Profiling and Optimizing Transactional Memory Applications

Ferad Zyulkyarov[†*], Srdjan Stipic[†*], Tim Harris[‡], Osman S. Unsal[†],
Adrián Cristal[†◇], Ibrahim Hur[†], Mateo Valero[†*]

[†]BSC-Microsoft Research Centre [*]Universitat Politècnica de Catalunya [‡]Microsoft Research
[◇]IIIA - Artificial Intelligence Research Institute CSIC - Spanish National Research Council
[†]{name.surname}@bsc.es, [‡]tharris@microsoft.com

Abstract

Many researchers have developed applications using transactional memory (TM) with the purpose of benchmarking different implementations, and studying whether or not TM is easy to use. However, comparatively little has been done to provide general-purpose tools for profiling and optimizing programs which use transactions.

In this paper we introduce a series of profiling and optimization techniques for TM applications. The profiling techniques are of three types: (*i*) techniques to identify multiple potential conflicts from a single program run, (*ii*) techniques to identify the data structures involved in conflicts by using a symbolic path through the heap, rather than a machine address, and (*iii*) visualization techniques to summarize how threads spend their time and which of their transactions conflict most frequently. Altogether they provide in-depth and comprehensive information about the wasted work caused by aborting transactions. To reduce the contention between transactions we suggest several TM specific optimizations which leverage nested transactions, transaction checkpoints, early release and etc.

To examine the effectiveness of the profiling and optimization techniques, we provide a series of illustrations from the STAMP TM benchmark suite and from the synthetic WormBench workload. First we analyze the performance of TM applications using our profiling techniques and then we apply various optimizations to improve the performance of the Bayes, Labyrinth and Intruder applications.

We discuss the design and implementation of the profiling techniques in the Bartok-STM system. We process data offline or during garbage collection, where possible, in order to minimize the probe effect introduced by profiling.

1 Introduction

Transactional Memory (TM) is a concurrency control mechanism which allows a thread to perform a series of memory accesses as a single atomic operation [22]. This avoids the need for the programmer to design fine-grained concurrency control mechanisms for shared-memory data structures. Typical implementations of TM execute transactions optimistically, detecting any conflicts which occur between concurrent transactions, and aborting one or other of the transactions involved [18].

However, if a program is to perform well, then the programmer needs to understand which transactions are likely to conflict and to adapt their program to minimize this [4]. Several studies report that the initial versions of transactional applications can have very high abort rates [15, 27, 29]—anecdotally, programmers tend to focus on the correctness of the application by defining large transactions without appreciating the performance impact.

Various *ad hoc* techniques have been developed to investigate performance problems caused by TM. These techniques are typically based on adding special kinds of debugging code which execute non-transactionally, even when they are called from inside a transaction. This non-transactional debugging allows a program to record statistics about, for example, the number of times that a given transaction is attempted.

In this paper we describe a series of methodical profiling and optimization techniques which aim to provide a way for a programmer to examine and correct performance problems of transactional applications. We focus, in particular, on performance problems caused by conflicts between transactions: conflicts are a problem for all TM systems, irrespective of whether the TM is implemented in hardware or software, or exactly which conflict detection mechanisms it uses.

We introduce our profiling techniques in Section 2. We follow two main principles. First, we want to report all results to the programmer in terms of constructs present in the source code (e.g., if an object X in the heap is subject to a conflict, then we should describe X in a way that is meaningful to the programmer, rather than simply reporting the object's address). Second, we want to keep the probe effect of using the profiler as low as we can: we do not want to introduce or mask conflicts by enabling or disabling profiling.

We identify three main techniques for profiling TM applications. The first technique identifies multiple conflicts from a single program run and associates each conflict with contextual information. The contextual information is necessary to relate the wasted work to parts of the program as well as constructing the winner and victim relationship between the transactions. The second technique identifies the data structures involved in conflicts, and it associates the contended objects with the different places where conflicting accesses occur. The third technique visualizes the progress of transactions and summarizes which transactions conflict most. This is particularly useful when first trying to understand a transactional workload and to identify the bottlenecks that are present.

Our profiling framework is based on the Bartok-STM system [20] (Section 2.5). Bartok is an ahead-of-time C# compiler which has language-level support for TM. Where possible, the implementation of our profiling techniques aims to combine work with the operation of the C# garbage collector (GC). This helps us reduce the probe effect because the GC already involves synchronization between program threads, and drastically affects the contents of the processors' caches; it therefore masks the additional work added by the profiler. Although we focus on Bartok-STM, we hope that the data collected during profiling is readily available in other TM systems.

We introduce our optimization techniques in Section 3. These techniques can be used after profiling a TM application to improve its performance by reducing the abort rate and wasted work. First, the programmer can try to change the location of the most conflicting write operations by moving them up or down within the scope of the `atomic` block. Depending on the underlying TM system, these changes may have significant impact on the overall performance making the application to scale well or bad (see Figure 16). Second, scheduling mutually conflicting `atomic` blocks to not execute in parallel would reduce the contention but when overused it may introduce new aborts and also serialize transactions. Third, checkpointing the transactions just before the most conflicting statements would reduce the wasted work by re-executing only the invalid part of the transaction. Forth, using pessimistic reads or treating transactional read operations as if they are writes can increase the forward progress in long running read-only transactions. Fifth, excluding memory references from conflict detection would increase the single-threaded performance and decrease aborts substantially. While the last approach might be very effective, applying it is rather subtle because such transformations do not preserve the program correctness.

In Section 4 we present a series of case studies to illustrate the use of our profiling and optimization techniques. We describe how we ported a series of TM programs from C to C#. Initially, three of these applications did not scale well after porting (Bayes, Labyrinth and Intruder from the STAMP suite [5]). Profiling revealed that our version of Bayes had false conflicts due to Bartok-STM's object-level conflict detection. Another performance problem in Bayes was the wasted work caused by the aborts of the longest `atomic` block which is read-only. The remedy for the former problem was to modify the involved data

```
int taskResult = 0;

while (!taskQueue.IsEmpty) {
   atomic {
      Task task = taskQueue.Pop();
      taskResult = task.Execute();
      for (int i < 0; i < n; i++) {
         if (x[i] < taskResult) {
            x[i]++;
         } else if (x[i] > taskResult) {
            x[i]--;
         }
      }
   }
}
```

Figure 1: An example loop that atomically executes a task and updates array elements based on the task's result.

structures and the remedy for the latter problem was to schedule the `atomic` block to not execute together with the `atomic` blocks which cause it to abort. Labyrinth did not scale well because the compiler instrumented calls to the STM library for all memory accesses inside the program's `atomic` blocks. In contrast, the C version performed many of these memory accesses without using the STM library. We were able to achieve good scalability in the C# version by using *early release* to exclude the safe memory accesses from conflict detection. The authors of the STAMP benchmark suite report that Intruder scales well on HTM systems but does not scale well on some STMs. Indeed, initially, Intruder scaled badly on Bartok-STM. However, after replacing a contended red-black tree with a hashtable, and rearranging a series of operations, we achieved scalability comparable to that of HTM implementations. We also showed how to reduce wasted work by using nested `atomic` blocks. In Intruder, wrapping the most conflicting statements in nested `atomic` blocks reduces the wasted work from 45.5% to 36.8% (Table 7 versions Base and Nested Insert). Finally, we verified that our modified version of Intruder continued to scale well on other STMs and HTMs. These results illustrate how achieving scalability across the full range of current TM implementations can be extremely difficult. Aside from these example, the remaining workloads we studied performed well and we found no further opportunities for reducing their conflict rates.

Finally, we discuss related work in Section 5 and conclude in Section 6.

2 Profiling Techniques

As with any other application, factors such as compiler optimizations, the operating system, memory manager, cache size, etc. will have effect on the performance of programs which use TM. However in addition to these factors, performance of transactional applications also depends on (*i*) the performance of the TM system itself (e.g., the efficiency of the data structures that the TM uses for managing the transactions' read-sets and write-sets), and (*ii*) the way in which the program is using transactions (e.g., whether or not there are frequent conflicts between concurrent transactions).

Figure 1 provides a contrived example to illustrate the difference between TM-implementation problems and program-specific problems. The code in the example executes transactional tasks (line 4) and, depending on the task's result, it updates elements of the array `x`. This code would execute slowly in TM systems using naïve implementations of lazy version management: every iteration of the `for` loop would require the TM system to search its write set for the current value of variable `taskResult` (lines 6 and 8). This would be an example of a TM-implementation problem (and, of course, many implementations exist that support lazy version management without naïve searching [18]). On the other hand, if the programmer had placed the `while` loop inside the `atomic` block, then the program's abort rate would increase regardless of the TM implementation. This would be an example of a program-specific problem.

Our paper focuses on program-specific problems. The rationale behind is that reducing conflicts is useful no matter what kind of TM implementation is in use; optimizing the program for a specific TM implementation may give additional performance benefits on that system, but the program might no longer perform as well on other TM systems. Nevertheless, together with the knowledge about the underlying TM implementation the same profiling techniques can give valuable information about the bottlenecks related to the TM-implementation.

