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 TxSampler—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 → Performance**; **Cross-computing tools and techniques**; • **Software and its engineering → Software performance**; *Software libraries and repositories*;

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

1 Introduction

Transactional memory (TM) [28] is a well-known technique for achieving optimistic concurrency in shared-memory parallel programs. TM provides simple interfaces to specify 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 [16] processors. Intel Haswell and its successors, and IBM POWER8, Blue Gene/Q and zEnterprise EC12 [29] processors now support HTM. HTM is also proposed in emerging GPGPUs [19]. In this paper, we target the programs running with Intel HTM implementation, known as transactional synchronization extensions (TSX) [34].

Transactions support speculative execution: if two transactions have no conflicting memory accesses [36], 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 [42], resort to the instrumentation and trace-replay schemes. TSXProf instruments

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

```c
void A(){  C(); }
void B(){  C(); }
int main(void) {
    // parallel section
    TM_BEGIN();
    A(); B();
    TM_END(); }
```

Listing 1. Example of functions in transactions.

```c
// parallel loop
for(int i=0; i<n; i++){
    TM_BEGIN();
    // low contention
    a[i]++;
    TM_END();
}
```

Listing 2. Example of small transactions inside a 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 [35]. 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 (~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 [27] 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, $\sim 20\%$ 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 [38] and VTune [30] 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 TxSampler, a lightweight and multi-scale profiler for hardware transactional memory. TxSampler utilizes event-based sampling [31] 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.

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.

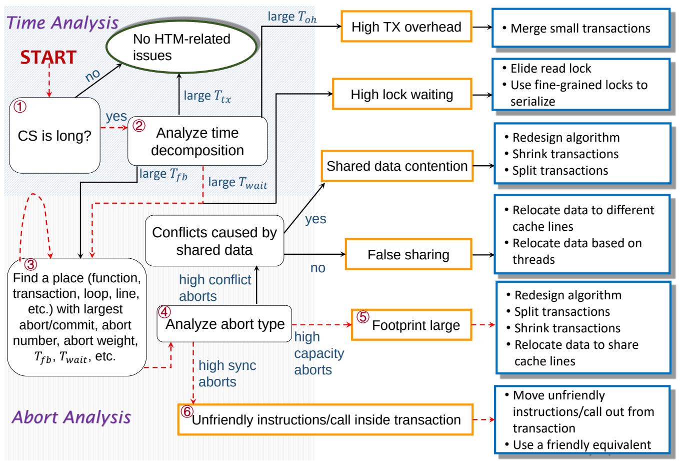

Time Analysis
START
① CS is long?
no
No HTM-related issues
yes
② Analyze time decomposition
large T_{tx}
large T_{oh}
High TX overhead
• Merge small transactions
High lock waiting
• Elide read lock
• Use fine-grained locks to serialize
large T_{fb}
large T_{wait}
Shared data contention
• Redesign algorithm
• Shrink transactions
• Split transactions
yes
Conflicts caused by shared data
no
False sharing
• Relocate data to different cache lines
• Relocate data based on threads
③ Find a place (function, transaction, loop, line, etc.) with largest abort/commit, abort number, abort weight, T_{fb}, T_{wait}, etc.
high conflict aborts
④ Analyze abort type
high capacity aborts
⑤ Footprint large
• Redesign algorithm
• Split transactions
• Shrink transactions
• Relocate data to share cache lines
high sync aborts
Abort Analysis
⑥ Unfriendly instructions/call inside transaction
• Move unfriendly instructions/call out from transaction
• Use a friendly equivalent

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 TxSampler addresses in leveraging PMU sampling to profile parallel programs with HTM.

