Visualizing Transactional Memory

Justin E. Gottschlich
Intel Corporation
Programming Systems Lab
justin.e.gottschlich@intel.com

Maurice P. Herlihy
Brown University
Dept. of Computer Science
herlihy@cs.brown.edu

Gilles A. Pokam
Intel Corporation
Programming Systems Lab
gilles.a.pokam@intel.com

Jeremy G. Siek
University of Colorado-Boulder
Dept. of Electrical and Computer Engineering
jeremy.siek@colorado.edu

ABSTRACT
This paper presents TMProf, a transactional memory (TM) profiler, based on three visualization principles. These principles are (i) the precise graphical representation of transaction interactions including cross-correlated information and source code, (ii) visualized soft real-time playback of concurrently executing transactions, and (iii) dynamic visualizations of multiple executions. We describe how these principles break new ground and create new challenges for TM profilers.

We discuss our experience using TMProf with InvalSTM, a state-of-the-art software TM, and show how TMProf’s feedback led to the design of two new contention managers (CMs). We demonstrate the performance benefits of these CMs, which generally led to improved performance as the amount of work and threads increase per benchmark. Our experimental results show that iBalanced, one of our newly designed CMs, can increase transaction throughput by nearly 10× over iFair, InvalSTM’s previously best performing CM.

Categories and Subject Descriptors: D.1.3 [Concurrent Programming]; Parallel programming; D.1.7 [Visual Programming]; D.2.8 [Metrics]: Performance measures.

General Terms: Algorithms, Measurement, Performance

Keywords: Transactional Memory, Visualization, Profiler

1. INTRODUCTION
With the announcement of upcoming hardware transactional memory (HTM) support in Intel’s Haswell and IBM’s Blue Gene/Q processors, tools that help programmers write correct and efficient transactional memory (TM) programs are now more important than ever [1,27]. Programs that use TM, like any other class of programs, need effective profilers. While many profiling tasks are common to all software systems, TM-based systems present unique profiling challenges [29,32]. In particular, if two transactions access the same data object, and at least one access is a write, then the transactions conflict [25,30]. Such conflicts can have a substantial impact on system performance [14]. In some TM implementations, conflicts are detected lazily, causing transactions that conflict to be aborted, that is, rolled back and restarted. In others, conflicts are detected eagerly, causing some transactions to block waiting for conflicting transactions to finish. No matter how conflicts are resolved, a TM profiler should help programmers understand how and when conflicts occur and provide insight into how they can be managed to improve a program’s performance.1

Profilers for single-threaded programs generally capture at least the following two performance metrics: (i) the total execution time inside a function and (ii) the number of times the function is executed [13]. When these metrics are used together, programmers can generally resolve most single-threaded performance bottlenecks. If similar metrics were used for multi-threaded TM programs, a TM profiler might track (i) the total execution time per transaction and (ii) the number of times each transaction is executed. Unfortunately, these metrics alone have limited utility in TM profiling because they do not capture key pieces of contextual information.

For example, only considering a transaction’s total execution time does not isolate the effects of transactional interference, those conflicting transactions that execute concurrently causing at least one transaction to abort or stall. It is important to capture the effects of transactional interference because such information quantifies the amount of time a transaction’s execution has, or has not, been impacted by conflicting transactions. Furthermore, only considering the number of times a transaction has executed (i.e., committed) will not include the number of times it was previously aborted. Understanding aborted transactional data can shed light on the type and potency of optimizations for transactions.

To address these and other issues with single-threaded profilers, TM profiler and optimizer research has emerged with a central focus on how aborts impact performance. Lev’s and Zyulkyarov et al.’s TM profiling research provides various facilities to track and quantify aborts [24,38], while Chakrabarti et al.’s TM optimizer automatically reduces the

1 An alternative to programmers using profiling data to improve a TM program’s performance is to use a system that automatically switches to the most efficient software TM, from a given set, based on the program’s execution pattern [16].
number of aborts produced by an execution by utilizing the information generated in its run-time abort graph [5]. In addition to the reasons described above, these works observe that aborts represent the most obvious form of wasted work and therefore are a natural target for optimization.

1.1 Overview of TMProf

Like these works, the profiler we present in this paper, TMProf, provides specific techniques to track and quantify aborts. Yet, unlike them, TMProf can help programmers understand the importance of aborts (and commits) with precise cross-correlated information. This information distills how aborts and commits in one thread influence the forward progress of transactions in another. In Section 2 we illustrate how the right type of abort at the right time can improve performance, rather than degrade it, and how visualizations reduce the challenge of identifying such behavior. We extend this notion in Section 3 to demonstrate how certain commits can degrade performance rather than improve it. We then use these techniques together with TMProf’s visualizations to find new contention management (CM) strategies.

In addition to cross-correlated abort and commit information, TMProf provides a visualized playback feature that replays the program’s execution in soft real-time [4]. TMProf’s playback feature conveys the complex interactions between transactions as they unfold in time and presents the effects of such interactions in the broader context of the program’s complete execution. With such playback, a programmer can track transactional interference amongst transactions in specific phases of the program and understand how such interference impacts the immediate and long-term forward progress of the execution.

