Figure 3 illustrates the idea. A sample occurs at POINT X in the transaction path. SInce that the call stack unwinding can only construct the calling context main $\rightarrow$ A, users cannot know whether D is called from B or C. Then we examine the LBR entries that come with the event sample (shown in Figure 3(b)). The abort and in-tsx columns are two bits that indicate whether the branch is an abort and whether it is in transaction respectively. The first LBR entry (LBR[0]) is caused by the interrupt and its abort bit is used in determining the correct path as discussed in Section 3.1. Inspecting LBR[1] and LBR[2], allows us to construct the path  $C \rightarrow D$ . If the address of the call C instruction equals to the from address of LBR Entry 2,  $C \rightarrow D$  must originate from A. Then we are able to reconstruct the calling context by concatenating the two call paths as shown in Figure 3(c). If the LBR entries overflow within a transaction, the concatenation may miss some call path prefix inside the transaction.

# 4 **TxSAMPLER's** Time Analysis

TxSAMPLER's time analysis, the first stage in the decision tree model, provides intuitive analysis for users who do not need any prior knowledge about the code under optimization. It avoids using HTM-related metrics but derives timing metrics that directly quantify the execution performance. TxSAM-PLER answers the following questions in its time analysis.

- 1. Compared to the entire program, what is the proportion of CPU cycles consumed in "hot" critical sections?
- 2. How many cycles are consumed in the transaction, fallback, or other components in a given critical section?
- 3. How to optimize critical sections protected by HTM to reduce unnecessary CPU cycle consumption?

TxSAMPLER first measures the whole program execution time, known as work W, in CPU cycles. It then decomposes W into cycles consumed in critical sections T and outside critical sections S as shown in Equation 1. Only when T is large enough (e.g., T/W > 20%) does TxSAMPLER recommend further analysis on the execution of critical sections. TxSAM-PLER further decomposes T to different critical sections and identifies the hot ones for optimization. This approach answers Question 1.

$$W = T + S \tag{1}$$

TxSAMPLER then investigates hot critical sections that have a high impact on the entire program performance. TxSAMPLER continues to decompose T into different states, as shown in Equation 2.  $T_{tx}$  is the cycles spent in transaction paths;  $T_{fb}$  is the cycles in fallback paths;  $T_{wait}$  is the cycles in waiting for the global lock; and  $T_{oh}$  is the cycles in other transaction-related code, such as transaction creation and cleanup, also known as transaction overhead. This approach answers Question 2.

$$T = T_{tx} + T_{fb} + T_{wait} + T_{oh}$$
<sup>(2)</sup>

With these timing metrics, TxSAMPLER can provide appropriate optimization guidance in the decision tree model to answer Question 3. If T is small, there is no need to optimize transactions since the optimization efforts lead to little performance gain. If T is large, TxSAMPLER can guide optimizations according to the four metrics in Equation 2.

- *T<sub>tx</sub> is large* ⇒ the transactional path accounts for most execution time. TxSAMPLER usually does not make optimization recommendation.
- $T_{fb}$  is large  $\implies$  most execution time is in the slow fallback path due to frequent aborts or long fallback path. TxSAMPLER recommends applying abort analysis, as described in Section 5, to further understand aborts before optimizing transaction code.
- *T<sub>wait</sub>* is large ⇒ waiting for the lock consumes most of the time. The lock is used to serialize transactions that need to execute in the slow fallback path. TxSAMPLER recommends relaxing the serialization algorithm to reduce the lock waiting time, or further applying abort analysis to explore the possibility of reducing abort rate.
- $T_{oh}$  is large  $\implies$  most time is consumed in initiating transactions, setting up retry mechanism or cleaning up transactions. TxSAMPLER suggests merging multiple small transactions into a larger one to reduce this overhead.

To compute these metrics, TxSAMPLER leverages the techniques described in Section 3.1 and 3.2 to co-design the HTM runtime library and the profiler. TxSAMPLER accumulates CPU cycle samples to the appropriate metrics, as shown in Figure 4. Upon each sample, TxSAMPLER first accumulates the sample to *W*. It then queries the state from RTM library to see whether the sample falls in a critical section. If the sample falls in a critical section, TxSAMPLER increments metric *T* associated with the call path (context) where the sample was taken; otherwise, TxSAMPLER stops analyzing this sample. Further attribution inside the critical section is similar but based on the queried state and LBR.

# 5 **TxSAMPLER's Abort Analysis**

As shown in the decision tree, abort-related metrics are useful when TxSAMPLER's time analysis shows a large amount of time spent in fallback paths or lock waiting. TxSAMPLER also collects abort information from PMUs and computes penalty and contention metrics for individual transactions.

**Penalty metrics** The weight captured by PMU quantifies the penalty of a transaction abort. Usually, a larger weight means a higher abort cost. TxSAMPLER computes an average weight per abort  $\hat{w}_t$  for each sampled transaction t using Equation 3, where  $\sum w_t$  represents the aggregate abort

1: /* Let ctxt be the calling context where the PMU sample occurred */
2: /* Always accumulate work */
3: ctxt.W++;
4: /* Query the state from the RTM library */
5: int state := GetState();
6: if IsSampleInCS(state) then
7: $ctxt.T++;$
8: /* Query the abort bit $B_{abort}$ of the latest entry of LBRs */
9: <b>if</b> $B_{abort} == 1$ then
10: $ctxt.T_{tx}$ ++;
11: else if IsSampleInFallback(state) then
12: $ctxt.T_{fb}$ ++;
13: else if IsSampleInLockWaiting(state) then
14: $ctxt.T_{wait}$ ++;
15: else
16: $ctxt.T_{oh}$ ++;
17: end if
18: end if

Figure 4. Computing timing metrics upon each sample.

weight of all the sampled transaction aborts of *t* and  $\sum s_t$  is the sum of all the sampled aborts of *t*.

$$\hat{w}_t = \frac{\sum w_t}{\sum s_t} \tag{3}$$

For a transaction with large  $\hat{w}$ , we further decompose its abort weight according to different abort reasons. TxSAM-PLER computes the ratio of conflict abort weight  $r_{conflict}$  in Equation 4, where  $\sum w_{conflict}$  denotes the aggregate weight due to conflict aborts of a transaction. Analogously, TxSAM-PLER also computes  $r_{capacity}$  and  $r_{sunchronous}$ .

$$r_{conflict} = \frac{\sum w_{conflict}}{\sum w} \times 100\%$$
 (4)

The penalty metrics help understand the root causes and the impacts of transaction aborts, which can guide future optimization. For example, if  $\hat{w_t}$  is large, one may reduce the transaction size to avoid high abort penalty. If  $r_{conflict}$ is high, we can investigate the transactional instructions pointed by TxSAMPLER to avoid unnecessary conflicts, such as false sharing. If  $r_{synchronous}$  is high, we may move the problematic instructions (e.g., system calls) identified by TxSAMPLER out of the transaction or enable prefetching when caused by a page fault from a memory access.