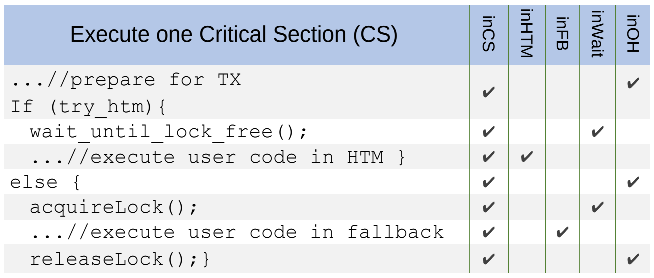
3.1 Challenge I: Handle Aborts due to Interrupts

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 [32] (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) [32] 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 ⟨`from, to`⟩ 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)	inCS	inHTM	inFB	inWait	inOH
`...//prepare for TX` `If (try_htm){`	✔				✔
`  wait_until_lock_free();`	✔			✔	
`  ...//execute user code in HTM }`	✔	✔			
`else {`	✔				✔
`  acquireLock();`	✔			✔	
`  ...//execute user code in fallback`	✔		✔		
`  releaseLock();}`	✔				✔

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.

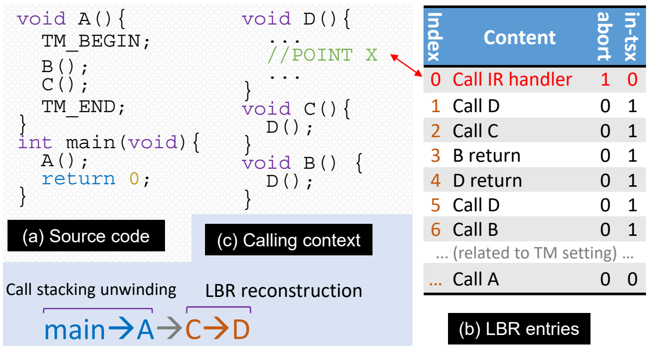
3.2 Challenge II: Attribute to Components

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 [17].



```
void A(){
  TM_BEGIN;
  B();
  C();
  TM_END;
}
int main(void){
  A();
  return 0;
}
```

```
void D(){
  ...
  //POINT X
  ...
}
void C(){
  D();
}
void B() {
  D();
}
```

Index	Content	abort	in-tsx
0	Call IR handler	1	0
1	Call D	0	1
2	Call C	0	1
3	B return	0	1
4	D return	0	1
5	Call D	0	1
6	Call B	0	1
	... (related to TM setting) ...		
...	Call A	0	0

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 [33].

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`→`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`→`D`. If the address of the `call C` instruction equals to the `from` address of LBR Entry 2, `C`→`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. TxSampler 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. TxSampler 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.