![Figure 1: Using TMProf to Replay an Execution.](image)

TMProf’s visualizations are also dynamic and can display as many complete program executions as are necessary to understand and analyze a specific behavior. A visualization displaying a single execution shows how transactions interact across several threads. A visualization displaying multiple executions of the same program illustrates how a change to the program has impacted its performance and the reason(s) for such a difference. TMProf can also display the transactional source code associated with any number of visualized executions, as shown in Figure 1, where each thread is represented as a horizontal bar and the various colors identify conflicts between transactions.

Providing TMProf’s visualization functionality does not come without cost and these costs create new challenges in both the data collection and the visual representation aspects of the system. Like the works of Chakrabarti et al. [3], Lev [24], and Zyulkyarov et al. [38], TMProf must gather enough data to provide meaningful profiling information without incurring a probe effect, where a program’s behavior is unintentionally altered by an external system that is analyzing it [9]. Minimizing the likelihood of a probe effect requires a trade-off between the amount of data gathered and the profiler’s precision. Yet, unlike prior TM profiler research, TMProf must perform full execution traces because it provides cross-correlation information rendered across entire executions. Moreover, TMProf’s execution playback functionality requires precise back-end timing data collection for near real-time playback. Gathering this data increases the data density per transaction and, therefore, the likelihood of a perturbation of the recorded program’s execution, as well as increasing the computational workload during TMProf’s front-end visualization.

For the remainder of this paper, we gradually explore TMProf’s visualization capabilities, the challenges that were overcome to support its functionality, and the TM optimizations we constructed based on its feedback. This paper makes the following technical contributions.

1. We present TMProf, a TM profiler that provides (i) graphical representation of transaction-level interactions including cross-correlated information and transactional code, (ii) visualized soft real-time playback of concurrently executing transactions, and (iii) dynamic visualizations of multiple TM executions (Section 2).

2. We present the challenges we encountered building TMProf and the unique design and implementation decisions we made to overcome them (Section 2).

3. We present a case study using TMProf, including the algorithmic details of the CMs we built using TMProf’s visualizations (Section 3).

4. We provide 8- to 24-threaded benchmarks that demonstrate the benefits of the CM optimizations we built yielding nearly a 10× scalable improvement over InvalSTM’s previously most efficient CM [12] (Section 4).

2. VISUALIZATION

Profiling parallel programs is much more challenging than profiling sequential programs. For multithreaded programs, shared-memory synchronization is a principal source of latency [2, 22]. For TM programs, synchronization costs may be even more substantial, because conflicts can cause longer waits or transaction aborts [14]. In general, reducing conflicting data accesses enhances concurrency and reduces the need to abort and restart transactions. One of the most important functions of a TM profiler is thus to capture and convey how transactions interact, with the goal of helping the developer understand and reduce conflicts. This observation implies that an effective TM profiler should be primarily visual, because graphical representation is the best way to convey complex interactions unfolding in time.

Figure 2 shows a four-threaded TM execution as presented by TMProf. Each horizontal bar represents a single thread.
The numbers to the left and right of each thread indicate the thread’s respective starting and ending times in milliseconds. Each thread has three bar segments: two outer and one inner. The outer segments indicate if the transaction was committed (green) or aborted (some other color). The inner segment indicates the transaction type when it was committed or aborted: green indicates read-only transactions, grey indicates transactions that both read and write, and black indicates write-only transactions.

At startup, TMProf assigns a unique color to each thread, which is indicated by the rectangle to the left of the thread’s execution. When a transaction is aborted, it is assigned either a blue color, which indicates the transaction was self-aborted, or a thread’s color, which indicates that the corresponding thread caused the transaction to abort due to a conflict.

2.1 Transaction Interference

Perhaps the most important function of a TM profiler is its ability to provide a clear and precise picture of how and why transactions abort. Because aborted transactions represent wasted work, they are a natural target for optimization [37]. Understanding why transactions abort requires an understanding of how their data accesses conflict [18]. Once the nature of the conflicts is understood, the developer can attempt to change or reduce conflicts, perhaps by restructuring the data, changing transaction functionality, or modifying the TM system or its contention manager (CM) [15, 33]. To achieve this kind of understanding, each of the prior works from Chakrabarti et al. [5], Lev [24], and Zyulkyarov et al. [38] provide unique facilities to track aborts.

TMProf introduces advanced cross-correlation abort and commit tracking facilities which complement the existing ones provided by prior research. TMProf’s cross-correlation feature visually conveys how aborts and commits improve or degrade the overall performance of the program. To demonstrate this, consider the execution shown in Figure 3. In this example, each thread performs an equal number of inserts on a shared linked list and the CM strategy is designed so a committing transaction always aborts conflicting inflight transactions.