**Contention metrics** Aggregate metrics alone are not enough to understand the contention across threads. For instance, a thread may always abort other threads, causing thread starvation. Therefore, TXSAMPLER records both perthread transaction aborts and commits, and plots them in a histogram across threads. If there exists an imbalanced distribution of transaction commits or aborts, TXSAMPLER reports this problematic transaction for investigation. A possible optimization is to redistribute the work across threads to balance the transaction execution.

In addition, TxSAMPLER leverages the lightweight analysis described in Section 3.3 to identify the memory instructions in transactions that involve true and false sharing.

# **6 TxSAMPLER** Implementation

TXSAMPLER is implemented atop HPCToolkit [4], a state-ofthe-art call path profiler for parallel programs. TXSAMPLER accepts a compiler-independent, fully optimized executable binary, which is linked with our customized HTM runtime library. TXSAMPLER's data collector monitors the binary execution and produces profiles during a single execution. Then TXSAMPLER's data analyzer post-processes all the profiles: merging the profiles across different threads, associating performance data with source code, and computing derived metrics. Finally, the analyzer records all the insights into files and passes them to TXSAMPLER'S GUI for visualization. In the following, we discuss each TXSAMPLER's component, especially its design for low overhead and high scalability.

**HTM Runtime Library** We adopt the HTM runtime library from an existing approach [40]. As mentioned in Section 3.2, we add 21 lines of code to support TxSAMPLER; 9 of them are used for the state query function, which are executed only when queried by a profiler. To use this library, users only need to enclose a critical section with TM\_BEGIN and TM\_END.

Online Data Collector TxSAMPLER's collector uses LD\_PRELOAD to install the necessary code in the address space of the binary being monitored. Such code utilizes PAPI-5.5.0 [10] to configure PMUs and sample CPU cycles<sup>1</sup>, transaction aborts/commits<sup>2</sup> and memory accesses<sup>3</sup> before the binary executes. TxSAMPLER predefines default sampling periods for these events:  $1 \times 10^7$  for CPU cycles,  $1 \times 10^4$  for RTM aborts/commits, and  $1 \times 10^4$  for memory loads/stores. One can adjust the sampling periods to tune the sampling rates. Empirically, an appropriate sampling rate that balances both overhead and accuracy is 50-200 samples per second per thread. TxSAMPLER also installs an interrupt signal handler to capture samples. Upon each sample, TxSAMPLER determines its call path as described in Section 3.4 and computes runtime metrics. Finally, TxSAMPLER outputs the profiles into files for further analysis.

**Offline Data Analyzer** TxSAMPLER's offline analyzer aggregates all the profiles from different threads and associates analysis data with source code. It merges metrics associated with transactions under the same call path. The profile coalescing overhead grows linearly with the number of threads used by the monitored program. TxSAMPLER leverages the reduction tree technique [47] applied in HPCToolkit to parallelize the merging process. TxSAMPLER requires less than ten seconds to produce the aggregate profiles for all of our studied programs. Besides aggregate metrics, TxSAMPLER also maintains per-thread metrics.

<sup>&</sup>lt;sup>1</sup>Event name: cycles.

<sup>&</sup>lt;sup>2</sup>Event name: RTM\_RETIRED: ABORTED/COMMIT.

<sup>&</sup>lt;sup>3</sup>Event name: MEM\_UOPS\_RETIRED: ALL\_STORES/ALL\_LOADS.

*GUI* TxSAMPLER'S GUI provides a calling context view to show the full call paths for all sampled transactions in the aggregate profile. From this view, one can quickly identify problematic transactions. Moreover, the GUI supports plotting per-thread metrics on any given context across all threads, making it intuitive to recognize the imbalance of transaction commits/aborts across threads.

# 7 Evaluation

We evaluate TxSAMPLER on a 14-core Intel Xeon E7-4830 v4 (Broadwell) processor clocked at 2.0 GHz. The memory hierarchy includes a private 64KB L1 cache, a private 256KB L2 cache, a shared 35MB L3 cache, and 256GB memory. We adapt a variety of multi-threaded programs with hardware transactional memory to evaluate TxSAMPLER. These programs include (1) TM benchmark suites, such as highly optimized STAMP [40], Lee-TM [6], QuakeTM [19], and RMS-TM [34], (2) popular multi-threaded benchmark suites, such as PAR-SEC [9], Parboil [46], NPB [8], CORAL [37], SPLASH2 [45], Synchrobench [23] and SSCA2.2 [7], and (3) multi-threaded applications, such as LeveIDB [11, 13], BPlusTree [17], Kyoto-Cabinet [36], BART [49], Berkeley DB [41], Memcached [2] and PBZip2 [21].

For TM benchmarks, we replace the original software transactions with Intel TSX transactions, while we elide pthread locks or atomics and introduce TSX transactions accordingly for other multithreaded programs. For programs using pthread conditional variables, we apply the techniques described in the literature [50] to avoid using pthread primitives: remove pthread conditional variables (e.g., pthread\_cond\_broadcast and pthread\_cond\_signal) and have threads spin wait for special conditions. For all of the code bases, we do not enlarge or shrink their originally defined critical sections. We only perform the lock substitution by applying HTM instructions around the critical sections. All the programs are compiled with GCC 4.8.5 -03 and run with all 14 cores. Each transaction is configured to retry five times before falling to the slow fallback path; we do not retry transactions with persistent aborts (e.g., unfriendly system calls). We use the suggested native inputs for all benchmarks. For those applications whose execution time is too short (i.e., less than 30s), we tune the inputs to make them run long enough in the native execution.