$$T = T_{tx} + T_{fb} + T_{wait} + T_{oh} \tag{2}$$

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_{tx} *is large* $\Longrightarrow$ the transactional path accounts for most execution time. TxSampler usually does not make optimization recommendation.
- T_{fb} *is large* $\Longrightarrow$ 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_{wait} *is large* $\Longrightarrow$ 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* $\Longrightarrow$ 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

```
/* Let $ctxt$ be the calling context where the PMU sample occurred */
/* Always accumulate work */
$ctxt.W$++;
/* Query the state from the RTM library */
int state := GetState();
if IsSampleInCS(state) then
    $ctxt.T$++;
    /* Query the abort bit $B_{abort}$ of the latest entry of LBRs */
    if $B_{abort}$ == 1 then
        $ctxt.T_{tx}$++;
    else if IsSampleInFallback(state) then
        $ctxt.T_{fb}$++;
    else if IsSampleInLockWaiting(state) then
        $ctxt.T_{wait}$++;
    else
        $ctxt.T_{oh}$++;
    end if
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. TxSampler 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, TxSampler also computes $r_{capacity}$ and $r_{synchronous}$.

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

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 per-thread 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-of-the-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 [43]. 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[1], transaction aborts/commits[2] and memory accesses[3] before the binary executes. TxSampler predefines default sampling periods for these events: 1×10^7 for CPU cycles, 1×10^4 for RTM aborts/commits, and 1×10^4 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 [52] 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.

[1]Event name: `cycles`.
[2]Event name: `RTM_RETIRED:ABORTED/COMMIT`.
[3]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 [43], Lee-TM [6], QuakeTM [20], and RMS-TM [37], (2) popular multi-threaded benchmark suites, such as PARSEC [9], Parboil [51], NPB [8], CORAL [40], SPLASH2 [50], Synchrobench [26] and SSCA2.2 [7], and (3) multi-threaded applications, such as LevelDB [11, 14], BPlusTree [18], KyotoCabinet [39], BART [54], Berkeley DB [44], Memcached [2] and PBZip2 [23].