In this section we describe our profiling techniques for transactional memory applications. We follow two main principles. First, we report the results at the source code language such as variable names instead of memory addresses or source lines instead of instruction addresses. Results presented in terms of structures in the source code are more meaningful as they convey semantic information relevant to the problem and the algorithm. Second, we want to reduce the probe effect introduced from profiling, and to present results that reflect the program characteristics and are independent from the underlying TM system. For this purpose, we exclude the operation time of the TM system (e.g. roll-back time) from the reported results.

2.1 Conflict Point Discovery

In an earlier paper we introduced a "conflict point discovery" technique that identifies the first program statements involved in a conflict [43]. However, after using this technique to profile applications from

```
increment() {
   counter++;
}

probability(int rate) {
   rnd = random() % 100;
   if (rnd <= rate) {
   atomic {
      increment();
   }
}
```

```
// Thread 1
for (int i < 0; i < 100; i++) {
   probability(80);
   probability(20);
}

// Thread 2
for (int i < 0; i < 100; i++) {
    probability(80);
    probability(20);
}
```

Figure 2: In this example code two threads call functions which increment a shared counter with different probabilities. Basic conflict point discovery will only report that the conflicts happen in `increment`. However, without knowing which function calls `increment` most, the user cannot find and optimize the sequence of function calls where most time is wasted. In this example the important calls would be via `probability(80)` to `increment`.

STAMP, we identified two limitations: (*i*) it does not provide enough contextual information about the conflicts and (*ii*) it accounts only for the first conflict that is found because one or other of the transactions involved is then rolled back. In this paper we refer to our earlier approach as *basic* conflict point discovery.

In small applications and micro-benchmarks most of the execution occurs in one function, or even in just a few lines. For such applications, identifying the statements involved in conflicts would be sufficient to find and understand the TM bottlenecks. However, in larger applications with more complicated control flow, the lack of contextual information means that basic conflict point discovery would only highlight the symptoms of a performance problem without illuminating the underlying causes.

For example, in Figure 2 the two different calls to function `probability` atomically increment a shared counter by calling the function `increment` with a probability of 80% and 20%. When `probability(80)` and `probability(20)` are called in a loop by two different threads, basic conflict point discovery will report that all conflicts happen inside the function `increment`. But this information alone is not sufficient to reduce conflicts because the user would need to distinguish between the different stack back-traces that the conflicts are part of. In this case, the calls involving `probability(80)` should be identified as more problematic than those going through `probability(20)`. Similarly, for other transactional applications, the reasons for the poor performance would most likely be for using, for example, inefficient parallel algorithms, using unnecessarily large `atomic` blocks, or using inappropriate data structures which has low degree of parallelism.

The second disadvantage of basic conflict point discovery is that it only identifies the first conflict that a transaction encounters. It is possible that two transactions might conflict on a series of memory locations and so, if we account for only the first conflict, the profiling results will be incomplete. As a consequence,

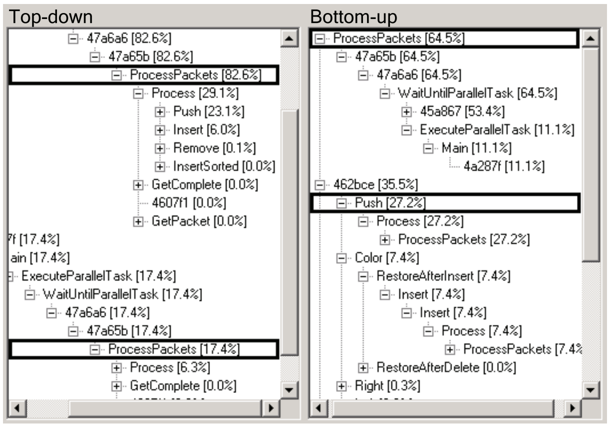
```
   // Thread 1                      // Thread 2
1: atomic {                         atomic {
2:    obj1.x = t1;                     ...
3:    obj2.x = t2;                     ...
4:    obj3.x = t3;                     ...
5:    ...                              obj1.x = t1;
6:    ...                              obj2.x = t2;
7:    ...                              obj3.x = t3;
8: }                                }
```

Figure 3: Basic conflict point discovery would only display the first statements where conflicts happen. On the given examples these statements are line 2 for Thread 1 and line 5 for Thread 2. However, the remaining statements are also conflicting and most likely revealed on the subsequent profiles.

the user will not be able to properly optimize the application and most likely will need to repeat the profiling several times until all the omitted conflicts are revealed. The programmer can end up needing to "chase" a conflict down through their code, needing repeated profile-edit-compile steps. Figure 3 provides an example: basic conflict point discovery would only identify the conflicts on `obj1` (line 2 for Thread 1 and line 5 for Thread 2). However, the remaining statements are also conflicting and most likely will be revealed by subsequent profiles once the user has eliminated the initial conflicting statements. We address the described limitations namely by providing contextual information about the conflicts and accounting for all conflicting memory accesses within aborted transactions.

The contextual information comprises the `atomic` block where the conflict happens and the call stack at the moment when the conflict happens. It is displayed via two views: top-down and bottom-up (Figure 4). In both cases, each node in the tree refers to a function in the source code. However, in the top-down view, a node's path to the root indicates the call-stack when the function was invoked, and a node's children indicate the other functions that it calls. The leaf nodes indicate the functions where conflicts happen. Consequently, a function called from multiple places will have multiple parent nodes. Conversely, in the bottom-up view, a root node indicates a function where a conflict happens and its children nodes indicate its caller functions. Consequently, a function called from multiple places will have multiple child nodes. Furthermore, to help the programmer find the most time-consuming stack traces in the program, each node includes a count of the fraction of wasted work that the node (and its children) are responsible for.

To find all conflicting objects in an aborting transaction, we simply continue checking the remaining read set entries for conflicts. In the rare case, when the other transactions that are involved in a conflict are still running, we force them to abort and re-execute each transaction serially. This way we collect the complete read and write sets of the conflicting transactions. By intersecting the read and write sets, we



```
Top-down
        47a6a6 [82.6%]
          47a65b [82.6%]
            ProcessPackets [82.6%]
              Process [29.1%]
                Push [23.1%]
                Insert [6.0%]
                Remove [0.1%]
                InsertSorted [0.0%]
              GetComplete [0.0%]
              4607f1 [0.0%]
              GetPacket [0.0%]
7f [17.4%]
ain [17.4%]
ExecuteParallelTask [17.4%]
  WaitUntilParallelTask [17.4%]
    47a6a6 [17.4%]
      47a65b [17.4%]
        ProcessPackets [17.4%]
          Process [6.3%]
          GetComplete [0.0%]
```

```
Bottom-up
ProcessPackets [64.5%]
  47a65b [64.5%]
    47a6a6 [64.5%]
      WaitUntilParallelTask [64.5%]
        45a867 [53.4%]
        ExecuteParallelTask [11.1%]
          Main [11.1%]
            4a287f [11.1%]
462bce [35.5%]
  Push [27.2%]
    Process [27.2%]
      ProcessPackets [27.2%]
  Color [7.4%]
    RestoreAfterInsert [7.4%]
      Insert [7.4%]
        Insert [7.4%]
          Process [7.4%]
            ProcessPackets [7.4%
    RestoreAfterDelete [0.0%]
  Right [0.3%]
```

Figure 4: On the left is top-down tree view and on the right bottom-up tree view obtained from the 4-threaded execution of non-optimized Intruder application. The top-down view (left) shows that almost 100% (82.6%+17.4% summed from the two trees) of the total wasted work is accumulated at function `ProcessPackets`. The bottom-up view (right) shows that 64.5% of the total wasted work is attributed to function `ProcessPackets`, and 27.2% to function `Queue.Push` which is called from `ProcessPackets` and the rest to other functions. The non-translated addresses are internal library calls. Because of different execution paths that follow from the main program thread and the worker threads the top-down view draws 2 trees instead of 1.

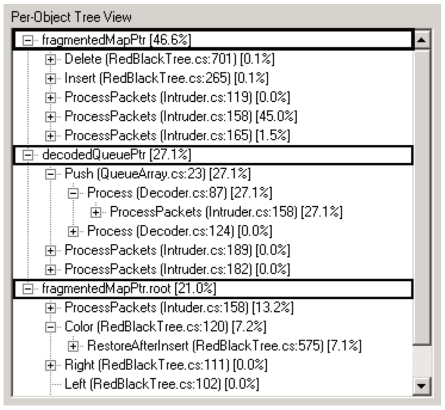
obtain the potentially conflicting objects. Unlike basic conflict point discovery, our approach will report that all statements in the code fragment from Figure 3 are conflicts. Our profiling tool displays the relevant information about the conflicting statements and conflicting objects in the bottom-up view (Figure 4) and the per-object view respectively (Figure 5).

Besides identifying conflicting locations, it is important to determine which of them have the greatest impact on the program's performance. The next section introduces the performance metrics which we use to do this, along with how we compute them.

2.2 Quantifying the Importance of Aborts

The profiling results should draw the user's attention to the `atomic` blocks whose aborts cause the most significant performance impact. As in basic conflict point discovery, a naïve approach to quantify the effect of aborted transactions would only count how many times a given `atomic` block has aborted. In this case results will wrongly suggest that a small `atomic` block which only increments a shared counter and aborts 10 times is more important than a large `atomic` block which performs many complicated computations but aborts 9 times. To properly distinguish between such `atomic` blocks we have used different metric called *WastedWork*. WastedWork counts the time spent in speculative execution which is discarded on abort.