The transactions of the thread second from the top suffer almost no aborts because its transactions insert items near the front of the linked list. Its transactions are shorter running and reach their commit phase sooner than the other transactions and its total execution time reflects this, which is almost $5 \times$ faster than the slowest thread. The top-most thread’s transactions suffer numerous aborts because its transactions insert items at the tail of the linked list. Its transactions run longer than others, making them susceptible to aborts by other committing transactions.

When considered in isolation, each thread’s behavior performs exactly as it should to keep aborts to a minimum. If the goal is to minimize the total number of aborts, the TM’s behavior is close to optimal. Yet, by viewing TMProf’s visualization of the program, one can immediately determine that an optimally low number of aborts is not necessarily what will make this program execute more efficiently. Instead, improved performance will likely come from more balanced execution times across all threads, which may mean having a disproportionately large number aborts for
the fastest executing thread, which can improve the efficiency of the slowest executing thread by reducing its aborts.

This example illustrates the importance of understanding aborts and commits within the context of an entire program’s execution. While it may seem generally reasonable to reduce aborts in an attempt to improve performance because they represent wasted work, when universally applied, such an approach may counterintuitively degrade performance, as shown in Figure 3, resulting in a notable disparity in execution time between threads.

2.2 Execution Playback

Multithreaded execution playback can be performed online or offline. An online playback system, such as Patil et al.’s PinPlay 31 or Lev’s T-PASS 24, replays an execution by re-executing the original program, instruction by instruction. These systems also include complex logic for correct multithreaded replay, which when combined with their instruction-by-instruction playback, can reduce replay speeds by upwards of 100x compared to a normal execution. Offline playback systems, such as TMProf, replay an execution by simulating certain aspects of the recorded execution using logged information that was generated during the original execution 7. While offline replayers do not re-execute each instruction, and, therefore, have limitations of the program state they can reproduce, they have the advantage of being able to simulate aspects of the original execution at various speeds, which can be instrumental in identifying and resolving performance bottlenecks.

TMProf supports forward and reverse playback control, pause, and user-controllable playback rates (from 0.0001x to 1000x normal execution speed). Although we cannot demonstrate TMProf’s playback functionality within this paper, a recorded demonstration of its playback functionality can be downloaded at http://tinyurl.com/7v5oj47 and viewed in any media software player compatible with AVI recordings. TMProf has been optimized so that it can manage the soft real-time playback of roughly a hundred threads at a time. In our experiments, however, we have found that replaying just a few threads, more than two but less than eight, provides enough detail of program contention to create optimizations that scale to dozens of threads.

2.3 Dynamic Visualizations

The work of Lev 24 and Zyulkyarov et al. 38 provides functionality to analyze and inspect a multithreaded TM execution from different visual viewpoints. We extend these techniques in TMProf by enabling full program executions to be visually compared to each other in a side-by-side fashion. Such side-by-side comparisons are useful because they provide the programmer with immediate feedback on how optimizations have impacted the dynamic behavior of the transactions, such as their throughput and interaction with one another. Figure 4 shows two side-by-side executions. TMProf also supports executions to be overlaid on top of each other with minimal loss of visual integrity 7.

Once a TM profiler supports the visualization of multiple executions and execution playback, a fixed representation of the visualizations may be insufficient for a programmer to gain a comprehensive understanding of the execution(s) 24. Instead, rendering of such complex visualizations should be

Figure 4: TMProf Overlaying Two Executions.

3. DESIGN AND OVERHEAD

Although TMProf can be used with any TM, its abort tracking system is particularly useful for TMs that use some form of invalidation, a conflict detection strategy where one active transaction aborts or stalls other active transactions 17 10. This is because such a system can generally capture information about which transactions conflict with others. Once this information is collected, it can be viewed in the context of TMProf’s transactional interference visualization. TMs that use validation, where a transaction checks its own memory for conflicts with previously committed transactions, may be less useful for TMProf because in these systems information detailing which transaction caused another transaction to abort may not be available. Without such information, TMProf cannot visualize the correlation of conflicts between transactions.

Herlihy and Moss’s original HTM is an example of an invalidation-based system, which uses a snoop-based coherence invalidation mechanism 21. Software TMs (STMs) that use invalidation include RSTM’s mixed invalidation 34, DSTM and DSTM2’s eager write-write conflict detection 20, Fraser and Harris’s OSTM which uses lazy write-write conflict detection 8, and InvalSTM which uses full invalidation 12. Furthermore, any TM that uses a CM to manage transactional conflicts can benefit from TMProf. The justification for each abort will be captured in TMProf’s data gathering phase and then displayed by its visualization, providing cross-correlated transactional interference information, which can then be used to optimize the program or TM system.