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 [55] 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 -O3` 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 is averaged over five of seven executions by excluding the smallest and largest ones; the error bars show the variation. TxSampler incurs ~4% runtime overhead on average.

TxSampler requires no specific adaptation to profile short-running 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 [47], 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.

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.

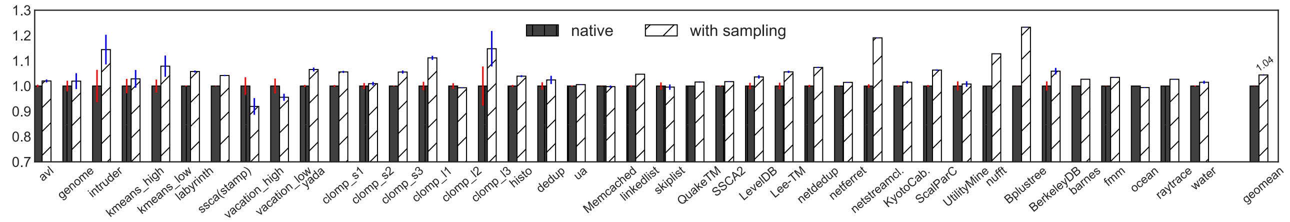


Figure 5. The runtime overhead of TxSampler.

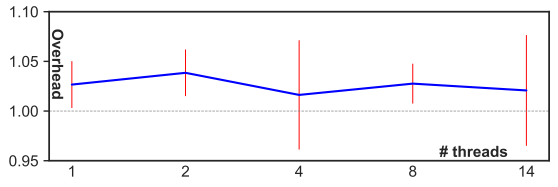


Figure 6. The average runtime overhead with variance bars according to different thread counts across all STAMP benchmarks.

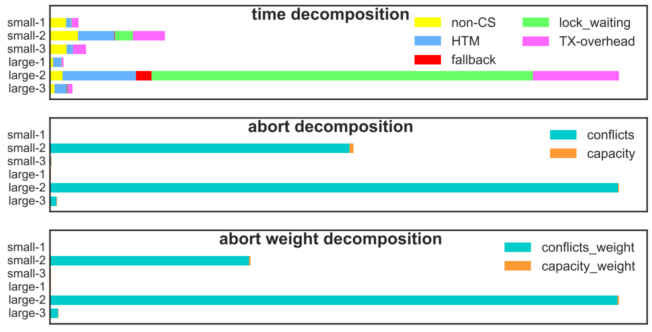


Figure 7. CLOMP-TM data from TxSampler.

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 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.

Code	Symptoms	Solutions	Speedup
*dedup [9]	high capacity abort	refine hash table	1.20×
	high synchronous aborts	remove system calls	
AVL Tree [15]	high T_{wait}	elide read lock	1.21×
*histo [51]	high T_{oh}	merge transactions	2.95×
	severe false sharing	sort the input array	
UA [8]	high T_{oh}	merge transactions	1.05×
vacation [43]	high abort rate	reduce transaction size	1.21×
*LevelDB [11, 14]	high abort rate	split transactions	1.05×
*SSCA2 [7]	high T_{oh}	merge transactions	1.10×
*netdedup [51]	high conflict abort	shrink transactions	2.6×
	high synchronous aborts	remove system calls	
*linkedlist [26]	high abort rate	limit transaction size	3.78×
	low average abort penalty	with auxiliary locks	

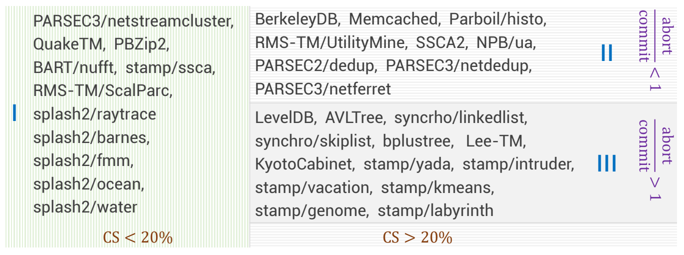
Table 2. Optimization overview (* newly found).

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 TxSampler, and demonstrate how TxSampler guides intuitive optimization with its decision tree model.



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

Figure 8. Application categorization.

8.1 PARSEC2 Dedup

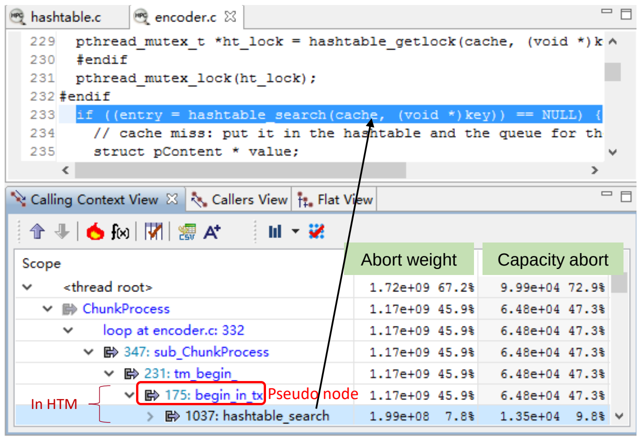
Dedup, a PARSEC benchmark, compresses data via deduplication. It employs pipeline parallelism, which consists of three major stages: `ChunkProcess`, `FindAllAnchors`, and `Compress`. By default, Dedup assigns four threads to each of the three stages. We iteratively optimize this benchmark based on the 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 random-access 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