Besides quantifying the amount of lost performance, it is equally important that the profiling results



```
Per-Object Tree View
⊟ fragmentedMapPtr [46.6%]
    ⊞ Delete (RedBlackTree.cs:701) [0.1%]
    ⊞ Insert (RedBlackTree.cs:265) [0.1%]
    ⊞ ProcessPackets (Intruder.cs:119) [0.0%]
    ⊞ ProcessPackets (Intruder.cs:158) [45.0%]
    ⊞ ProcessPackets (Intruder.cs:165) [1.5%]
⊟ decodedQueuePtr [27.1%]
    ⊟ Push (QueueArray.cs:23) [27.1%]
        ⊟ Process (Decoder.cs:87) [27.1%]
            ⊞ ProcessPackets (Intruder.cs:158) [27.1%]
        ⊞ Process (Decoder.cs:124) [0.0%]
    ⊞ ProcessPackets (Intruder.cs:189) [0.0%]
    ⊞ ProcessPackets (Intruder.cs:182) [0.0%]
⊟ fragmentedMapPtr.root [21.0%]
    ⊞ ProcessPackets (Intuder.cs:158) [13.2%]
    ⊟ Color (RedBlackTree.cs:120) [7.2%]
        ⊞ RestoreAfterInsert (RedBlackTree.cs:575) [7.1%]
    ⊞ Right (RedBlackTree.cs:111) [0.0%]
    Left (RedBlackTree.cs:102) [0.0%]
```

Figure 5: Per-object bottom-up abort tree. This view shows the contended objects and the different locations within the program where they have been involved in conflicts. Results shown are obtained from the 4 threaded execution of non-optimized Intruder application. For example, object `fragmentedMapPtr` has been involved in conflict at 5 different places - 3 in function `ProcessPackets`, 1 in `Delete` and 1 in `Insert`. Each object is also cumulatively assigned wasted work. Non-translated addresses are internal library calls.

surface the possible reasons for the aborts. For example, the Bayes application has 15 separate `atomic` blocks, one of which aborts much more frequently than the others (`FindBestInsertTask`). The WastedWork metric will tell us at which `atomic` block the performance is lost, but to reduce the number of aborts the user will also need to find the `atomic` blocks which cause `FindBestInsertTask` to abort. To mitigate this, we have introduced an additional metric *ConflictWin*. ConflictWin counts how many times a given transaction wins a conflict with respect to another transaction which aborts.

Using the information from the WastedWork and ConflictWin metrics, we construct the *aborts graph*; we depict this graphically in Figure 12, although our current tool presents the results as a matrix. The aborts graph summarizes the commit-abort relationship between pairs of `atomic` blocks; it is similar to Chakrabarti's dynamic conflict graphs [8] in helping to link the symptoms of lost performance to their likely causes.

2.3 Identifying Conflicting Data Structures

Atomic blocks abstract the complexity of developing multi-threaded applications. When using `atomic` blocks, the programmer needs to identify the atomicity in the program whereas using locks the programmer should identify the shared data structures and implement atomicity for the operations that manipulate them. However, based on our experience using `atomic` blocks, it is difficult to achieve good performance without understanding the details of the data structures involved [15, 42].

If the programmer wants transactional applications to have good performance it is necessary to know the shared data structures and the operations applied to them. In this case the programmer can use `atomic` blocks in an optimal way by trying to keep their scope as small as possible. For example, as long as the program correctness is preserved, the programmer should use two smaller `atomic` blocks instead of one large `atomic` block or as in Figure 1 put the `atomic` block inside the `while` loop instead of outside. In an earlier paper, we illustrated examples where smaller `atomic` blocks aborted less frequently and incurred less wasted work when they did abort [15, 23, 28].

In addition, the underlying TM system may support language-level primitives to tune performance, or provide an API that the programmer can use to give hints about the shared data structures. For example, Yoo *et al.* [40] used the `tm_waiver` keyword [26] to instruct the compiler to not instrument thread-private data structures with special calls to the STM library. In Haskell-STM [17] the user must explicitly identify which variables are transactional. To reduce the overhead of privatization safety, Spear *et al.* [34] have described a system that lets the programmer explicitly indicate which transactions privatize data [35]. We believe that profiling results can help programmers use these techniques by describing the shared data structures used by transactions, and how conflicts occur when accessing them.

In small workloads which in total have few data structures, the results from conflict point discovery (Section 2.1) would be sufficient to identify the shared data structures. For example, in the STAMP applications, there are usually only a small number of distinct data structures, and it is immediately clear which transaction is accessing which data. However, in larger applications, data structures can be more complex, and can also be created and destroyed dynamically. To handle this kind of workload, our prototype tool provides a tree view that displays the contended objects along with the places where they are subject to conflicts (Figure 5). In the example, the object `fragmentedMapPtr` has been involved in conflicts at 5 different places which have also been called from different functions.

In our profiling framework we have developed an effective and low-overhead method for identifying the conflicting data structures, both static and dynamic. It is straightforward to identify static data structures such as global shared counters: it is sufficient to translate the memory address of the data structure back to a variable. However, it is more difficult when handling dynamically allocated data structures such as an internal node of a linked list; the node's current address in memory is unlikely to be meaningful to the programmer.

For instance, suppose that the `atomic` block in Figure 6 conflicts while executing `list[2]=33` (assigning a new value to the third element in a linked list). To describe the resulting conflict to the programmer,

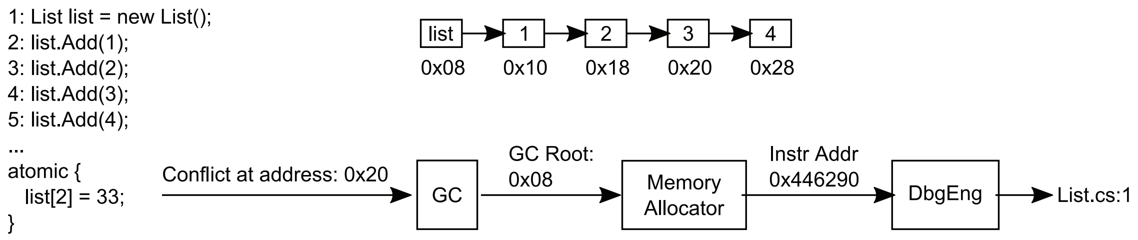


Figure 6: This figure demonstrates our method of identifying conflicting objects on the heap. The code fragment on the left creates a linked list with 4 elements. When the TM system detects a conflict in the `atomic` block, it logs the address of the contended object. During GC, the conflicting address is traced back to the GC root which is the list node. Then the memory allocator is queried at which instruction the memory at address ”0x08” was allocated. At the end, by using the debugger engine the instruction is translated to a source line.

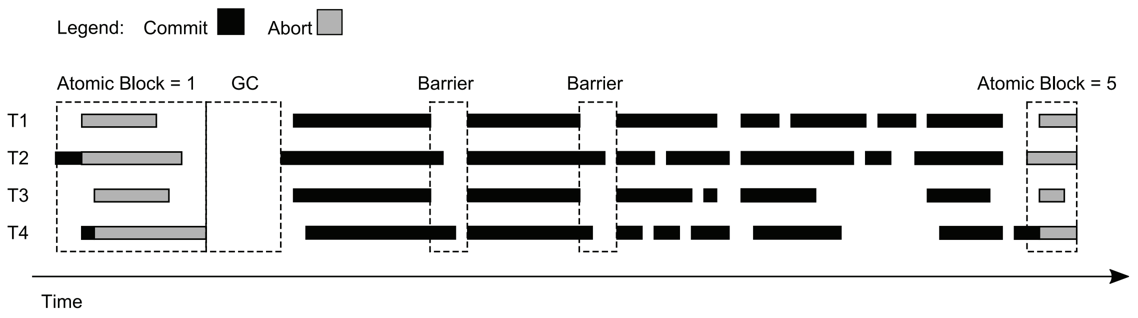


Figure 7: The transaction visualizer plots the execution of Genome with 4 threads. Successfully committed transactions are colored in black and aborted transactions are colored in gray. From this view, we can easily distinguish the different phases of the program execution such as regions with high aborts. By selecting different regions in this view, our tool summarize the profiling data only for the selected part of the execution. To increase the readability of the data, we have redrawn this figure based on a real execution.

we find a path of references to the internal list node from an address that is mapped to a symbol. This approach is similar to the way in which the garbage collector (GC) finds non-garbage objects. Indeed, in our environment, we map the conflicting objects to symbols by finding the GC roots that they are reachable from. If the GC root is a static object then we can immediately translate the address to a variable name. If the GC root is dynamically created, we use the memory allocator to find the instruction at which GC root was allocated and translate the instruction to a source line. To do this, we extended the memory allocator to record allocation instructions (i.e. places where objects are allocated).

2.4 Visualizing Transaction Execution

The next aspect of our profiling system is a tool that plots the execution of all the transactions on a time line (Figure 7). In the view pane the transactions start from the left and progress to the right. Successfully committed transactions are colored black and aborted transactions are colored gray. The places where a color is missing means that no transaction has been running. The view in Figure 7 plots the execution of the Genome application from STAMP. From this view we can easily identify the phases where aborts are most frequent. In this case, most aborts occur during the first phase of the application when repeated gene segments are filtered by inserting them in a hashtable and during the last phase when building the gene sequence.