3.1 No Sampling

A common problem associated with profilers, regardless of their single or multithreaded nature, is the computational and spatial overhead they introduce when gathering execution data 8 7. This overhead is sometimes referred to as intrusion overhead, because the external system’s intrusion increases the execution time of the program it is analyzing 26. In an attempt to minimize the impact of intrusion overhead many profilers use sampling, where a subset of program’s execution is analyzed to estimate the behavior of the entire program. Sampling can reduce both the amount of time the program’s execution is perturbed and the spatial impact of recording data, both of which can reduce the like-
likelihood of a probe effect [3, 9]. For these and other reasons, Chakrabarti et al.'s TM optimizer and Lev's TM profiler use sampling. Zyulkyarov et al.'s profiler supports sampling, but they do not use it [38].

Unfortunately, because TMProf's visualization and replay features require complete execution traces, it cannot use sampling. If sampling were used, TMProf's execution traces would contain gaps resulting in incomplete visualizations. Furthermore, TMProf would be unable to provide its cross-correlated transaction interference feature, because not all transaction interactions would be captured. This places TMProf in a unique design space when compared to prior works that perform sampling. However, just as the prior systems of Chakrabarti et al., Lev, and Zyulkyarov et al., must minimize their intrusion overhead and probe effect, so too must TMProf, otherwise its resulting reports might be misleading. We explain how TMProf achieves this in the following sections.

3.2 Begin and End Events

To make TMProf practical, it does not track transactional memory accesses as is done in prior works [3, 23, 38]. Although such additional data would provide TMProf with more exhaustive execution information, we believe it would exceed limitations in both TMProf's graphical front-end and its data collection back-end. A data collection system that tracks transactional read, write, and opacity validation events has an \(O(N)\) space complexity, where \(N\) is the number of read and write elements per transaction, and, usually, an \(O(N^2)\) or \(O(N)\) time complexity if opacity read validations are tracked or not, respectively.

To reduce this complexity, TMProf limits its data collection to begin and end transactions, as shown in Figure 5. Transactions generally begin in the same fashion, but end with either a commit or an abort. In Figure 5 we label these end actions (EA) as \(EAN\), where \(N\) is a unique identifier for the transaction. By only tracking begin and end events, TMProf's space and time overhead for both its data collection back-end and its graphical front-end is reduced to \(O(1)\) per transaction.

3.3 Space and Time Optimizations

To reduce TMProf's intrusion overhead, it uses thread local storage (TLS) and the thread-level parallelism that already exists in the multithreaded executions that it analyzes [28]. Although not novel, with TLS and a sufficient number of available cores, TMProf can concurrently gather execution data for any number of threads without introducing additional serialization. This is important in reducing intrusion overhead and limiting program perturbation.

TMProf also uses in-memory stream buffers when gathering execution data to reduce the computational latency associated with non-volatile I/O storage. Although in-memory buffers have the advantage of reducing time overhead, they introduce space overhead. To reduce this overhead, TMProf uses a hash-based compression algorithm that encodes a transaction with a unique identifier the first time it is encountered. When the same transaction is subsequently

![Figure 5: TMProf's High-Level Design.](image)

To demonstrate the impact this has on TMProf's graphical front-end, consider Figure 6, which shows the number of polygons needed to display various events. For 64-threaded visualizations, with 16,384 transactions per thread and 32 reads and 1 write per transaction, displaying read, write, and opacity events requires three orders of magnitude more polygons than only displaying begin and end events. If opacity events are ignored, displaying only read and write events requires roughly an order and a half of magnitude, about \(34 \times\) more polygons than displaying only begin and end events. Because TMProf supports soft real-time playback, some percentage of these polygons, based on the current replay point and the user's chosen dynamic display, must be displayed roughly \(30 \times\) per second so the visualizations are perceived as occurring in real-time. For these reasons, having TMProf display begin and end events, seems to align well with its soft real-time playback requirement.

![Figure 6: TMProf's Data Representation Overhead.](image)

### 3.2.1 Haswell’s Restricted Transactional Memory

An additional advantage of using begin and end events is their direct mapping to the Restricted Transactional Memory (RTM) instructions in Intel's upcoming Haswell processor [1]. Although we have not yet tested TMProf with Haswell, as it is not yet available, the Transactional Synchronization Extensions (TSX) instructions of Haswell's RTM – XBEGIN, XEND, and XABORT – map directly to TMProf's begin transaction, commit transaction, and abort transaction events. As such, it is our belief that TMProf will integrate well with Haswell's RTM. We plan to integrate TMProf with Haswell as soon as it becomes available.
executed, the in-memory representation of that transaction is reduced to a single identifier (32-bits in TMProf’s current implementation). TMProf uses an additional byte for the transaction event type and a 32-bit segment for the time of the event. For commit events one additional byte is needed to capture the type of transaction (i.e., read, write, or read and write). Abort events use an extra 32-bits to capture the unique thread identifier that caused the transaction to abort, which can be reduced to 8-bits for most practical purposes.

When a thread terminates, TMProf transfers the in-memory transactions of that thread to a thread-specific trace file, as shown in Figure 5. In TMProf’s initial design, it decompressed each transaction that was stored in the in-memory buffers before writing them to their appropriate trace file. However, this resulted in unacceptably large trace files. In the current design, all in-memory transactions remain compressed in their respective trace files. These files are then decompressed during the startup of TMProf’s graphical front-end, which reduced the trace file sizes by \( \approx 80\% \) to 95\% depending on the code size associated with the transactions, number of unique transactions, and frequency in which transactions were invoked.