```
  pthread_mutex_t *ht_lock = hashtable_getlock(cache, (void *)k
  #endif
  pthread_mutex_lock(ht_lock);
#endif
  if ((entry = hashtable_search(cache, (void *)key)) == NULL) {
    // cache miss: put it in the hashtable and the queue for th
    struct pContent * value;
```

Scope	Abort weight	Capacity abort
<thread root>	1.72e+09 67.2%	9.99e+04 72.9%
ChunkProcess	1.17e+09 45.9%	6.48e+04 47.3%
loop at encoder.c: 332	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_tx	1.17e+09 45.9%	6.48e+04 47.3%
1037: hashtable_search	1.99e+08 7.8%	1.35e+04 9.8%

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 `XOR`ed 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 [14] 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.

```
for (i = 0; i<img_width*img_height; ++i) {
  const unsigned int value = img[i];
  TM_BEGIN();
  if (histo[value] < UINT8_MAX)
    ++histo[value];
  TM_END();
}
```

Listing 3. Original HTM implementation in Histo.

```
for (i = 0; i<img_width*img_height/txn_gran+1; ++i) {
  TM_BEGIN();
  for(j=0; j<txn_gran && (i*txn_gran+j)<img_width*
       img_height; j++){
    const unsigned int value = img[i*txn_gran+j];
    if (histo[value] < UINT8_MAX)
      ++histo[value];
  }
  TM_END();
}
```

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 2D-array with a maximum bin count of 255 [51]. 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 HTM-based 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×.

8.4 Synchrobench LinkedList

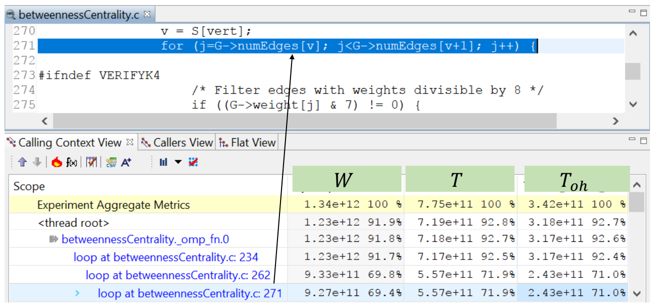
The Synchrobench suite includes several common data structures for synchronization evaluation [26]. It provides a transactional memory interface and can be easily adapted with HTM. This linkedlist micro-benchmark first creates a sorted linked list with 16, 000 elements and then keeps performing operations—20% insertion/removal and 80% search—on 32, 000 possible elements with 14 threads within 100 seconds. The performance is evaluated by the throughput, measured by the number of executed operations.

TxSampler identifies that more than 99% of the entire execution time is in critical sections. However, the ratio of abort/commit is as high as 4.2. Under default implementation, each of the three operations (`add`, `remove`, and `search`) is protected by a transaction. All three operations need to iterate the linked list and act on the proper positions. TxSampler shows that the average abort weight $\hat{w}_t$ is only 154 cycles, indicating that most transactions are aborted shortly after started. The severe conflict aborts are due to the fact that a change near the beginning of the linked list aborts all the transactions that have already traversed beyond this point. Aborted transactions restart from the beginning and increase the contention on the first few nodes of the linked list. Even worse, traversing the linked list introduces random memory accesses, causing capacity aborts as well.

According to the decision tree, TxSampler recommends reducing the transaction size to alleviate abort penalty and shrink memory footprint. Thus, we adopt an optimization similar to [13]. Instead of putting the entire linked list in a single transaction, we form multiple transactions, each of them processing a small number of nodes. That is, each thread maintains a local counter and accumulates the number of nodes processed in the current transaction. When a thread processes N nodes, it commits the transaction and starts a new one before processing the next node. In this way, each transaction has lower retry penalty and smaller memory footprint.

To guarantee the correctness in the boundaries, we use fine-grained auxiliary locks as follows:

1. Add a `lock` field in each node and initialize with *unlocked*.
2. When a transaction is about to commit (i.e., after handling N nodes), check the next node. If the next node is locked,