In the rest of this section, we first evaluate the overhead of TxSAMPLER, then verify the correctness of TxSAMPLER, and finally characterize all the programs and summarize the optimization guided by TxSAMPLER.

## 7.1 TxSAMPLER's Overhead.

Figure 5 reports the runtime overhead of TxSAMPLER. We tune the sampling rate to guarantee that TxSAMPLER collects more than 50 samples per thread per second. The overhead

Input #	ScatterMode	Expected Characteristics
1	Adjacent	Rare conflicts, cache prefetch friendly
2	FirstParts	High conflicts, cache prefetch friendly
3	Random	Rare conflicts, cache prefetch unfriendly

Table 1. Inputs for CLOMP-TM.

is averaged over five of seven executions by excluding the smallest and largest ones; the error bars show the variation. TxSampler incurs  $\sim$ 4% runtime overhead on average.

TxSAMPLER requires no specific adaptation to profile shortrunning programs. However, the proportion of overhead to the execution time can increase because of the fixed overhead of preloading the profiling library and setting up PMUs becomes nontrivial. For example, water and ocean from SPLASH originally run less than 0.1s; TxSAMPLER incurs 15× runtime overhead on average. It is worth noting that the short-running programs often receive little interest in performance tuning, so our investigated HTM programs have reasonably long execution time. Furthermore, TxSAMPLER introduces less than 5MB memory overhead per thread on average independent of the execution time.

We further use STAMP to evaluate the sensitivity of the runtime overhead to the number of threads in Figure 6. The overhead value is averaged across all STAMP benchmarks. We can see that TXSAMPLER maintains low overhead regardless of thread counts.

# 7.2 **TxSAMPLER's** Correctness

We validate the accuracy of TxSAMPLER with a set of microbenchmarks. These microbenchmarks trigger low, moderate, and high transaction abort ratios due to various abort reasons, such as true sharing, false sharing, and special instructions. We obtain the ground truth from the instrumentation in the HTM runtime library. TxSAMPLER generates the profiles that exactly match the ground truth.

Moreover, it is critical for TxSAMPLER to correctly reason about HTM performance behaviors. For this purpose, we perform a set of controlled experiments on CLOMP-TM [43], for which we know what to expect in our profiles. CLOMP-TM provides two configurations (small or large transactions), together with three inputs to trigger different memory access patterns and various degrees of memory conflicts. Table 1 shows the three inputs with different characteristics, each of which runs with two configurations: small transactions and large transactions. With this benchmark, one can (1) compare the behaviors of the same configuration but different inputs, or (2) compare the performance of different configurations with the same input. Figure 7 shows the performance data collected by TxSAMPLER for CLOMP-TM running with 14 threads, which can correctly explain the performance behavior of CLOMP-TM.





Figure 6. The average runtime overhead with variance bars according to different thread counts across all STAMP benchmarks.



Figure 7. CLOMP-TM data from TxSAMPLER.

**Different behaviors across inputs.** The top of Figure 7 is the CPU cycles decomposition. For small transactions, regardless of the inputs, TxSAMPLER reports high HTM overhead  $T_{oh}$ , which matches our intuition that small transactions have relatively higher overhead. For large transactions, we can see that with low conflicts and small footprint (input 1), most of the execution time is in transactions  $T_{tx}$  and there are nearly no aborts. For high conflicts (input 2), most of the time is spent in lock waiting  $T_{wait}$ ; also it has a large number of conflict aborts and the weight associated with the abort computed by TxSAMPLER is high. For input 3 with large memory footprint, TxSAMPLER reports a larger portion of capacity aborts compared to high conflicts.

**Different performance between configurations.** We obtain two insights from studying CLOMP-TM: (1) With low conflicts, large transactions perform better, and (2) with high conflicts, small transactions perform better. TXSAMPLER provides the data to explain these observations. First, with low conflicts, both small and large transactions rarely fall into the slow paths, so they both have small  $T_{wait}$  and  $T_{fb}$ . Thus, the overhead  $T_{oh}$  has a high impact on the performance. As small transactions have larger  $T_{oh}$ , they perform worse than

Code	Symptoms Solutions		Speedup	
*dedup [0]	high capacity abort	refine hash table	1.20×	
ueuup [9]	high synchronous aborts	remove system calls	1.20×	
AVL Tree [14]	high T <sub>wait</sub>	elide read lock	1.21×	
*histo [46]	high T <sub>oh</sub>	merge transactions	2.05	
Insto [40]	severe false sharing	sort the input array	2.75	
UA [8]	high T <sub>oh</sub>	merge transactions		
vacation [40]	high abort rate	reduce transaction size	2 1.21×	
*LevelDB [11, 13]	high abort rate	split transactions	1.05×	
*SSCA2 [7]	high T <sub>oh</sub>	merge transactions	1.10×	
*netdedun [46]	high conflict abort	shrink transactions	2.6×	
netueuup [40]	high synchronous aborts	remove system calls		
*1: 1 11: . [00]	high abort rate	te limit transaction size		
linkedist [25]	low average abort penalty	with auxiliary locks	3./8X	

Table 2. Optimization overview (\* newly found).

large ones. Second, with high conflicts, small transactions have better performance than large ones. By causing a large number of aborts, large transactions have more execution time in  $T_{wait}$  and  $T_{fb}$  since large transactions incur higher abort penalty and serializing large transactions is more costly than small ones.

#### 7.3 HTM Program Characterization.

Figure 8 categorizes all the benchmarks we have studied based on two metrics—the critical section duration ratio  $r_{cs}$ and the abort/commit ratio  $r_{a/c}$ . The critical section duration ratio  $r_{cs}$  computed as T/W, quantifies the significance of critical sections in each program. Empirically, if  $r_{cs} < 0.2$ , the program does not benefit much from optimizing transactions; we categorize these programs in Type I. Otherwise, we further classify programs into two types according to  $r_{a/c}$ . If  $r_{a/c} < 1$ , the program belongs to *Type II* since it usually suggests overall low transaction conflicts according to our experience. However, there are still optimization opportunities such as reducing  $T_{oh}$  and increasing the commit rate of specific transactions. The remaining programs, falling into *Type III*, are usually worth optimizating to alleviate conflicts inside transactions. Table 2 overviews our optimization for some programs in Type II and III.