The transaction visualizer provides a high-level view of the performance. It is particularly useful at the first stage of the performance analysis when the user identifies the hypothetical bottlenecks and then analyzes each hypothesis thoroughly. Another important application of the transaction visualizer is to identify different phases of the program execution (e.g., regions with heavily aborting transactions).

To obtain information at a finer or coarser granularity, the user can respectively zoom in or zoom out. Clicking at a particular point on the black or gray line displays relevant information about the specific transaction that is under the cursor. The information includes: read set size, write set size, `atomic` block id, and if the transaction is gray (i.e., aborted) it displays information about the abort. By selecting a specific region within the view pane, the tool automatically generates and displays summarized statistics only for the selected region.

Existing profilers for transactional applications operate at a fixed granularity [2, 6, 28, 31]. They either summarize the results for the whole execution of the program or display results for the individual execution of `atomic` blocks. Neither of these approaches can identify which part of a program's execution involves the greatest amount of wasted work. But looking at Figure 7 we can easily tell that in Genome transactions abort at the beginning and the end of the program execution.

The statistical information summarized for the complete program execution is too coarse and hides phased executions, whereas per-transaction information is too fine grain and misses conclusive information for the local performance. Obtaining local performance summary is important for optimizing transactional applications because we can focus on the bottlenecks on the critical path and then effectively apply Amdhal's law.

By using the transaction visualizer, the programmer can easily obtain a local performance summary for the profiled application by marking the region that (s)he is interested in. This will automatically generate summary information about the conflicts, transaction read and write set sizes, and other TM characteristics, but only for the selected region. The local performance summary from Figure 7 shows that aborts at the beginning of the program execution happen only in the first `atomic` block and aborts at the end of the program execution happen at the last `atomic` block in program order.

#Threads	Bayes+	Bayes-	Gen+	Gen-	Intrd+	Intrdr-	Labr+	Labr-	Vac+	Vac-	WB+	WB-
2	4.39	4.69	0.09	0.10	3.69	3.51	0.19	0.15	0.80	0.80	0.00	0.00
4	16.29	27.31	0.29	0.50	14.90	13.65	0.35	0.36	2.30	2.45	0.00	0.00
8	53.74	66.08	0.50	0.82	39.64	37.41	0.40	0.47	4.91	5.30	0.02	0.02

Table 1: The abort rate (in %) when the profiling is enabled ("+") and disabled ("-"). Results show that the profiling framework introduces small probe effect by reducing the abort rate for some applications. Results are average of 10 runs. Results for 1 are omitted because there are no conflicts.

#Threads	Bayes+	Bayes-	Gen+	Gen-	Intrd+	Intrdr-	Labr+	Labr-	Vac+	Vac-	WB+	WB-
1	1.59	1.00	1.28	1.00	1.29	1.00	1.07	1.00	1.26	1.00	0.71	1.00
2	1.00	0.56	0.92	0.65	0.97	0.58	0.64	0.61	0.83	0.59	0.60	0.55
4	0.23	0.23	0.91	0.50	0.91	0.36	0.45	0.46	0.58	0.40	0.41	0.33
8	0.21	0.20	0.72	0.50	1.57	0.38	0.72	0.56	0.53	0.34	0.33	0.22

Table 2: Normalized execution time with profiling enabled ("+") and profiling disabled ("-"). Results are average of 10 runs and normalized to the single threaded execution of the respective workload but with profiling disabled.

The global performance summary that our tool generates includes most of the statistics that are already used in the research literature. These are total and averaged results for transaction aborts, read and write set sizes, etc. In addition we build a histogram about the time two or more transactions were executing concurrently. This histogram is particularly useful when diagnosing lack of concurrency in the program. For example, it is possible that a program has very low wasted work but it still does not scale because transactions do not execute concurrently.

2.5 Profiling Framework

We have implemented our profiling framework for the Bartok-STM system [20]. Bartok-STM updates memory locations in-place by logging the original value for rollback in case a conflict occurs. It detects conflicts at object granularity, eagerly for write operations and lazily for read operations. The data collected during profiling is typical for many other TM systems, of course.

The main design principle that we followed when building our profiling framework was to keep the probe effect and overheads as low as possible. We sample runtime data only when a transaction starts, commits or aborts. For every transaction we log the CPU timestamp counter and the read and write set sizes. For aborted transactions we also log the address of the conflicting objects, the instructions where these objects were accessed, the call stack of aborting thread and the `atomic` block id of the transactions that win the conflict. We process the sampled data offline or during garbage collection.

We have evaluated the probe effect and the overhead of our profiling framework on several applications from STAMP and WormBench (Table 1 and Table 2). To quantify the probe effect, we compared the application's overall abort rate when profiling is enabled versus the abort rate when profiling is disabled; a low probe effect is indicated by similar results in these two settings.

Our results suggest that profiling reduces the abort rate seen, but that it does not produce qualitative changes such as masking all aborts. These effects are likely to be due to the additional time spent collecting data reducing the fraction of a thread's execution during which it is vulnerable to conflicts. In addition, logging on abort has the effect of contention reduction because it prevents transactions from being restarted aggressively.

In applications with large numbers of short-running transactions, overheads can be higher since costs incurred on entry/exit to transactions are more significant. Profiling is based on thread-private data collection, and so the profiling framework is not a bottleneck for the applications' scalability.

3 Profile Guided Optimization Techniques

In this section we describe several approaches to optimize transactional memory applications. The goal of these optimization techniques is to reduce the contention between the transactions and also the wasted work incurred on abort. Some of them, for example moving statements, are TM-implementation specific and others such as transaction checkpointing are TM-implementation agnostic. To use the TM-implementation specific optimizations properly, the programmer should know the implementation of the underlying TM system, whether it is eager or lazy versioning, conflict detection etc. Other optimization approaches, such as `atomic` block scheduling are double edged – they improve performance for the cost of transaction serialization. In like manner, early release could be very effective optimization however its use is not safe and should be used with care.

3.1 Moving Statements

Moving statements such as hoisting loop invariants outside of a loop is a pervasive technique that optimizing compilers apply. Similarly, to reduce the cache miss rate, one can decide to pre-fetch data by manually moving a memory reference statement up in the code. Analogous to these examples, TM applications can also perform better by simply moving assignment statements (or statements that update memory) up or down in the code. Figure 16 plots the execution time of the Intruder application from the STAMP [5] benchmark suite using Bartok-STM [20]. In *Beginning* a call to a method which pushes an entry to a queue is moved

```
// Beginning            // End                  // Nested
atomic {                atomic {                atomic {
   counter++               <statement 1>           <statement 1>
   <statement 1>           <statement 2>           <statement 2>
   <statement 2>           <statement 3>           <statement 3>
   <statement 3>           ...                     ...
   ...                     counter++;              atomic {
}                       }                             counter++;
                                                   }
                                                }
        (a)                     (b)                     (c)
```

Figure 8: A code where the increment of the shared counter is: (a) moved up (hoisted) to the beginning of the `atomic` block, (b) moved down to the end of `atomic` block, and (c) wrapped inside a nested `atomic` block.

to the beginning of the `atomic` block, and in *End* the call to the same method is moved to the end of the `atomic` block. Figure 8 is a contrived code example which represent how the code changes in *Beginning* and *End* look like.

The reason for the performance difference lies in the way how memory updates are handled by the TM system. In Bartok-STM, all update operations first lock the object and keep it locked until commit. If the requesting transaction sees that another transaction has already locked the object for update it aborts itself. In STMs like Bartok-STM and TinySTM [14] with encounter time locking, updates at the beginning of an `atomic` block on a highly contended shared variable such as a shared counter (Figure 8 (a)) may have the effect of a global lock. When one transaction successfully locks the object it will keep the lock until commit. In the mean time all the threads that try to execute the same `atomic` block will not be able to acquire the object's lock and will abort. This will serialize the program execution at this point. On the other hand, when the same update operation is at the end of the `atomic` block (see Figure 8 (b)) the transaction will keep the object locked for short time thus allowing other threads to execute the code concurrently until the problematic statement.

Because the approach of improving performance by moving the location of the statements relies on detecting WaW conflicts eagerly, it may not have effect on other TM systems. For example, when executed on the TL2 STM library [10], the location of the same statement affects the performance comparatively much less (see Figure 17). TL2 buffers updates and detects all types of conflicts lazily at commit time.

We can easily identify the statements to move by using conflict point discovery. A statement which updates the memory and causes large wasted work would be a candidate for moving its location. However, the changes that the programmer makes should preserve the program correctness.

3.2 Atomic Block Scheduling

The purpose of transaction scheduling is to reduce the contention for the cost of serialization. There is significant research on how transaction scheduling can be automated but to the best of our knowledge the problem of scheduling `atomic` blocks statically has not been studied.