```
270                 v = S[vert];
271                 for (j=G->numEdges[v]; j<G->numEdges[v+1]; j++) {
272
273 #ifndef VERIFYK4
274                     /* Filter edges with weights divisible by 8 */
275                     if ((G->weight[j] & 7) != 0) {
```

Scope	W	T	T_{oh}
Experiment Aggregate Metrics	1.34e+12 100 %	7.75e+11 100 %	3.42e+11 100 %
<thread root>	1.23e+12 91.9%	7.19e+11 92.8%	3.18e+11 92.7%
betweennessCentrality._omp_fn.0	1.23e+12 91.8%	7.18e+11 92.7%	3.17e+11 92.6%
loop at betweennessCentrality.c: 234	1.23e+12 91.7%	7.17e+11 92.5%	3.17e+11 92.4%
loop at betweennessCentrality.c: 262	9.33e+11 69.8%	5.57e+11 71.9%	2.43e+11 71.0%
loop at betweennessCentrality.c: 271	9.27e+11 69.4%	5.57e+11 71.9%	2.43e+11 71.0%

Figure 10. TxSampler's time analysis identifies the line 271 with a large portion of transactional overhead T_{oh}.

retry the current transaction; otherwise, unlock the starting node of this transaction, lock the next node, commit the current transaction, and start a new transaction with the newly locked node as the starting node.

3. When a transaction performs memory writes in add and remove operations, only commit this transaction when the nodes involved in the writes are unlocked. Otherwise, retry this transaction.

This optimization avoids frequent transaction retry from the beginning of the linked list. Moreover, the fine-grained auxiliary locks only exist in the starting node of each transaction, maximizing the parallelism. Each thread randomly chooses a different N to avoid transactions starting from the same starting nodes. By selecting N in a range of $[10, 40]$, this optimization reduces abort/commit rate to 0.1, yielding 3.78× throughput improvement.

8.5 SSCA2

SSCA2 [7], a graph benchmark, has four computation kernels: graph construction, weighted edge extraction, subgraph extraction, and betweenness centrality calculation. We modify the benchmark by utilizing HTM to protect all the critical sections that are originally protected by locks. We set SCALE as 19 and the number of threads as 14.

TxSampler reports that the time consumption in critical sections T is as high as 58% of the total execution time W (shown in Figure 10). In addition, TxSampler shows low abort rate (abort/commit < 0.01), indicating rare memory conflicts. The further time analysis pinpoints high HTM overhead T_{oh}, which accounts for 44% of total execution time in critical sections. In addition, the time in the fallback path and the lock waiting state is negligible. Hence TxSampler recommends merging small transactions to reduce the HTM overhead.

After exploring the profiles via TxSampler's GUI, we find that SSCA2 has all transactions in loops. TxSampler identifies the first two loops (at line 271 and 366 in file `betweennessCentrality.c`) account for 92.5% of the total HTM overhead (Figure 10 only shows line 271). For optimization, we merge transactions in every D iterations into a large one. We set D as 3 and 8 for the two loops, respectively, yielding a 1.10× speedup for the entire program.

8.6 PARSEC3 netdedup

Netdedup has a computation kernel similar to dedup, except that it receives the input data from a client instead of directly reading from the input file. In PARSEC 3.0, netdedup improves the hash function of dedup in PARSEC 2.1. We run netdedup with three threads for each of the four stages for the server, monitored by TxSampler.

TxSampler shows that 95% of the time is in critical sections and it is worth investigating. Then TxSampler pinpoints two functions `write_file` and `sub_Compress` with the highest transaction abort.

Like dedup in Section 8.1, `write_file` function incurs synchronous aborts due to unfriendly system calls inside transactions. Applying the similar optimization technique, we move these system calls outside of the transaction.

For `sub_Compress`, most aborts are caused by memory conflicts. The transaction in `sub_Compress` compresses a block of data and then changes its status from UNCOMPRESSED to COMPRESSED. Since only the status of a block is shared and the data are private, we only need to protect the status variable instead of the whole compressing procedure. Thus, we shrink this transaction and only keep the status change operation inside.

This optimization reduces the total number of aborts by 52%, yielding a 2.6× speedup for the entire program.