# 8 Case Study

In this section, we study several programs with HTM-related performance bottlenecks, which are newly found by TxSAM-PLER, and demonstrate how TxSAMPLER guides intuitive optimization with its decision tree model. More case studies are in the supplementary material, which is available at https://github.com/ScalableMachinesResearch/TxSampler

#### TXSAMPLER

PARSEC3/netstreamcluster, QuakeTM, PBZip2, BART/nufft, stamp/ssca, RMS-TM/ScalParc, splash2/raytrace splash2/barnes, splash2/barnes, splash2/focean, splash2/ocean, splash2/water	BerkeleyDB, Memcached, Parboil/histo, RMS-TM/UtilityMine, SSCA2, NPB/ua, PARSEC2/dedup, PARSEC3/netdedup, PARSEC3/netferret		abort commit < 1
	LevelDB, AVLTree, syncrho/linkedlist, synchro/skiplist, bplustree, Lee-TM, KyotoCabinet, stamp/yada, stamp/intruder, stamp/vacation, stamp/kmeans, stamp/genome, stamp/labyrinth	Ш	$\frac{\text{abort}}{\text{commit}} > 1$
CS < 20%	CS > 20%		

Figure 8. Application categorization.

## 8.1 PARSEC2 Dedup

Dedup, a PARSEC benchmark, compresses data via deduplication. It employs pipeline parallelism, which consists of three major stages: ChunkProcess, FindAllAnchors, and Compress. By default, Dedup assigns four threads to each of the three stages. We iteratively optimize this benchmark based on the guidance from TxSAMPLER. The red dotted arrows and the numbers in Figure 1 show how TxSAMPLER traverses the decision tree to reach the optimization guidance step by step.

As shown in Figure 1, TxSAMPLER with its time analysis (1) reports significant (38% of the total execution) time inside critical sections and (2) identifies that  $T_{wait}$  accounts for 78% of the execution time in critical sections, making it worth of abort analysis (3). We sort abort weight and repeat (3) to narrow down our focus on a function hashtable\_search inside the transaction as shown in Figure 9, a screenshot of TxSAMPLER's GUI.

The GUI consists of three panes: the top one displays the source code; the bottom left one presents the program structures; the bottom right one shows the metrics associated with program contexts. Specifically, if a sample is inside a transaction, its in-transaction call path will be attributed under a node labelled begin\_in\_tx. With the abort analysis (4), Figure 9 reveals that hashtable\_search, inside a critical section, within its call path accounts for 7.8% of the total abort weight. Furthermore, there are 9.8% capacity aborts attributed to this line. TxSAMPLER recognizes that the large memory footprint (5) of the transaction causes high capacity abort rate, which is the root cause of the bottleneck.

With source code study, we find that the problematic code, searching for an item in a linked list associated with a hash table entry, resides in the conflict resolution path of the hash table. As the linked list is a cache-unfriendly data structure, the memory footprint can become large due to the randomaccess pattern, which leads to excessive capacity aborts. Furthermore, the frequent update and search on the linked list also incur a lot of conflict aborts. These findings lead us to investigate the implementation of the hash function.

We notice that only 2.2% of hash table slots have been occupied, and the occupied slots have a long linked list of keys

🖻 hashtable.c 🛛 🖻 encoder.c 🕅		- 8	
229 pthread_mutex_t *ht_lock = hasht	able_getlock(c	ache, (void *)k ^	
230 #endif			
<pre>231 pthread_mutex_lock(ht_lock);</pre>			
232 #endif			
<pre>233 if ((entry = hashtable_search(ca</pre>	che, (void *)k	ey)) == NULL) {	
234 // cache miss: put it in the h	aghtable and t	he queue for th	
235 struct pContent * value; v			
<		>	
💊 Calling Context View 🛛 🗞 Callers View 👬 Flat V	jew		
🗈 🗣   🌜 😥   🕅 📰 🗛 👘 🖬 👻 🏒			
Scope	Abort weight	Capacity abort	
✓ <thread root=""></thread>	1.72e+09 67.2%	9.99e+04 72.9%	
✓	1.17e+09 45.9%	6.48e+04 47.3%	
<ul> <li>loop at encoder.c: 332</li> </ul>	1.17e+09 45.9%	6.48e+04 47.3%	
✓ ➡ 347: sub_ChunkProcess	1.17e+09 45.9%	6.48e+04 47.3%	
✓ ➡ 231: tm_begin	1.17e+09 45.9%	6.48e+04 47.3%	
🔽 🔛 175: begin in ty Pseudo nod	e 1 17e+09 45 98	6.48e+04 47.3%	
	- 1.1/0/05 40.56		

Figure 9. TxSAMPLER's report for Dedup, showing a problematic critical section.

since many entries are mapped to the same slots. Our optimization is to improve the hash function to balance the distribution of hash keys. In the function hash() in hashtable.c, we delete all the bit shifting operations involved in computing the hash key and bitwise XORed the key with lower 32 bits of itself. This change improves the utilization of the hash table to 82% and reduces capacity aborts by 97%.

Re-applying abort analysis (3) and (4), TXSAMPLER identifies another problematic transaction in write\_file function, where 78% aborts are synchronous aborts due to system calls (6). As write\_file is only executed by the master thread, we move system calls outside of the critical section, which reduces the synchronous aborts by 26%.

In summary, after applying all these optimizations guided by TxSAMPLER, we speed up the entire program by 1.20×.

#### 8.2 LevelDB

LevelDB [13] is an embedded key-value database based on Google's BigTable structure, and we examine its HTM implementation [11]. We use the benchmarking tool db\_bench released along with LevelDB to evaluate its performance. We keep the default configuration except setting --num parameter as 200, 000 and launch it with 14 threads.