### 3.4 Intrusion Overhead

Figure 7 displays TMProf’s intrusion overhead for the microbenchmarks we discuss in Section 4. These microbenchmarks are linked list, hash table, red-black tree, and 1-writer / N-readers. We list TMProf’s intrusion overhead only for 4-threaded instances of these benchmarks because the optimizations discussed in Section 4 were found by profiling 4-threaded versions of the respective workloads.

![Figure 7: TMProf’s Data Collection Overhead.](image)

In each benchmark, with the exception of the 1-writer / N-readers, each thread populates its shared container with 4096 nodes. For the 1-writer / N-readers benchmark, one thread performs tail insertion to a shared linked list, while the other three threads perform iterative lookups on it. TMProf incurs 3.58\%, 14.32\%, 8.94\%, and 9.61\% intrusion overhead for the linked list, hash table, red-black tree, and 1-writer / N-reader benchmarks, respectively.

Chakrabarti et al.’s TM optimizer, which uses sampling to gather profiling data, has an intrusion overhead percentage that ranges from 14.1\% to 21.1\% for STAMP benchmarks using 16-threaded executions. Zyulkyarov et al.’s TM profiler, which captures read and write set sizes and begins and end events for every transaction and the address of conflicting memory locations for aborted transactions, has an intrusion overhead that ranges from -2.2\% to 252\% for STAMP benchmarks using 4-threaded executions. Lev does not report intrusion overhead introduced by his profiler.

It is important to note that there are many factors that could lead to substantial differences in the intrusion overhead between TMProf and Chakrabarti et al.’s RAG optimizer and Zyulkyarov et al.’s TM profiler, such as, benchmarks analyzed, number of threads used, sampling versus tracing, processor configuration, and run-time events captured. Therefore, it would be inappropriate for us to make any claims about TMProf’s intrusion overhead compared to these works. With that said, we include Chakrabarti et al.’s optimizer and Zyulkyarov et al.’s profiler overhead numbers to demonstrate that TMProf is competitive with them.

### 4. Observations and Experiments

Regardless of the features a profiler provides, its intrusion overhead, or the data it collects, the measure of a profiler’s true utility comes from its ability to help programmers optimize programs. In this section, we present several examples of such cases using TMProf. Although the figures in this section do not include TMProf’s transactional source code feature, where the code associated with each transaction’s visualization is shown along side it, we would like to note that we relied, heavily, on this feature during our experimental analysis, as it helped us determine which transactions were conflicting and, consequently, how we could build optimizations that would efficiently manage such conflicts.

#### 4.1 Experimental Overview

In our prior work developing InvalSTM, we found that CMs would be a natural target area for future optimizations. This is because InvalSTM provides a CM with more information (i.e., knowledge) than is generally available in TMs that do not use full invalidation [10, 11]. In this section, we show how TMProf helped us identify ways to more efficiently manage contention, leading to the construction of more efficient CMs. We describe how TMProf guided us to the construction of each CM’s algorithmic details, and the experiments we used to demonstrate each CM’s scalability and generality, or lack thereof. We used InvalSTM, a state-of-the-art STM that is competitive with TL2 [5], as our baseline system and the iFair CM, which is InvalSTM’s best performing CM [12]. All experiments were run on a 24-way Intel Xeon X5650 running at 2.67GHz.

We profiled the same benchmarks used in the initial InvalSTM study: (i) 1-writer / N-readers, (ii) linked list, (iii) hash table, and added a (iv) red-black tree (RBtree) benchmark. For all benchmarks, except the 1-writer / N-readers benchmark (details forthcoming) each thread populates the same shared container using a non-overlapping range of incrementing values. Once a thread has inserted all of its values, it performs lookup operations on the inserted values as a sanity check. We also added remove operations for each element inserted after the sanity check had been performed. Each of these operations, insert, lookup, and remove, were programmed as individual transactions.
4.2 The iFair CM

To understand how we derived our optimized CMs from TMProf’s visualizations, we must first provide a brief outline of iFair. In general, iFair aims to limit wasted work by minimizing aborts and enabling transactions with large read and write sets to commit. Each transaction is given a dynamic priority that is increased each time it is aborted and a weighted average value, \( V \), that is based on its dynamic priority multiplied by the number of reads and writes it has performed. Writes are given a greater weight than reads because they require more work to perform. When conflicts are resolved at commit-time, the only time they are resolved with InvalidSTM, a committing transaction’s \( V \) is compared against all of the other transactions with which it conflicts. If the committing transaction’s \( V \) is greater than the sum of the conflicting transactions’ \( V \), it is permitted to commit, aborting the conflicting transactions. Otherwise, it aborts. iFair ensures no transaction starves by increasing the transaction’s dynamic priority each time it is aborted. We call iFair a transaction-level fair CM, because it attempts to achieve fairness with respect to individual transactions. For example, if transaction \( T_1 \) is aborted by transaction \( T_2 \), \( T_1 \)’s priority is subsequently increased and is therefore less likely to be aborted by future transactions. Once a transaction successfully commits, its dynamic priority is reset to 1.