8.7 AVL Tree

An AVL tree [22] is a self-balancing binary search tree, which is widely utilized in many applications, such as Linux kernel memory management [46]. We have implemented a parallel micro benchmark to mimic the scenario that different threads perform different operations, such as insertion and deletion, on an AVL tree concurrently. We first initialize an AVL tree with 16K nodes (i.e. half of its maximum size) and then spawn 14 threads to perform operations on the tree 2 million times per thread where the synchronization is guaranteed by hardware transactional memory. In our experiment, each thread performs 50% insertion and 50% deletion operations, so the size of the tree stays almost the same.

TxSampler reports that the overall transaction abort/commit ratio is 4.9, which means most of the critical sections are completed through the fallback path. In the fallback path, the thread needs to acquire the lock, which not only serializes the fallback path but also prevents the transaction path from execution. The lock waiting time T_{wait} accounts for as high as 81% of the total execution time. However, we find that serialization is unnecessary in many circumstances. For example, if a key already exists in the tree for an insertion operation or does not exist in the tree for a deletion operation, there is no need to serialize these operations, as the tree does not change. Thus, we can relax the serialization between the transaction and fallback paths: we allow the concurrent execution of read-only operations to avoid unnecessary lock waiting.

Our optimization for AVL tree follows the technique described in the literature [15]. We modify the RTM library and the AVL tree code as follows:

1. We add a global boolean variable `write_flag` to indicate whether any write operation has been performed so far.
2. We allow a thread to enter a transaction if the `write_flag` is not set. It needs not wait for the global lock. Once entering the transaction, the thread immediately adds `write_flag` to its read set by accessing it.
3. In the fallback path, we clear the `write_flag` just before releasing the global lock if the `write_flag` is set.

In AVL tree's source code, we set `write_flag` immediately before every write operation (i.e., real insertion and deletion) in critical sections. Thus, if `write_flag` is set in a transaction and the global lock is also acquired, this transaction is aborted due to the conflict on `write_flag`.

After this optimization, TxSampler reports that T_{tx} (time in transactions) is increased by 93% and both T_{fb} (time in the fallback path) and T_{wait} (time in lock waiting) are reduced by 26% and 27%, respectively. This data indicates that our optimization significantly improves the concurrency in AVL tree operations. This optimization leads to a 1.21× speedup for our AVL tree micro benchmark.

8.8 NAS UA

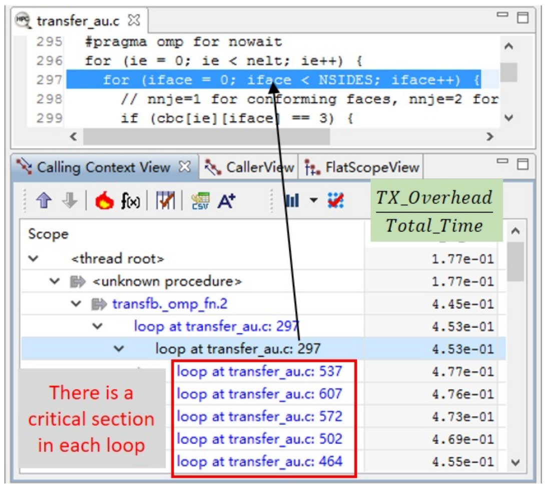
UA, from NAS Parallel Benchmark (NPB) suite, works on an unstructured adaptive mesh. In our experiment, we study its C-version [49]. We modify `transfer_au.c` to replace all critical sections (marked with `omp atomic`) with HTM. We run UA with input size C and 14 threads.

As shown in Figure 11, TxSampler reports that the ratio of transaction overhead over the total execution time (T_{oh}/W=18%) is high. The high overhead is incurred in `loop 297`, where many transactions keep updating a shared array. Thus we decide to merge small transactions into a large one to avoid high overhead.

We investigate the top five transactions with the largest overhead, highlighted in a red rectangle in Figure 11. We expand each transaction section to include all the operations of the innermost loop. Moreover, we coalesce several small consecutive transactions under the guidance of TxSampler. After this optimization, the HTM overhead T_{oh} of UA is reduced by 12%, leading to a 1.05× overall speedup.

8.9 STAMP Vacation

Vacation, highly optimized by IBM [43], emulates a travel reservation system. We launch Vacation with 14 clients to execute 16,777,216 operations and use TxSampler to monitor its execution. With the time analysis, TxSampler identifies


transfer_au.c
Calling Context View
CallerView
FlatScopeView
$\frac{TX\_Overhead}{Total\_Time}$
There is a critical section in each loop