TxSAMPLER's time analysis shows that LevelDB consumes 47% of the total execution time in critical sections. Reported by the abort analysis, abort/commit ratio is as high as 2.8 and the aborts are mostly due to conflicts. Around 97% of the aborts occur in the method db\_->Get() called from ReadRandom(). After examining the Get() method, we find that two transactions at this function start and end are problematic. In the first transaction, the reference counts of three shared objects are incremented with their own Ref methods, while decremented later in the second transaction with their own Unref methods. The Unref method also deletes the object itself when the reference count reaches 0. As db\_->Get() is called extensively by all the threads, these shared-reference counts incur high transactional aborts.

#### Listing 3. Original HTM implementation in Histo.

```
1 for (i = 0; i<img_width*img_height/txn_gran+1; ++i) {</pre>
    TM_BEGIN();
2
   for(j=0; j<txn_gran && (i*txn_gran+j)<img_width*</pre>
3
          img_height; j++){
      const unsigned int value = img[i*txn_gran+j];
if (histo[value] < UINT8_MAX)</pre>
4
5
6
         ++histo[value]:
7
    3
    TM_END();
8
93
```

## Listing 4. Optimized HTM implementation in Histo.

TxSAMPLER suggests to reduce transaction sizes to avoid high abort penalty. We split the two transactions and thus obtain smaller transactions that only include shared reference count updates. The change is safe since these shared counters are only used in db\_->Get() after initiation and counter incrementing and decrementing are placed before and after any other operations respectively to guarantee consistency. This optimization reduces the abort/commit rate from 2.8 to 0.38 yielding a 2.06× speedup in function ReadRandom() and a 1.05× speedup for the entire execution.

#### 8.3 Parboil Histo

Histo, a Parboil benchmark, computes a histogram of a 2Darray with a maximum bin count of 255 [46]. In the main function, it iterates a 2D-array and updates the histo array in a critical section, protected by OpenMP omp critical. We replace the OpenMP implementation with HTM and evaluate it with 14 threads under two different inputs with the same size: Input-1 results in an unevenly distributed output while Input-2 leads to a more evenly distributed result. It is worth noting that HTM-based Histo runs much faster than the original OpenMP-based version.

We use TxSAMPLER to study the performance of HTMbased Histo implementation further. TxSAMPLER reports that the HTM overhead  $T_{oh}$  is high: more than 40% of the total execution time for both inputs. Thus, TxSAMPLER suggests fusing small transactions into a large one to reduce HTM overhead. Listing 3 shows the original code: a small transaction is in a loop nest. We optimize it by coalescing every txn\_gran iterations into one large transaction as shown in Listing 4. In practice, we set txn\_gran to 10,000 for Input-1 and 1,000 for Input-2. For Input-1, this optimization reduces  $T_{oh}$  to 0.3% of the total execution time, yielding a 2.95× speedup.

However, for Input-2, the application is slightly slowed down even though  $T_{oh}$  of the total execution time is reduced to 0.6%. TxSAMPLER further reports that the abort/commit ratio increases dramatically from 0.002 to 5.7, which leads to high abort penalty. TxSAMPLER suggests there are a large

number of false sharings when incrementing the histo array. We sort the input 2D-array first so that each thread could have a more concentrated access footprint based on the OpenMP static scheduling strategy. This sorting operation reduces the false sharing metric by more than 99% and the abort/commit ratio is improved to 3.7, yielding a 2.91× speedup.

It is worth noting that our optimized transaction implementation of Histo outperforms its implementation with atomic operations by 2.32×.

# 9 Related Work

Tools such as PGI's PGPROF [48], OpenSpeedShop [44], Oracle Solaris Studio [42], and HPCToolkit [4] use PMU sampling to monitor threaded codes with low measurement overhead. However, these tools do not handle HTM particularly, resulting in attributing all the PMU samples to fallback paths and no insight into transaction internals. In this section, we only review the profilers specific to TM.

### 9.1 Profilers for STM

There are several tools [22, 38, 51, 52] to identify performance bottlenecks in STM systems. None of these profilers can directly apply to HTM. First, STM profilers can monitor the read and write sets maintained by STM runtime systems, which are not accessible to software in HTM. Furthermore, tracking memory accesses in transactions may incur high overhead. The runtime overhead can be easily larger than  $30\times$ , which may disturb transaction execution. Finally, STM and HTM have different behaviors, so the insights obtained from STM may not apply to HTM.

#### 9.2 Profilers for HTM

HTM runtime systems capture the status code of each transaction instance recorded by the HTM hardware. From the status code, one can know whether the transaction aborts or commits, and if an abort, what the abort reason is. To provide deeper insights, Gaudet developed Transactional Event Profiler (TEP) [20] for IBM Blue Gene/Q (BGQ) TM. It extracts the events from the status code, logs them in a memory buffer, and visualizes the traces. A drawback of this approach is the high memory overhead. Score-P [32] instruments BGQ TM and aggregates all the abort/commit events across all transaction instances and presents them in full calling contexts. However, Score-P may incur high runtime overhead, especially for small transactions in loops.

**TSXProf** To reduce both runtime and memory overhead and provide more profound insights, Liu et al. developed TSXProf [39], based on the record-and-replay technique. It uses lightweight timestamp counters to record HTM execution (the record phase) and replays transactions to obtain more information (calling contexts, abort rate, transaction contention set, etc.) with auxiliary TM libraries. We distinguish TXSAMPLER from TSXProf from the following aspects:

- Overhead. TxSAMPLER only requires one pass execution and incurs ~4% runtime overhead on average, while TSX-Prof needs at least two-pass executions. Although the record phase (first phase) of TSXProf is lightweight, the replay phase can introduce high overhead due to (1) the heavyweight instrumentation to a transaction's read and write sets and (2) the costly calling context determination (> 3× overhead). The memory overhead is less than 5MB per thread on average for TxSAMPLER since it merges the metrics under the same calling context. As TSXProf has to record all the attempted transactions into a trace file in the first phase, the required disk usage is proportional to the number of attempted transactions and abort rate.
- **Time decomposition.** TxSAMPLER can provide the time decomposition view of each transaction as shown in Section 4 but TSXProf does not.