Dynamic transaction scheduling introduces overhead at runtime because of the additional bookkeeping necessary to decide how to schedule the transactions. Static scheduling does not introduce such overheads. In addition, the scheduling requirements of a transactional application may be simple and not require any adaptive runtime algorithms. For example, Bayes from STAMP TM benchmark suite [5] has 15 `atomic` blocks but almost all the wasted work in the application is caused only by two `atomic` blocks that abort each other. For this case, a decision to statically schedule the two `atomic` blocks to not execute at the same time would be trivial. To decide exactly which `atomic` blocks to schedule, the programmer needs to know the `atomic` block which is responsible for the major part of the wasted work as well as the list of the other `atomic` blocks that it conflicts with. Such information can be obtained through abort graphs [44] (see Figure 12). However, the programmer should be aware that scheduling may not always deliver the expected performance. It is possible that after setting a specific schedule new conflicts appear or the program execution serializes.

3.3 Checkpoints

Various mechanisms have been proposed to implicitly checkpoint transactions at runtime [3, 37]. If a checkpointed transaction aborts, it is rolled back up to the earliest valid checkpoint. Checkpoints can improve the performance of transactional applications because (*i*) the transaction is not re-executed from the beginning and (*ii*) the valid checkpoints are not rolled back. The latter is particularly important for eager versioning (i.e. in-place update) TM systems because rollback operations are expensive. For example, suppose that we checkpoint the code in Figure 8 (b) at line 5. If conflict is detected at line 6 when incrementing the `counter` and the remaining part of the transaction (i.e. lines 1–5) is valid, then only the increment will be rolled back and re-executed.

Techniques to automatically checkpoint transactions exists, but to the best of our knowledge there is no study on statically placing checkpoints. In the ideal case, transactions would re-execute only the code that is not valid. To achieve this, every transactional memory reference should be checkpointed, however

```
// AB1
atomic {
   local_X = X;
   <statement 1>
   ...
   <statement N>
}
```

```
// AB2
atomic {
   X++;
}
```

Figure 9: AB1 is a long running `atomic` block which uses the value X and AB2 is a short running `atomic` block which increments X. If AB1 and AB2 execute concurrently, AB1 will be most of the time aborted by AB2.

this would cause excessive overhead. Therefore, it is necessary to identify where exactly to checkpoint a transaction. Good checkpoint locations are just before the memory references that cause most of the conflicts. We can easily identify these locations by using conflict point discovery [44]. The programmer can manually checkpoint transactions just before the statements that cause most of the conflicts or this can be automated via feedback directed compilation. Similarly to a transaction scheduling (Section 3.2), static checkpointing can be combined with dynamic checkpointing to off-load the runtime for the known conflicts.

Table 6 and Table 7 show the effect of checkpointing an `atomic` block in Intruder. In this experiment we used nested `atomic` blocks as shown in Figure 8 (c) because our STM library did not have checkpointing mechanisms. In this case, if the nested `atomic` block is invalid but the code in the outer block is valid, only the nested `atomic` block will re-execute. In effect this is the same as checkpointing at line 5 in Figure 8 (a).

As we can see, one can implement checkpoints by combining the use of nested `atomic` blocks. Furthermore, unlike checkpoints, nested `atomic` blocks are composable and can be used in functions that are called within other `atomic` blocks or outside `atomic` blocks [19]. A better technique are abstract nested transactions (ANTs) [21]. Unlike checkpoints and nested transactions, the TM system can re-execute ANTs at later point when they are detected to be invalid.

3.4 Pessimistic Reads

To detect conflicts between transactions, the underlying TM implementation needs to know which memory references are accessed for read and for write. High performance STMs are not obstruction-free [13, 16], an implication of such design would allow one transaction be always aborted by another transaction. For example, consider a simple program of two `atomic` blocks AB1 and AB2. Suppose that AB1 is a long running transaction which uses the value of a shared variable X to perform complicated operations and AB2 has only a single instruction which increments X. In this case, AB2 will cause AB1 to abort repeatedly because AB1 will not be able to reach the commit point before AB2 (Figure 9).

To overcome this problem the user may use pessimistic reads or treat read operations as if they are writes. In the first approach it is necessary to update all transactional references to X with the proper pessimistic read operations. Without compiler support, finding all such references manually might be difficult and in some cases impossible. The latter approach is less intrusive because the programmer does not need to update the other references to X. Using pessimistic reads or opening X for write in AB1 from Figure 9 would subsequently cause AB2 to abort and let AB1 to make forward progress. However, this kind of modification, while providing forward progress for AB1, may introduce new aborts.

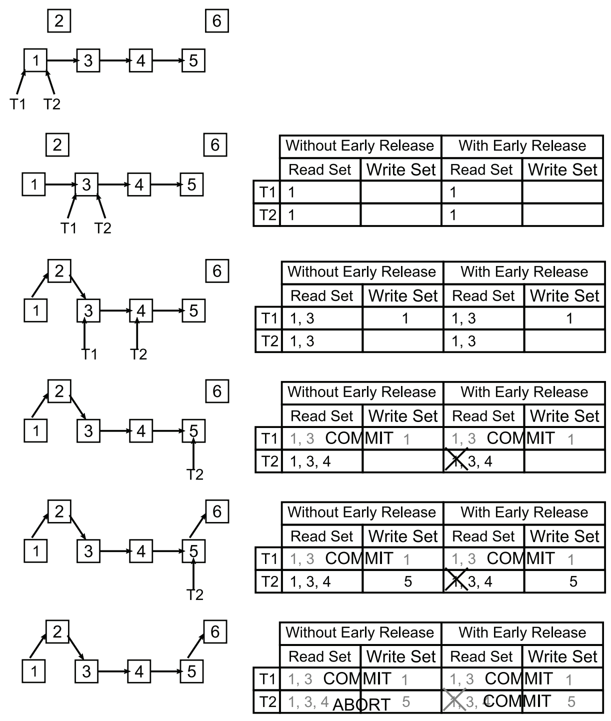
We can find conflicting read operations such as X in AB1 from Figure 9 by looking at the results of conflict point discovery. From these results we can explicitly tell the compiler to open the read operations involved in many conflicts for write.

3.5 Early Release

Early release is a mechanism to exclude entries in the transaction's read set from conflict detection [30, 33]. In certain applications it is possible that the final result of an `atomic` block is still correct although the read set is not valid. For example, consider an `atomic` block which inserts entries in a sorted linked list (Figure 10). Thread T1 wants to insert value 2 and thread T2 wants to insert value 6. To find the right place to insert the new values the two threads iterate over the the list nodes and consequently add them to the transaction's read set. T2 aborts because T1 finishes first and invalidates T2's read set. However, T2 could still correctly insert the node although some entries in its read set are invalid. In this case we can exclude all nodes except 5 from conflict detection.

After carefully studying the Lee's path routing algorithm, Watson *et. al.* [38] have used early release to exclude a major part of the transaction's read set from conflict detection. To achieve similar results, Yoo *et al.* [40] instructed the compiler and Cao Minh *et al.* [5] deliberately skipped inserting calls to the STM library while copying the shared matrix into a thread local variable in Labyrinth. Caching the values of shared variables to a thread local storage, as in Bayes, is another form of excluding the shared variables from conflict detection.

The experience of these studies reports that early release improves the application performance significantly. However, the programmer should not forget that it is not a safe operation (i.e. it can break program correctness). Applying this technique requires prior knowledge about the shard data structures used in the



	Without Early Release		With Early Release	
	Read Set	Write Set	Read Set	Write Set
T1	1		1	
T2	1		1	

	Without Early Release		With Early Release	
	Read Set	Write Set	Read Set	Write Set
T1	1, 3	1	1, 3	1
T2	1, 3		1, 3	

	Without Early Release		With Early Release	
	Read Set	Write Set	Read Set	Write Set
T1	1, 3 COM	MIT 1	1, 3 COM	MIT 1
T2	1, 3, 4		~~1~~, 3, 4	

	Without Early Release		With Early Release	
	Read Set	Write Set	Read Set	Write Set
T1	1, 3 COM	MIT 1	1, 3 COM	MIT 1
T2	1, 3, 4	5	~~1~~, 3, 4	5

	Without Early Release		With Early Release	
	Read Set	Write Set	Read Set	Write Set
T1	1, 3 COM	MIT 1	1, 3 COM	MIT 1
T2	1, 3, 4 ABO	RT 5	~~1~~, 3, 4 COM	MIT 5

Figure 10: Transaction T1 inserts number 2 and transaction T2 inserts number 6 in sorted linked list. Without using early release T2 will abort and when using early release T2 will commit successfully.

algorithm and the operations applied on them – namely wether or not the algorithm can be relaxed. The available profiling tools can help in identifying the shared objects that are involved in conflicts (Section 2.3). Provided with this information, the programmer can focus on the specific objects and try to use early release when possible or use different implementations for the data structures [44].

4 Case Studies

In this section we present a series of case studies of profiling and optimizing the performance of applications from the STAMP TM benchmark suite [5] and from the synthetic WormBench workload [41] by using our techniques. The goal of these case studies is to evaluate the effectiveness of our profiling and optimization techniques: namely wether the profiling techniques reveal the symptoms and causes of the performance lost due to conflicts in these applications and wether our optimization techniques indeed improve the performance of these applications.