4.3 The threadFair CM

We began our experiments by analyzing the behavior of iFair using the 1-writer / N-readers benchmark, where one thread writes to a shared container and the other N threads read values from it. When contention arises, it is always with the writer transaction, because read-only transactions cannot conflict with one another. As shown in Figure 8(a) highlighted by the red rectangles, the number of aborts is reasonably low. This means that, in practice, the reader transactions are unlikely to simultaneously access the same location as the writer transaction. Therefore, it is unlikely that iFair can be optimized by reducing aborts, at least for this benchmark. Yet, because reducing aborts is the most common way to optimize TMs, such approaches have little opportunity for optimization here.

By analyzing TMProf’s visualization of the 1-writer / N-readers benchmark using iFair, as depicted in Figure 8(a), we were able to make the following observations. First, we noticed that the writer thread, the top-most thread, completes its execution ≈ 40% sooner than the other threads. Second, all aborts occurred in the reader threads. Third, by successively playing, rewinding, and replaying at slower rates, we were able to determine that, in many cases, aborts in the reader threads (the bottom three threads of Figure 8(a)) come in successive waves. We would not have observed this without TMProf’s playback feature, because when such self-aborts occur, they cannot be seen from a static view. This is because a static view provides no way to distinguish when a self-abort is a single, long running transaction or a grouping of multiple, short running ones.

From these observations, we tried to imagine how to build a more efficient CM. From our first observation, it seemed clear that a more efficient CM would extend the execution time of the writer thread to shorten the execution time of the reader threads. It seemed that we could achieve this by using the information we found from our second observation, which illustrated that all aborts originated from the reader threads. Moving those aborts to the writer thread would likely increase the writer thread’s overall execution time, subsequently reducing the readers’ execution time. Our third, and most important observation, provided the clue that iFair’s transaction-level fairness, where the priority of a transaction is increased each time it is aborted, did not have any meaningful impact in this benchmark. This is because the reader transactions were aborted in succession, which iFair’s transaction-level fairness was supposed to mitigate. Therefore, an alternative fairness mechanism would be needed. To fix this behavior, we would need to rely on something other than iFair’s weighted \( V \) to balance the execution.

From these conclusions, we built the threadFair CM, which performs thread-level fairness, an algorithm that enables transaction \( T_1 \) to abort transaction \( T_2 \) only if \( T_1 \)’s thread has fewer commits than \( T_2 \)’s thread. To correct the behavior we saw in our third observation with iFair’s \( V \), threadFair provides no weighted value for a transaction. This guarantees that thread-level fairness will be respected by the writer transactions, preventing them from unfairly aborting other transactions.

Algorithm 1 describes threadFair in pseudocode. It returns true when the committing transaction, \( commitTx \), is permitted to abort the set conflicting transactions, \( conflicts \),

![Figure 8: 1-Writer / N-Readers Visualized in TMProf.](image-url)
otherwise it aborts \texttt{commitTx}. \texttt{NextTx} returns a pointer to the next transaction in a set of transactions that are passed to it in the first parameter, returning NULL when the end of the set has been reached. If NULL is supplied, it begins iteration over the set. The \texttt{commits()} function returns the total number of commits for the thread. The \texttt{commitTx.sleep(S)} puts \texttt{commitTx}'s thread to sleep for \texttt{S} milliseconds after it is aborted.

**Algorithm 1 The threadFair Contention Manager**

1: procedure \texttt{threadFair(Transaction \texttt{commitTx}, TransactionSet \texttt{conflicts})}
2: \hspace{1em} boolean \texttt{canAbort} = false
3: \hspace{1em} Transaction \texttt{tx} = \texttt{NULL}
4: \hspace{1em} loop
5: \hspace{2em} \texttt{tx} = \texttt{NextTx(\texttt{conflicts}, \texttt{tx})}
6: \hspace{2em} if (\texttt{tx} \equiv \texttt{NULL}) then
7: \hspace{3em} break
8: \hspace{2em} if (\texttt{commits(\texttt{commitTx})} > \texttt{commits(\texttt{tx})}) then
9: \hspace{3em} \texttt{canAbort} = false; \texttt{\texttt{commitTx.sleep(S)}}; break
10: return \texttt{canAbort}

TMProf’s visualization of threadFair, shown in Figure 9(c), successfully illustrates our assumptions about the how to improve the CM performance of the 1-writer / N-readers benchmark. The visualization shows (i) the writer thread’s time is increased in favor of shortening the reader threads, (ii) aborts are moved from the reader threads to the writer thread, and (iii) using thread-level fairness, rather than transaction-level fairness, keeps the threads’ overall progress closer together. Figures 8(a) and 8(c) show that threadFair (12443ms) yields a 6.6% performance improvement over iFair (13264ms). In Figure 10, we applied threadFair to 8-, 16-, and 24-threaded 1-writer / N-readers workloads and found that it scales well. For example, in the 24-threaded 1-writer / N-readers benchmark, threadFair, labeled “thread”, outperforms iFair by upwards of 2.14×, a notable margin for a state-of-the-art STM.