**Perf** Perf [35] can use hardware events to sample programs to monitor transactions with low overhead. It pinpoints hotspots in transactions by sampling CPU cycles and uses transaction-related events to identify aborts and their reasons. We distinguish TXSAMPLER from Perf as follows:

- **Time decomposition.** Unlike TxSAMPLER's time analysis, Perf does not quantify the cycles in different components of an HTM-based critical section.
- **Calling context.** Perf can also utilize LBR to reconstruct the full calling contexts. However, the calling contexts may be incomplete due to the limited number of LBR entries. TxSAMPLER uses call stack unwinding to construct the calling contexts outside transactions and takes advantage of LBR to deduce the calling contexts inside transactions, which can maximally preserve the context information.
- **Comprehensive optimization suggestions.** Perf does not provide detailed optimization guidance as TxSAMPLER does in Figure 1.

**Intel VTune** Intel VTune [27], another state-of-the-art sampling-based tool, provides two analyses (tsx-hotspots and tsx-exploration) for HTM applications. The tsx-hotspots analysis uses the hardware event to identify the performance-critical sections inside transactions. VTune can enable Processor Trace (PT) to obtain additional cycle and instruction information but with higher overhead. The tsx-exploration analysis uses the TSX related events to understand the behavior of transactions such as the number of aborts and wasted cycles due to aborts. We distinguish TXSAMPLER from VTune as follows:

- **Time decomposition.** Similar to Perf, VTune does not quantify the cycles in different components of an HTM-based critical section as well.
- **Calling context.** VTune, utilizing LBR only, does not provide full calling context inside HTM, while TXSAMPLER

uses LBR and call stack unwinding to construct calling contexts for all analyses even inside HTM.

• **Comprehensive optimization suggestions.** VTune does not provide detailed decision-tree based optimization guidance.

In summary, Perf and VTune are state-of-the-art samplingbased profilers. However, they are hard to pinpoint problematic transactions with pathology in time analysis as shown in Histo (Section 8.3), SSCA2, UA, and AVL tree. Since they do not provide contention analysis, they do not distinguish whether the contention is true or false sharing (e.g., the Histo benchmark in Section 8.3). Furthermore, the time component information can guide users to target the transactions worthy of optimization.

# 10 Limitations

Like other sampling-based tools such as Vtune and Perf, TxSAMPLER has the following limitations: (1) The sampling strategy only identifies statistically significant performance issues; (2) the profiling result depends on the input; (3) the sampling rate should be adjusted manually. TxSAMPLER currently works on Intel TSX. However, modifications to adapt to POWERPC machines (e.g. Blue Gene/Q, zEC12 and POWER8) are minimal. The accuracy of abort analysis on a different architecture depends on the PMU counters. For instance, Intel cores only give 6 kinds of abort reasons while POWER8 provides 11.

# 11 Conclusions

TxSAMPLER is a sampling-based lightweight call path profiler for programs using hardware transactional memory. TxSAM-PLER provides insights of time decomposition, PMU metrics of aborts and commits, and memory contention of hardware transactions. We have adapted HTM runtime library to support sampling better. The proposed extensions to the HTM runtime library are lightweight and simple to implement. Standardizing our proposed extensions will enable performance tools to take advantage of lightweight HTM monitoring on multiple architectures. We have demonstrated the strength of TxSAMPLER on a newly developed HTMBench suite to showcase its low overhead, superior scalability, and rich user interfaces. TxSAMPLER's systematic approach to tackling performance issues in HTM codes is effective - with little effort we were able to identify intricate performance losses arising from hardware-software interactions in a suite of complex HTM applications unfamiliar to us and were able to optimized them to yield speedups as high as 3.78×.

# Acknowledgments

We thank anonymous reviewers for their valuable comments.

### Qingsen Wang, Pengfei Su, Milind Chabbi, and Xu Liu

# References

- 2017. linux/intel-pt.txt at master · torvalds/linux. https://github.com/ torvalds/linux/blob/master/tools/perf/Documentation/intel-pt.txt. (Accessed on 04/06/2018).
- [2] 2017. Memcached. https://memcached.org.
- [3] 2017. RDTSC: Read Time-Stamp Counter (x86 Instruction Set Reference). https://c9x.me/x86/html/file\_module\_x86\_id\_278.html. (Accessed on 10/25/2017).
- [4] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. 2010. HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs Http://Hpctoolkit.Org. *Concurr. Comput. : Pract. Exper.* 22, 6 (April 2010), 685–701. https: //doi.org/10.1002/cpe.v22:6
- [5] Ali-Reza Adl-Tabatabai, Tatiana Shpeisman, and Justin Gottschlich. 2012. Draft Specification of Transactional Language Constructs for C++. https://sites.google.com/site/tmforcplusplus. (2012).
- [6] Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. 2008. Lee-TM: A Non-trivial Benchmark for Transactional Memory. In ICA3PP '08: Proceedings of the 7th International Conference on Algorithms and Architectures for Parallel Processing. LNCS, Springer.
- [7] David Bader and Kamesh Madduri. 2005. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. *High Performance Computing–HiPC 2005* (2005), 465–476.
- [8] D. H. Bailey, E. Barszcz, et al. 1991. The NAS parallel benchmarks – summary and preliminary results. In *Proc. of the 1991 ACM/IEEE conference on Supercomputing (Supercomputing '91)*. ACM, New York, NY, USA, 158–165.
- [9] Christian Bienia. 2011. Benchmarking Modern Multiprocessors. Ph.D. Dissertation. Princeton University.
- [10] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. 2000. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications* 14, 3 (Fall 2000), 189–204. citeseer.ist.psu.edu/ browne00portable.html
- [11] Vamsi Chitters, Adam Midvidy, and Jeff Tsui. 2013. Reducing Synchronization Overhead Using Hardware Transactional Memory. Technical Report. Technical report, University of California at Berkeley, Berkeley CA.
- [12] Cliff Click. 2009. Azul's Experiences with Hardware Transactional Memory. In 2009 Transaction Memory Workshop.
- [13] Jeff Dean and Sanjay Ghemawat. 2011. LevelDB: A Fast Persistent Key-Value Store. https://opensource.googleblog.com/2011/07/ leveldb-fast-persistent-key-value-store.html
- [14] Dave Dice, Alex Kogan, and Yossi Lev. 2016. Refined transactional lock elision. In Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, 19.
- [15] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. 2009. Early Experience with a Commercial Hardware Transactional Memory Implementation. In Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV). ACM, New York, NY, USA, 157–168. https: //doi.org/10.1145/1508244.1508263
- [16] Alexandre Eichenberger et al. 2014. OpenMP Technical Report 2 on the OMPT Interface. http://www.openmp.org/wp-content/uploads/ ompt-tr2.pdf.
- [17] embedded2016. 2016. GitHub embedded2016/bplus-tree: B+ tree implementation in C. https://github.com/embedded2016/bplus-tree. (Accessed on 08/18/2017).
- [18] Wilson W. L. Fung, Inderpreet Singh, Andrew Brownsword, and Tor M. Aamodt. 2011. Hardware Transactional Memory for GPU Architectures. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44). ACM, New York, NY, USA, 296–307. https://doi.org/10.1145/2155620.2155655