To see whether our profiling and optimization techniques can be equally applied across a range of TM implementations we utilize two different STMs – TL2 [10] and Bartok-STM [20]. TL2 buffers speculative

#Threads	BayesNonOpt	BayesOpt	IntrdNonOpt	IntrdOpt	LabrNonOpt	LabrOpt
1	1.00	1.00	1.00	1.00	1.00	1.00
2	0.32	0.56	1.16	0.58	5.25	0.61
4	1.49	0.23	2.92	0.36	30.42	0.46
8	4.81	0.20	n/a	0.38	n/a	0.56

Table 3: The normalized execution time of Bayes, Labyrinth and Intruder before and after optimization. Results are average of 10 runs and the execution time for each applications is normalized to its single threaded execution time. ”n/a” means that the application run longer than 10 minutes and was forced termination.

updates and detects conflicts lazily at commit time for both reads and writes. It operates at word granularity by hashing a memory address to transactional word descriptor. Bartok is an ahead of time C# to x86 compiler with language level support for STM. Bartok-STM updates memory locations in-place by logging the original value for rollback in case a conflict occurs. It detects conflicts at object granularity, eagerly for write operations and lazily for read operations.

For this experiment we have ported several applications from the STAMP suite from C to C#. We did this in a direct manner by annotating the `atomic` blocks using the available language construct that the Bartok compiler supports. In the original STAMP applications, the memory accesses inside `atomic` blocks are made through explicit calls to the STM library, whereas in C# the calls to the STM library are automatically generated by the compiler. WormBench is implemented in the C# programming language.

4.1 Bayes

Bayes implements an algorithm for learning the structure of Bayesian networks from observed data. Initially our C# version of this application scaled poorly (see Table 3). By examining the data structures involved in conflicts, we found that the most heavily contended object is the one used to wrap function arguments in a single object of type `FindBestTaskArg` (Figure 11(a)). Bartok-STM detects conflicts at object granularity, and so concurrent accesses to the different fields of the same object result in false conflicts. The false conflicts caused 98% of the total wasted work. With 2 threads the wasted work constituted about 24% of the program's execution, and with 4 threads it increased to 80%. We optimized the code by removing the wrapper object `FindBestTaskArg` and passing the function arguments directly (see Figure 11(b)). After this small optimization Bayes scaled as expected (Table 3).

From this point we wanted to see wether we can improve the performance of Bayes more. We noticed that out of 15 `atomic` blocks only one, `atomic` block AB12, aborts most and causes 92% of the total wasted work. AB12 calls the method `FindBestInsertTask` and from the per-`atomic` block statistics

```
//Function declaration with wrapper object
Task FindBestInsertTask(FindBestTaskArg argPtr) {
   Learner learnerPtr            = argPtr.learnerPtr;
   Query[] queries               = argPtr.queries;
   ...
}
...
// Preparing a wrapper object
FindBestTaskArg argPtr = new FindBestTaskArg();
argPtr.learnerPtr      = learnerPtr;
argPtr.queries         = queries;
...
// Pass arguments with a wrapper object
FindBestInsertTask(argPtr);
                      (a)

// Function declaration with explicit parameters
Task FindBestInsertTask(
       Learner learnerPtr, Query[] queries, ...)
...
// Passing arguments without a wrapper object
FindBestInsertTask(learnerPtr, queries, ...)
                      (b)
```

Figure 11: Code fragments from Bayes: a) the original code with the wrapper object `FindBestTaskArg`; b) the optimized code with the removed wrapper object and passing the function parameters directly.

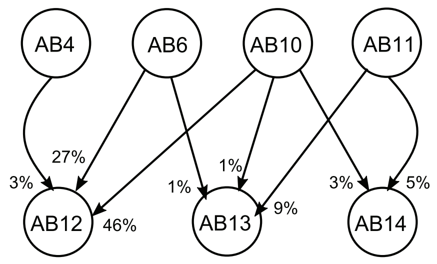


Figure 12: Aborts graph of Bayes when `atomic` blocks AB11 and AB12 are scheduled to not execute in parallel. In this figure AB10 aborts AB12 and the wasted work due to these aborts is 46% from the total program execution. Results are obtained from an execution with 8 threads.

we could see that it is the longest read-only transaction. Aborts graph shows that `atomic` block AB12 is always being aborted by a non-read-only `atomic` block AB11. AB11 is a very short running `atomic` block which updates and caches the shared variables `baseLogLikelihood` and `numTotalParent` into a thread local variable. Based on this profiling information we have decided to statically schedule `atomic` blocks AB11 and AB12 to not execute in parallel. The results in Figure 12 showed to be slightly better but not encouraging because new pairs of aborting `atomic` blocks appeared. Now the aborts dominated between B10 and AB12 constituting 46% of the total wasted work. Despite adding an additional schedule between AB10 and AB12 the execution time did not get better while wasted work was evenly distributed among the non-scheduled `atomic` blocks

Figure 13 is a histogram which shows the time when the execution of two or more transactions are

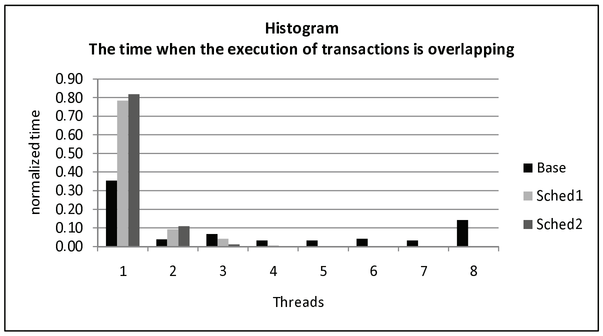


Figure 13: Bayes - this figure shows a histogram of the time when the execution of two or more transactions have overlapped.

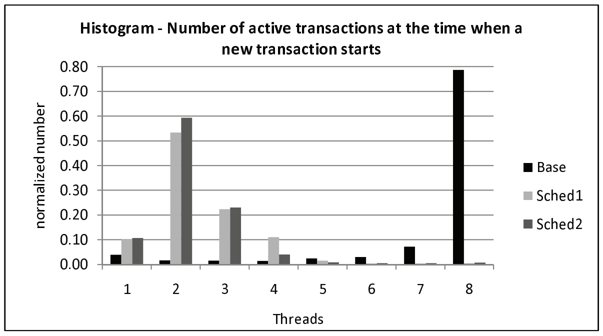


Figure 14: Bayes - this figure shows a histogram of the number of active transactions when a new transaction starts execution.

overlapping and Figure 14 is a histogram which shows the number of active transactions at the moment when a new transaction starts. In the both figures we can see that scheduling `atomic` blocks limits the parallelism – fewer transactions overlap during execution (Figure 13) and there are fewer active transactions at the moment when a new transaction starts (Figure 14). Furthermore, in Figure 13 we can see that in the *Base* version (i.e. with no scheduling) about 35% of the time there is only one transaction executing and 14% of the time there are eight transactions executing in parallel. Considering that 83% of execution in Bayes is spent in transactions [5] the results from the histogram might suggest that the execution of transactions simply do not overlap. However, the actual reason is different. Bayes has few very long running `atomic` blocks and the remaining `atomic` blocks are comparably shorter (e.g. 100x to 10 000x shorter). Most of the time only one thread is executing one of these long transactions and the remaining threads execute the short transactions. This can be confirmed with the results from Figure 14. In the *Base* version 80% of the time when a new transaction stars there are already 7 other transactions running. After we schedule AB11 (i.e. a short transaction) and AB12 (i.e. a 40 000 times longer transaction) to not execute in parallel the number of active transactions drops significantly.

#Threads	TCC-Orig	TCC-Opt	Eazy-Orig	Eazy-Opt	TL2-Orig	TL2-Opt
1	1.00	1.01	1.00	0.96	1.00	0.80
2	0.73	0.67	0.61	0.59	0.92	0.60
4	0.51	0.43	0.37	0.35	0.63	0.48
8	0.39	0.31	0.26	0.22	0.65	0.52

Table 4: Execution time of Intruder before and after optimization on Scalable-TCC, Eazy-HTM and TL2. Results are average of 10 runs and normalized to the single threaded original version of Intruder.

In Table 2 the non-optimized version of Bayes scales superlinearly from 1 to 2 threads. This phenomena happens because the algorithm for learning the structures is relaxed by using a cached version of two shared variables. The subsequent operations may operate on outdated values and cause the learning process to be shorter or longer. In our case, for the suggested input the learning process was shorter.

4.2 Intruder

Intruder implements a network intrusion detection algorithm that scans network packets and matches them against a dictionary of known signatures. The authors of STAMP report that this application scales well on HTM systems but does not scale well on STMs [5]. Therefore understanding and eliminating the bottlenecks of this application was a challenge for us.

Our profiling techniques showed that the most contended objects in Intruder are `fragmentedMapPtr` and `decodedQueuePtr`. In 4-threaded execution, aborts in which `fragmentedMapPtr` was involved caused 67.6% wasted work and aborts in which `decodedQueuePtr` was involved caused 27.1% of wasted work. The wasted work of the both objects constituted 92.7% of the total program execution. The `fragmentedMapPtr` object is a map data structure used to reassemble the fragmented packets. Its implementation is based on red black tree and most important conflicts were happening during lookup. On the other hand, the lookup was invoked while adding a new entry to check if it already exists. Our approach of resolving the bottleneck at `fragmentedMapPtr` was to replace the underlying implementation with a chained hashtable. Unlike red black tree, when using hashtable transactions access fewer objects (i.e. their read set is smaller) and consequently have lower probability of conflict. We have experimentally verified that using hashtable instead of red black tree improves the application performance across different STM and HTM implementations (see Table 4). For this experiment we used state-of-the-art HTM systems (Scalable-TCC [7] and Eazy-HTM [36]) in a simulated environment.