**4.4 The iBalanced CM**

The first observation we made after creating threadFair is that while it performs well for the 1-writer / N-readers benchmark, it does not yield the same performance benefits for other benchmarks, as shown in Figure 9(c) which visualizes a 4-threaded RBtree. To begin to understand why, we explored threadFair’s RBtree execution using TMProf’s replay functionality. This led to a second observation: a substantial number of aborts, across all threads, were self-aborts (i.e., blue horizontal bars). We were unable to see this behavior using TMProf’s static view because, (i) with a static view aborts are blended together, making it challenging to visually disambiguate one transaction’s color from another; (ii) in playback mode, each thread’s most recent transaction is displayed with a whitespace to its right, making its color easier to discern; and (iii) in playback mode, our concentration stayed fixed on the active transaction, making it easier to comprehend the events as they occur in sequence. We believe these reasons further reinforce the importance of TMProf’s playback functionality.

Our third and final observation required almost all of TMProf’s functionality. In short, we aimed to determine if self-aborts from one thread resulted in subsequent commits for the in-flight transactions of the other threads. If we could observe this behavior it would tell us if self-aborts resulted in some form of forward progress. Attempting this required control of several of TMProf’s features.

First, we reduced the playback rate of the execution to no greater than $10^{-3} \times$ the normal execution speed, which allowed us enough time to pause the execution as soon as a specific transaction completed, but before the next transaction appeared. We then began execution playback. When the transaction we were interested in self-aborted, we paused the execution and zoomed-in to ensure it was a self-abort. Once we verified the self-abort, we zoomed out to see all threads, enabling us to see the next transaction’s outcome. We resumed playback, paused it as soon as the next transaction appeared, and then zoomed in to see if the transaction committed.

Using this approach, we were able to determine the exact behavior of the other in-flight transactions once one of them was self-aborted.\footnote{What we found, we believe, was the principle reason behind threadFair’s degraded performance. In short, throughout the execution when one transaction was self-aborted it was generally \textit{not} the case that at least one subsequent transactions committed from the other threads. This means that threadFair was unnecessarily aborting a transaction at commit-time in favor of another in-flight transaction that subsequently did not commit, resulting in no forward progress. We tried to observe the same behavior in the iFair execution, but found that because self-aborts were rare, it did not contain such behavior.} What we found, we believe, was the principle reason behind threadFair’s degraded performance. In short, throughout the execution when one transaction was self-aborted it was generally \textit{not} the case that at least one subsequent transactions committed from the other threads. This means that threadFair was unnecessarily aborting a transaction at commit-time in favor of another in-flight transaction that subsequently did not commit, resulting in no forward progress. We tried to observe the same behavior in the iFair execution, but found that because self-aborts were rare, it did not contain such behavior.

By combining these observations, most especially the latter, we built the iBalanced CM. The essence of iBalanced is to combine the best pieces of threadFair and iFair to create a more efficient CM. From threadFair, we lifted thread-level fairness, because it helps ensure that threads have reduced...
variation in their total execution time, helping eliminate obvious critical paths (see Figure 9). From iFair, we lifted the goal of minimizing self-aborts at commit-time, such as ensuring a transaction with the most reads or writes commits, because such transactions represent forward progress if permitted to commit. However, instead of comparing a committing transaction’s reads or writes against the sum of all other in-flight transactions (iFair’s V), we only require the committing transaction have the largest read or write set, overall, which decreases the likelihood of self-aborts. We also introduced new predicates, not originally part of iFair, that we found by analyzing TMProf’s visualizations, such as when a transaction is in its commit phase and its thread has the fewest commits of any conflicting transaction that it should be allowed to commit regardless of the transactions it may abort. When any of these new predicates are satisfied, they guarantee a transaction will commit.

The iBalanced algorithm is shown in Algorithm 2. WC, RC, FCC, and SC represent constant multipliers for a committing transaction’s write set, read set, fewest commits, and set size, respectively. These multipliers control the importance of each characteristic, where a higher multiplier value denotes more importance and a lower multiplier value, with 1 being the lowest, denotes less. For our experiments, we found WC = 3, RC = 1, FCC = 3, and SC = 3, yield the best general performance. Functions reads() and writes() return the number of elements a transaction has read and written, respectively. The prior() function returns the transaction’s priority, which is incremented each time the transaction is aborted and reset to 1 when it commits.