- [19] Vladimir Gajinov, Ferad Zyulkyarov, Osman S Unsal, Adrian Cristal, Eduard Ayguade, Tim Harris, and Mateo Valero. 2009. QuakeTM: parallelizing a complex sequential application using transactional memory. In Proceedings of the 23rd international conference on Supercomputing. ACM, 126–135.
- [20] Matthew Gaudet. 2014. Serialization Management Driven Performance in Best-Effort Hardware Transactional Memory Systems. Master dissertation. University of Alberta.
- [21] Jeff Gilchrist. 2015. Parallel BZIP2 (PBZIP2). http://compression.ca/ pbzip2/. (Accessed on 08/25/2017).
- [22] Justin E. Gottschlich, Maurice P. Herlihy, Gilles A. Pokam, and Jeremy G. Siek. 2012. Visualizing Transactional Memory. In Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12). ACM, New York, NY, USA, 159–170. https://doi.org/10.1145/2370816.2370842
- [23] Vincent Gramoli. 2015. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. In ACM SIGPLAN Notices, Vol. 50. ACM, 1–10.
- [24] Part Guide. 2016. Intel® 64 and IA-32 Architectures Software Developer's Manual. *Volume 3C: System programming Guide, Part 3* 3C (2016).
- [25] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings* of the 20th Annual International Symposium on Computer Architecture (ISCA '93). ACM, New York, NY, USA, 289–300. https://doi.org/10. 1145/165123.165164
- [26] IBM. 2015. z/Architecture Principles of Operation. http://www-01.ibm.com/support/docview.wss?uid= isg2b9de5f05a9d57819852571c500428f9a. SA22-7832-09 (2015).
- [27] Intel 2013. Intel VTune Amplifier XE 2013. http://software.intel.com/ en-us/intel-vtune-amplifier-xe. Last accessed: Dec. 12, 2013.
- [28] Intel Corp. NA. Hardware Event-based Sampling Collection. https: //software.intel.com/en-us/node/544067.
- [29] Intel Corp. NA. Intel Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide. https://software.intel.com/sites/default/files/m/5/2/c/f/1/ 30320-Nehalem-PMU-Programming-Guide-Core.pdf.
- [30] Intel Corporation. 2010. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2, Number 253669-032.
- [31] Intel Corporation. 2016. Intel 64 and IA-32 Architectures Software Developer's Manual.
- [32] Jie Jiang, Peter Philippen, Michael Knobloch, and Bernd Mohr. 2014. Performance Measurement and Analysis of Transactional Memory and Speculative Execution on IBM Blue Gene/Q. Springer International Publishing, Cham, 26–37.
- [33] Ken Kennedy and John R. Allen. 2002. Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- [34] Gokcen Kestor, Vasileios Karakostas, Osman S Unsal, Adrian Cristal, Ibrahim Hur, and Mateo Valero. 2011. RMS-TM: A comprehensive benchmark suite for transactional memory systems. In ACM SIGSOFT Software Engineering Notes, Vol. 36. ACM, 335–346.
- [35] Andreas Kleen. 2013. Intel Transactional Synchronization Extensions (Intel TSX) profiling with Linux perf. https://goo.gl/BlltQ4. 3 May 2013.
- [36] FAL Labs. 2011. Kyoto Cabinet: a straightforward implementation of DBM. http://fallabs.com/kyotocabinet/. (Accessed on 08/18/2017).
- [37] Lawrence Livermore National Laboratory. 2014. LLNL Coral Benchmarks. https://asc.llnl.gov/CORAL-benchmarks. Last accessed: Dec. 12, 2013.
- [38] Yossi Lev. 2010. Debugging and Profiling of Transactional Programs. Ph.D. dissertation. Brown University.