Although we achieved satisfiable scalability for Intruder we continued to examine its performance in more depth. Intruder has in total three `atomic` blocks and our per-`atomic` block profiling showed that

#Threads	AB1	AB2	AB3
1	0.00%	0.00%	0.00%
2	5.48%	91.01%	4.51%
4	3.38%	94.90%	1.72%
8	5.45%	93.43%	1.12%

Table 5: The wasted work caused by the aborts of the different `atomic` blocks in Intruder. Results are normalized.

only one of them causes significant wasted work (Table 5). The subject `atomic` block contains only a call to method `Decoder.Process` (see Figure 15). We used our profiling tool to see exactly which statements from this `atomic` block are involved in conflicts. The results of conflict point discovery are shown in Table 6 (version Base).

Most of the conflicts in our system are read-after-write (RaW) or write-after-read (WaR) type and therefore detected at commit time (line 39). When the number of threads is low, significant amount of wasted work is caused due to conflicts at the statement which calls method `decodedQueuePtr.Push` (line 31). `decodedQueuePtr` data structure maintains the list of the packets which are assembled from several segments. Conflicts at this statement are of write-after-write (WaW) type which Bartok-STM detects eagerly. When the number of threads increases, the wasted work at the call to method `fragmentedListPtr.InsertSorted` becomes dominant. `fragmentedListPtr` is a helper data structure (sorted list) used to assemble a packet from several segments. Conflicts at the call to `InsertSorted` are also WaW. Contention at this point increases with the number of threads because the probability of multiple threads inserting different segments belonging to the same packet increases.

We tried to reduce wasted work by moving the call to `Push` from the end of the `atomic` block (line 31) to the beginning of the `atomic` block (line 8). We anticipated that detecting conflicts earlier and aborting transactions earlier would generate less wasted work – speculative execution and state to rollback. However, opposite to our expectations the performance of the application degraded (see Figure 16). The conflict point analysis for the modified version showed that the poor performance is due to the increase in the number of re-executions and the abort rate of the `atomic` block (Table 6 version Push Move Up).

The reason for the increase in the number of re-executions and consequently the abort rate is specific to the implementation of Bartok-STM. When threads are about to update the `decodedQueuePtr` object, the TM system first locks the object. In this case when one thread successfully acquires object's lock all the other threads fail and abort until the lock is released during commit. In fact, the updates on `decodedQueuePtr` have the same effect as if it is a global lock. When the update is at the end of the `atomic` block (line

```
public Error Process(Packet packetPtr) {
...
   if (numFragment > 1) {
      ...
      if (fragmentedListPtr == null) {
         ...
      } else {
         ...
         fragmentedListPtr.InsertSorted(packetPtr);
         if (fragmentedListPtr.GetSize() == numFragment) {
            int i, numByte = 0;
            foreach (Packet fragmentPtr in fragmentedListPtr) {
               if (fragmentPtr.FragmentId != i) {
                  fragmentedMapPtr.Remove(flowId);
                  return Error.ERROR_INCOMPLETE;
               }
               numByte += fragmentPtr.Length;
               i++;
            }

            char[] data = new char[numByte];
            int dst = 0;
            foreach (Packet fragmentPtr in fragmentedListPtr){
               Array.Copy(fragmentPtr.Data, data, dst);
               dst += fragmentPtr.Length;
            }
            Decoded decodedPtr = new Decoded();
            decodedPtr.flowId = flowId;
            decodedPtr.data = data;

            decodedQueuePtr.Push(decodedPtr);
            fragmentedMapPtr.Remove(flowId);
         }
      }
   } else {
      ...
   } // end of if (numFragment > 1)
   return Error.ERROR_NONE;
}
```

Figure 15: Code fragment from Intruder. Method `Decoder.Process` is called inside an `atomic` block. Because of space constraints some irrelevant code such as initializations are omitted.

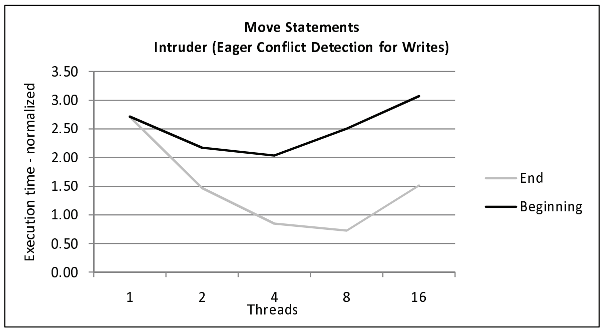


Figure 16: This figure shows the effect of changing the location of only one statement inside an `atomic` block on typical STM systems which detect Write-After-Write conflicts eagerly. At *Beginning* an update operation is near the beginning of an `atomic` block and at *End* the update operation is near the end of the `atomic` block.

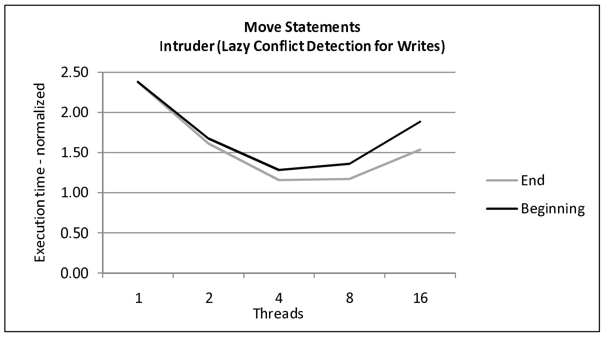


Figure 17: This figure shows the effect of changing the location of only one statement inside an `atomic` block on typical STM systems which detect Write-After-Write conflicts lazily. At *Beginning* an update operation is near the beginning of an `atomic` block and at *End* the update operation is near the end of the `atomic` block.

31) threads can execute large part of the `atomic` block concurrently, but when it is at the beginning of the `atomic` block (line 8) threads serialize trying to acquire the lock for `decodedQueuePtr`. The serialized execution is also confirmed by reading the histogram of the time when transactions are executed concurrently. However, on TM systems that detect WaW conflicts lazily at commit time such code changes do not have significant effect. We have performed the same experiment using TL2. In this case the performance of Intruder is similar in both cases (see Figure 17).

As discussed in Section 3.3, the high abort rate at the statements which call `Push` and `InsertSorted` suggests that using checkpoints or nested `atomic` blocks would improve the performance. We have carried three different experiments: (*i*) we have wrapped the call to `Push` in a nested `atomic` block (Table 6 version Nested Push), (*ii*) we have wrapped the call to `InsertSorted` in a nested `atomic` block (Table 6 version Nested Insert) and (*iii*) we have wrapped both calls in nested `atomic` blocks (Table 6 version Nested Ins. + Push). We have extended Bartok-STM to support partial roll back for nested transactions i.e. if the outer transaction is valid, only the nested transaction will re-execute.

#Threads	Version	InsertSorted	Push	Commit	Abort	#Re-execute	Wasted Work
2	Base	2.94%	48.06%	49.00%	1.88%	0.02	2.28%
	Push Move Up	0.00%	100.00%	0.00%	58.48%	1.43	40.16%
	Nested Push	11.77%	9.22%	79.01%	1.42%	0.01	1.14%
	Nested Insert	60.80%	38.79%	0.37%	1.38%	0.01	2.02%
	Nested Ins. + Push	98.54%	0.70%	0.76%	1.38%	0.01	1.36%
4	Base	19.16%	23.41%	57.42%	13.78%	0.16	14.74%
	Push Move Up	0.0%	100.00%	0.0%	70.60%	2.41	63.30%
	Nested Push	28.26%	2.01%	69.73%	9.50%	0.11	10.60%
	Nested Insert	88.15%	10.33%	1.52%	18.48%	0.27	14.36%
	Nested Ins. + Push	97.53%	0.08%	2.39%	8.27%	0.09	11.30%
8	Base	38.38%	13.31%	48.31%	36.16%	0.57	40.84%
	Push Move Up	0.00%	100.00%	0.0%	77.10%	3.38	83.45%
	Nested Push	44.13%	0.11%	55.76%	28.46%	0.40	42.02%
	Nested Insert	90.32%	1.50%	8.18%	13.45%	0.16	23.83%
	Nested Ins. + Push	99.05%	0.04%	0.91%	25.40%	0.34	39.60%

Table 6: The transactional characteristics of the `atomic` block which executes function `Decoder.Process` from Figure 15. *InsertSorted*, *Push* and *Commit* indicate the wasted work caused by the conflicts detected respectively at the calls to methods `InsertSorted` (line 9), `Push` (line 31) and when transaction commits (line 39). *Abort* indicates the abort rate of this `atomic` block. *#Re-execute* indicates the number of consecutive re-executions when abort happens. *Wasted Work* indicates the part of this `atomic` block execution which was wasted because of aborts.