TMProf’s visualization of the RBtree benchmark (Figure 9) shows iFair outperform threadFair by 61.4%. It also shows iBalanced outperform threadFair by 73.7% and iFair by 5.8%. When using iBalanced in the same benchmark with more threads and larger workloads (including remove operations), as shown in Figure 11, it performs ≈ 10% faster than iFair except for small-sized workloads. Although this margin is small, it demonstrates that, in this case, iBalanced chooses a more optimal set of transactions to abort than iFair, thereby improving performance.

4.5 Linked List and Hash Table

For the linked list benchmark, shown in Figure 12, iBalanced outperform threadFair by upwards of 9.28× and threadFair by upwards of 15.84×. The key reason for iBalanced’s modest performance improvement over the other CMs in this
Algorithm 2 The iBalanced Contention Manager

1: procedure iBal(Tx → commitTx, TransactionSet conflicts)
2:  Transaction c = commitTx, tx = NULL
3:  integer setSize = (writes(c) * (pri0(c) + WC)) + (reads(c) * (prio(c) + RC))
4:  integer abortPrio = 0, abortSetSize = 0, highestPrio = 0
5:  boolean mostReads = true, mostWrites = true
6:  boolean fewestCommits = true, canAbort = false
7:  loop
8:    tx = NextTx(conflicts, tx)
9:    if (tx ≡ NULL) then break
10:   if (commits(tx) < commits(c)) then fewestCommits = false
11:    if (reads(tx) > reads(c)) then mostReads = false
12:    if (writes(tx) > writes(c)) then mostWrites = false
13:    if (prio(tx) > highestPrio) then highestPrio = prio(tx)
14:    abortSetSize += writes(tx) + reads(tx)
15:    abortPrio += prio(tx)
16:    integer p = prio(c)
17:    if (fewestCommits) then
18:      p = p * SCC
19:      setSize = setSize * SC
20:    if (p ≥ highestPrio) then canAbort = true
21:    if (mostReads ∨ mostWrites ∨ fewestCommits) then canAbort = true
22:    if (setSize ≥ abortPrio + abortSetSize) then canAbort = true
23:  return canAbort

For our RBtree, linked list, and hash table experiments, iBalanced outperforms iFair by upwards of 2.05× and threadFair by upwards of 3.27×. iBalanced achieves this performance benefit for the same reason it outperforms the other CMs for linked list benchmark. However, the performance benefits for the hash table benchmark are not as notable, because the initial insert operations on the hash table are, at least initially, highly parallel (i.e., no conflicts). Once the buckets of the hash table reach a certain threshold, however, transactions begin to conflict and iBalanced’s more optimal choice of transactional aborts and commits results in improved performance over iFair.

4.6 Revisiting 1-Writer / N-Readers

For our RBtree, linked list, and hash table experiments, iBalanced is more efficient than iFair and threadFair. However, it is not more efficient than threadFair for the 1-writer / N-readers benchmark, where threadFair outperforms both iFair and iBalanced. This is due to threadFair’s algorithmic design being an excellent match for exploiting the concurrency of the 1-writer / N-readers benchmark. Because of this, we believe threadFair is a realistic CM candidate for certain TM use cases, especially those that have an even, or close to even, distribution of work across all threads.

Even though iBalanced does not perform as well as threadFair for the 1-writer / N-readers benchmark, it does perform as well or better than iFair. InvalSTM’s previously best-performing CM, for all of the benchmarks we analyzed. Furthermore, iBalanced’s efficiency rate over iFair tends to improve on each benchmark as the benchmarks scale to a greater number of threads and larger workloads.

![Figure 12: Linked List Benchmarks.](image-url)
5. CONCLUSION

This paper presented TMProf, a transactional memory (TM) profiler based on three visualization principles that extend the current state of TM profilers and optimizers. These principles are (i) the precise graphical representation of transaction interactions including cross-correlation information and transaction source code, (ii) visualized soft real-time playback of concurrently executing transactions, and (iii) dynamic visualizations of multiple executions.

We described our experience using TMProf and showed how TMProf’s feedback led to the design of two new CMs. We illustrated how these CMs generally led to increased transaction throughput as the threads and size of the workloads increased. Our experimental results showed that iBalanced, one of our newly designed CMs, can increase transaction throughput by nearly 10 x over iFair, InvalSTM’s previously best performing CM, and is the new best-performing CM for InvalSTM for our studied experiments.

The key contribution of this paper was to demonstrate that, due to the complexity of TM, textual representations of such programs are insufficient for profiling, especially because they do not clearly illustrate contention amongst transactions. Instead, we proposed that a TM profiler should be primarily visual, as graphical representation is the best way to convey complex interactions that unfold over time. Furthermore, we argued that such a visualized TM profiler should include execution playback functionality so that the results of transactional conflict mitigation can be observed over the course of the entire execution.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their insightful suggestions. We also sincerely appreciate the feedback offered by Ferad Zyulkyarov, which helped us improve the technical correctness of the paper. Finally, we thank Lori Peek for her numerous edits across several revisions of this paper. Any remaining errors that exist in this paper are our own.

REFERENCES