#### TXSAMPLER

- [39] Yujie Liu, Justin Gottschlich, Gilles Pokam, and Michael Spear. 2015. TSXProf: Profiling Hardware Transactions. In Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15). IEEE Computer Society, Washington, DC, USA, 75–86. https://doi.org/10.1109/PACT.2015.28
- [40] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M Michael, and Hisanobu Tomari. 2015. Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and power8. In Proceedings of the 42nd Annual International Symposium on Computer Architecture. ACM, 144–157.
- [41] Oracle. 2007. Oracle Berkeley DB. http://www.oracle.com/ technetwork/database/database-technologies/berkeleydb/overview/ index.html. (Accessed on 08/18/2017).
- [42] Oracle. 2017. Oracle Solaris Studio. http://www.oracle.com/ technetwork/server-storage/solarisstudio/overview/index.html.
- [43] Martin Schindewolf, Barna Bihari, John Gyllenhaal, Martin Schulz, Amy Wang, and Wolfgang Karl. 2012. What scientific applications can benefit from hardware transactional memory?. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society Press, 90.
- [44] Martin Schulz, Jim Galarowicz, Don Maghrak, William Hachfeld, David Montoya, and Scott Cranford. 2008. OpenSpeedShop: An open source infrastructure for parallel performance analysis. *Sci. Program.* 16, 2-3 (April 2008), 105–121.
- [45] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. 1992. SPLASH: Stanford parallel applications for shared-memory. ACM SIGARCH Computer Architecture News 20, 1 (1992), 5–44.
- [46] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012).
- [47] Nathan R. Tallent, Laksono Adhianto, and John M. Mellor-Crummey. 2010. Scalable Identification of Load Imbalance in Parallel Executions Using Call Path Profiles. In SC 2010 International Conference for High-Performance Computing, Networking, Storage and Analysis. ACM, New York, NY, USA.
- [48] The Portland Group. 2011. PGPROF Profiler Guide Parallel Profiling for Scientists and Engineers. http://www.pgroup.com/doc/pgprofug.pdf.
- [49] Martin Uecker, Frank Ong, Jonathan I Tamir, Dara Bahri, Patrick Virtue, Joseph Y Cheng, Tao Zhang, and Michael Lustig. 2015. Berkeley advanced reconstruction toolbox. In *Proc. Intl. Soc. Mag. Reson. Med*, Vol. 23. 2486.
- [50] Richard M Yoo, Christopher J Hughes, Koonchun Lai, and Ravi Rajwar. 2013. Performance evaluation of Intel® transactional synchronization extensions for high-performance computing. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for.* IEEE, 1–11.
- [51] Ferad Zyulkyarov, Srdjan Stipic, Tim Harris, Osman S. Unsal, Adrián Cristal, Ibrahim Hur, and Mateo Valero. 2010. Discovering and Understanding Performance Bottlenecks in Transactional Applications. In Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10). ACM, New York, NY, USA, 285–294. https://doi.org/10.1145/1854273.1854311
- [52] Ferad Zyulkyarov, Srdjan Stipic, Tim Harris, Osman S. Unsal, Adrián Cristal, Ibrahim Hur, and Mateo Valero. 2012. Profiling and Optimizing Transactional Memory Applications. *International Journal of Parallel Programming* (2012).

# A.1 Abstract

We have prepared a three-part artifact evaluation:

**Overhead of TxSAMPLER:** We reproduce the overhead of execution time as shown in Figure 5 in the paper.

*Speedup after optimization:* We reproduce the speedup of optimized applications as shown in Table 2 in the paper.

**Profile Analysis** We run TxSAMPLER on the case studies discussed in Section 8 and produce profiles databases. One can view the generated profiles with HPCViewer to navigate the performance problems and optimization opportunities as discussed in the paper.

# A.2 Artifact check-list (meta-information)

- Program: TxSAMPLER
- Compilation: gcc -O3
- Data set: RMS-TM, parboil, parsec-2.1, parsec-3.0, stamp-0.9.10, bart, Lee, splash2
- Run-time environment: Linux 3.10.0
- Hardware: Intel Xeon Haswell and its successors with Hardware Transactional Memory supported
- Execution: python/bash script
- Output: Figures, text files and TxSAMPLER Profiles
- How much disk space required (approximately)?: 100GB
- Publicly available?: Yes

## A.3 Description

### A.3.1 How delivered

All the source codes and inputs of TXSAMPLER and HTM programs are delivered via github and Google drive.

## A.3.2 Hardware dependencies

The hardware should be Intel Xeon Haswell and its successors with Transactional Synchronization Extensions (TSX) supported.

# A.3.3 Software dependencies

We need Docker CE and Java SE. We have tested TxSAMPLER on Linux 3.10.0. We noticed a problem in Linux 4.x, which seems to have incompatibility with PAPI 5.5.0 version we use when using four hardware performance counters. We will make an effort to eliminate the dependency on PAPI in the future, which was not possible in the short deadline of the artifact evaluation.

## A.3.4 Data sets

RMS-TM, parboil, parsec-2.1, parsec-3.0, stamp-0.9.10, bart, Lee, splash2.

## A.4 Installation

To avoid the complexity of installing all the dependencies, we have prepared a docker file so that users can build their docker image and perform all the experiments inside the docker. Please refer to https://github.com/jqswang/txsampler-ae for detailed instructions.

## A.5 Experiment workflow

Follow the instructions shown in https://github.com/jqswang/txsampler-ae to build and run the docker image. Once inside the container, download the benchmark suite and input by the following command, and then set the environmental variables.

```
1 $ get_benchmark_and_input.sh
```

```
2 $ cd /data/txsampler_benchmark
```

```
3 $ source set_env
```

Edit /data/txsampler\_benchmark/run.conf to modify the number of threads and the CPU affinity setting instructed in the file comment.

The /data directory inside the container is mapped to ./mydata directory from the host, which can be used to transfer files between the container and the host.

## A.5.1 Overhead of TxSAMPLER

Use following command to generate the figure of runtime overhead similar to Figure 5.

\$ measure\_overhead.py all

The generated figure is under the current directory.

## A.5.2 Speedup after optimization

Issue the following command to measure the speedup of optimized applications as shown in Table 2.

\$ measure\_speedup.py all

The result is shown in the standard output.

## A.5.3 Profile analysis

Issue the command to produce profile databases:

1 \$ generate\_profile.py -1 # list available applications 2 \$ generate\_profile.py [application name]

One can use HPCViewer to open the database directory and check the result. HPCViewer prebuilt on different platforms is available at http://hpctoolkit.org/download/hpcviewer/.

## A.6 Evaluation and expected result

Depending on the hardware and the configuration of thread number and core binding, the result may be different from the one presented in the paper.

#### A.6.1 Overhead of TxSAMPLER

The generated figure is similar to Figure 5. The overhead of a specific application should be between 0  $\sim$  20% and the geo-mean is less than 10%.

#### A.6.2 Speedup after optimization

In the speedup analysis, the speedup number will be shown in the terminal which is like Table 2. The speedup should be nontrivial but the exact speedup number is sensitive to the configuration.

# A.6.3 Profile analysis

One can follow Figure 1 to find optimization opportunities. Section 8.1 provides detailed instructions. The metric pane shows many metrics, from which we can derive new metrics. For example, the capcity abort in Figure 9 is the sum of capacity abort read and capacity abort write. For backup, we have also provided a profile database of dedup (see https://github.com/jqswang/txsamplerae), which can be directly opened by HPCViewer.