```
295   #pragma omp for nowait
296   for (ie = 0; ie < nelt; ie++) {
297     for (iface = 0; iface < NSIDES; iface++) {
298       // nnje=1 for conforming faces, nnje=2 for
299       if (cbc[ie][iface] == 3) {
```

Scope	
<thread root>	1.77e-01
<unknown procedure>	1.77e-01
transfb._omp_fn.2	4.45e-01
loop at transfer_au.c: 297	4.53e-01
loop at transfer_au.c: 297	4.53e-01
loop at transfer_au.c: 537	4.77e-01
loop at transfer_au.c: 607	4.76e-01
loop at transfer_au.c: 572	4.73e-01
loop at transfer_au.c: 502	4.69e-01
loop at transfer_au.c: 464	4.55e-01

Figure 11. TxSampler pinpoints the loops with high HTM overhead.

a transaction in `client.c` accounts for more than 64% of the total execution time. Furthermore, the contention is high since $^{\#aborts}/_{\#commits} > 4$. With the abort analysis in this transaction, TxSampler pinpoints 31% of the aborts triggered in function `TMhashtable_find()`, which searches a hashtable. This function is first called to check the existence of an item (e.g. car, flight or room) and called again to find its price. If the item is for reservation, we need to search for the item for the third time and then update its availability.

To avoid the repeated searches, we save the item after the first search and reuse it later. This optimization (firstly reported in [24]) shrinks this transaction size, which reduces the number of aborts in `TMhashtable_find()` by 13% and achieves a 1.21× speedup for the entire program.

9 Related Work

Tools such as PGI's PGPROF [53], OpenSpeedShop [48], Oracle Solaris Studio [45], 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 [25, 41, 56, 57] 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×, 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) [21] 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 [35] 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 [42], 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 TSXProf 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 [38] 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 [30], 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 sampling-based 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. TxSampler 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.

References

[1] 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] Nachshon Cohen, Maurice Herlihy, Erez Petrank, and Elias Wald. 2017. POSTER: State Teleportation via Hardware Transactional Memory. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17).* ACM, New York, NY, USA, 437–438. https://doi.org/10.1145/3018743.3019026

[14] 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

[15] 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.

[16] 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

[17] Alexandre Eichenberger et al. 2014. OpenMP Technical Report 2 on the OMPT Interface. http://www.openmp.org/wp-content/uploads/ompt-tr2.pdf.

[18] embedded2016. 2016. GitHub - embedded2016/bplus-tree: B+ tree implementation in C. https://github.com/embedded2016/bplus-tree. (Accessed on 08/18/2017).

[19] 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

[20] 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.

[21] Matthew Gaudet. 2014. *Serialization Management Driven Performance in Best-Effort Hardware Transactional Memory Systems.* Master dissertation. University of Alberta.

[22] Evgenii Landis Georgy Adelson-Velsky, G.M. 1962. An algorithm for the organization of information. In *Proceedings of the USSR Academy of Sciences (in Russian).* 263–266.

[23] Jeff Gilchrist. 2015. Parallel BZIP2 (PBZIP2). http://compression.ca/pbzip2/. (Accessed on 08/25/2017).

[24] Bhavishya Goel, Ruben Titos-Gil, Anurag Negi, Sally A McKee, and Per Stenstrom. 2014. Performance and energy analysis of the restricted transactional memory implementation on haswell. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International.* IEEE, 615–624.

[25] 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

[26] 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.

[27] Part Guide. 2016. Intel® 64 and IA-32 Architectures Software Developer's Manual. *Volume 3C: System programming Guide, Part 3* 3C (2016).

[28] 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

[29] IBM. 2015. z/Architecture Principles of Operation. http://www-01.ibm.com/support/docview.wss?uid=isg2b9de5f05a9d57819852571c500428f9a. *SA22-7832-09* (2015).

[30] Intel 2013. Intel VTune Amplifier XE 2013. http://software.intel.com/en-us/intel-vtune-amplifier-xe. Last accessed: Dec. 12, 2013.

[31] Intel Corp. NA. Hardware Event-based Sampling Collection. https://software.intel.com/en-us/node/544067.

[32] 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.

[33] Intel Corporation. 2010. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2, Number 253669-032.

[34] Intel Corporation. 2016. Intel 64 and IA-32 Architectures Software Developer's Manual.

[35] 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.

[36] Ken Kennedy and John R. Allen. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[37] 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.

[38] Andreas Kleen. 2013. Intel Transactional Synchronization Extensions (Intel TSX) profiling with Linux perf. https://goo.gl/BlltQ4. 3 May 2013.

[39] FAL Labs. 2011. Kyoto Cabinet: a straightforward implementation of DBM. http://fallabs.com/kyotocabinet/. (Accessed on 08/18/2017).

[40] Lawrence Livermore National Laboratory. 2014. LLNL Coral Benchmarks. https://asc.llnl.gov/CORAL-benchmarks. Last accessed: Dec. 12, 2013.

[41] Yossi Lev. 2010. *Debugging and Profiling of Transactional Programs*. Ph.D. dissertation. Brown University.

[42] 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

[43] 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.

[44] Oracle. 2007. Oracle Berkeley DB. http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html. (Accessed on 08/18/2017).

[45] Oracle. 2017. Oracle Solaris Studio. http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html.

[46] David A Rusling. 1999. The linux kernel.

[47] 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.

[48] 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.

[49] Sangmin Seo, Gangwon Jo, and Jaejin Lee. 2011. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*. IEEE, 137–148.

[50] 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.

[51] 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).

[52] 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.

[53] The Portland Group. 2011. PGPROF Profiler Guide Parallel Profiling for Scientists and Engineers. http://www.pgroup.com/doc/pgprofug.pdf.

[54] 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.

[55] 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.

[56] 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

[57] 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 Artifact Appendix

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**
- **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.

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.

```
$ get_benchmark_and_input.sh
$ cd /data/txsampler_benchmark
$ 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:

```
$ generate_profile.py -l  # list available applications
$ 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/txsampler-ae), which can be directly opened by HPCViewer.