#Thrd	Version	Norm. Time	Abort	WW
2	Base	0.54	2.38%	3.20%
	Push Move Up	0.80	28.35%	27.34%
	Nested Push	0.54	3.14%	2.66%
	Nested Insert	0.54	2.76%	3.30%
	Nested Ins. + Push	0.54	3.08%	2.98%
4	Base	0.31	11.52%	17.56%
	Push Move Up	0.75	72.22%	77.90%
	Nested Push	0.30	12.64%	16.10%
	Nested Insert	0.31	10.98%	18.48%
	Nested Ins. + Push	0.32	9.93%	15.77%
8	Base	0.27	32.12%	45.50%
	Push Move Up	0.92	90.10%	96.03%
	Nested Push	0.28	33.40%	53.48%
	Nested Insert	0.25	26.45%	36.80%
	Nested Ins. + Push	0.30	29.78%	47.38%

Table 7: Transactional characteristics of Intruder summarized for the whole program execution. *Norm. Time* is the normalized execution time of each version to its single threaded execution, *Abort* is the abort rate, *WW* is the wasted work caused by aborts.

From conflict point discovery we can see that invoking `Push` inside a nested transaction reduces the wasted work and improves the performance of the outer `atomic` block (Table 6 version Nested Push). The nested `atomic` saves time by preventing the outer transaction from rollback and re-execution when it is valid. This modification has also changed the balance over the sources of wasted work by shifting some of the wasted work to `InsertSorted` and `Commit`. When only `InsertSorted` is wrapped in a nested `atomic` block we can see that the wasted work at the call to `InsertSorted` increases with the same amount at which conflicts on `Commit` decrease. This suggests that besides the WaW conflicts, there are also RaW and WaR conflicts which are detected at the end of the commit. When using nested transactions, most of these conflicts are detected when the nested transaction commits, otherwise the same conflicts are detected when the outer transaction commits. In other words, the nested `atomic` block changes the conflict detection to an earlier point during the execution of the outer `atomic` block (i.e. the end of the nested `atomic` block). In effect, this reduces the amount of speculative execution due to conflicts which otherwise would be discovered at the end of the outer `atomic` block. Using nested `atomic` blocks at both places subsumes the observed results from conflict point discovery (Table 6 version Nested Ins. + Push).

Table 7 shows the summarized results over the whole program execution for the different versions of Intruder. These results suggest that the best performance for 4 threads is achieved when `Push` is called inside a nested `atomic` and for 8 threads when `InsertSorted` is called inside nested `atomic` block. Despite the lower wasted work the execution time of Intruder is not significantly better than the base version. The reason is that nested `atomic` blocks incur small runtime overhead which is not always amortized by the saved wasted work.

Early release, which is demonstrated in the following section, is another technique that can squiz a bit more performance from Intruder. As described in Figure 10, it is possible to use early release when packet segments are inserted in sorted order in `fragmentedListPtr` (Figure 15 line 9).

Last but not least, we would like to note that the authors of STAMP have designed this benchmark suite with the purpose to benchmark the performance of different TM implementations. Therefore, to benchmark broad spectrum of implementations it is not necessary that applications in this suite are implemented in the most optimal way and expected to scale. In fact, Intruder is a very useful workload because it illustrates how an application's behavior can be dependent on the TM system that it uses. We also believe that STAMP authors were aware that using hashtable instead of red black tree would make the application more scalable for STMs.

#Threads	Application	Abort	Wasted Work
2	Genome	0.10%	0.10%
	Vacation	0.80%	1.20%
	WormBench	0.00%	0.00%
4	Genome	0.50%	0.20%
	Vacation	2.45%	4.80%
	WormBench	0.01%	0.02%
8	Genome	0.82%	0.50%
	Vacation	5.30%	7.90%
	WormBench	0.03%	0.07%

Table 8: Percentage of the wasted work due to aborts in Genome, Vacation and WormBench.

4.3 Labyrinth

Labyrinth implements a variant of Lee's path routing algorithm used in drawing circuit blueprints. The only data structure causing conflicts in this application was the grid on which the paths are routed. Almost all conflicts were happening in the method that copies the shared grid into a thread local memory. The wasted work due to the aborts at this place amounted to 80% of the total program execution with 2 threads and 98% with 4 threads. In this case we followed a well known optimization strategy described by Watson *et al.* [38]. The optimization is based on domain specific knowledge that the program still produces correct result even if threads operate on an outdated copy of the grid. Therefore, we annotated the `grid_copy` method to instruct the compiler to not instrument the memory accesses inside `grid_copy` with calls to the STM library, which in fact is functionally the same as using early release. After this optimization Labyrinth's execution was similar to the one reported by the STAMP suite's authors [5] (see Table 3).

Although our prior knowledge of the existing optimization technique, this use case serves as a good example when TM applications can be optimized by giving hints to the TM system in similar way as with early release.

4.4 Genome, Vacation, WormBench

Genome, Vacation and WormBench scaled as reported by their respective authors and had very little wasted work (see Table 8). In these applications, there was not any opportunity for further optimizations.

In Vacation we saw that the most aborting `atomic` block encloses a while loop. We were tempted to move the `atomic` block inside the loop as in Figure 1 but that would change the specification of the application that the user can specify the number of the tasks to be executed atomically. Moving the `atomic` block inside the loop would always execute one task and therefore reduce the conflict rate but the user will no longer be able to specify the number of the tasks that should execute atomically. Also, similar changes may not always preserve the correctness of the program because they may introduce atomicity violation errors. In Genome, though very few, aborts occurred in the first and the last `atomic` blocks in the program order (see Figure 7). In our setup, WormBench had almost not conflicts — in 8-threaded execution from 400 000 transactions only about 1100 aborted.

5 Related Work

Chafi *et al.* developed the Transactional Application Profiling Environment (TAPE) which is a profiling framework for HTMs [6]. The raw results that TAPE produces can be used as input for the profiling techniques that we have proposed. This would enable profiling transactional applications that execute on HTMs or HyTMs.

In a similar manner, the Rock processor provides a status register to understand why transactions abort [9] (reflecting conflicts between transactions, and aborts due to practical limits in the Rock TM system). Examples include transactions being aborted due to a buffer overflow or a cache line eviction. Profiling applications in this way is complementary to our work which will allow users to further optimize their code for certain TM system implementations.

Concurrent with our own work, Chakrabarti [8] introduced dynamic conflict graphs (DCG). A coarse grain DCG represents the abort relationship between the `atomic` blocks similar to aborts graph (see Figure 12). A fine grain DCG represents the conflict relationship between the conflicting memory references. To identify the conflicting memory references, Chakrabarti proposed a technique similar to basic conflict point discovery [43]. Our new extensions over basic conflict point discovery (Section 2.1) would generate more complete DCGs. The more detailed fine grain DCGs would complement the profiling information by linking the symptoms of lost performance to the reasons at finer statement granularity. In addition, identifying conflicting objects is another feature which relates the different program statements where conflicts happen with the same object and vice versa.

Independently from us, Lourenço *et al.* [24] have developed a tool for visualizing transactions similar to the transaction visualizer that we describe in Section 2.4. They also summarize the common transactional characteristics that are reported in the existing literature such as abort rate, read and write set, etc. over the whole program execution. Our work complements theirs by reporting results in source language such as variable names instead of machine addresses. Also, we provide local summary which is helpful for examining the performance of specific part of the program execution.

In an earlier paper we profiled Haskell-STM applications using per-`atomic` block statistics [31]. We extend this work by providing mechanisms to obtain statistics at various granularity, including per-transaction, per-`atomic` block, local and global summary. In addition, our statistics include contextual information comprising the function call stack which is displayed via the top-down and bottom-up views. The contextual information helps relating the conflicts to the many control flows in large applications where `atomic` blocks can be executed from various functions and where `atomic` blocks include library calls. In the same work we also explored the common statistical data used in the research literature to describe the transactional characteristics of the TM applications: time spent in transactions, read set, write set, abort rate, etc. In addition to these results we generate a histogram about how much of the transactions' execution interleave. This information is particularly useful to see the amount of parallelism in the program and find cases when a program does not abort but also does not scale.

Adl-Tabatabai *et al.* [1] and Harris *et al.* [20] have described and implemented transactional memory optimizations in compilers with language level support of software transactional memory. Some of these leverage existing compiler optimizations such as loop transformations or common subexpression elimination on transactional code. Others are transactional memory specific and target detecting and eliminating redundant calls to the STM library such as repeated logging of the same object. For, example when the compiler sees that an object is first read and then updated, then the compiler can skip instrumenting `OpenForRead` and instrument only one `OpenForWrite` call for both operations. This can be seen similar to using pessimistic reads (Section 3.4) however pessimistic reads can be used also for objects that are only read but not updated. Our optimization techniques are complementary and can be applied on a code which is automatically optimized by the compiler. Unlike automatic compiler optimizations, our techniques rely on prior profiling information about the program execution and the underlying TM implementation.

To reduce aborts, Sonmez *et al.* [32] have interchangeably used pessimistic and optimistic reads in the Haskell runtime. Whenever an object becomes highly congested it uses pessimistic reads and whenever it becomes less congested it switched back to optimistic reads. Identifying conflicting objects at runtime and switching between optimistic and pessimistic logging comes with additional overhead. Using conflict point discovery, the programmer can easily identify the always conflicting objects and by using local transactional summaries the programmer can see the phases when an object is contended and when not. In such case the programmer can statically specify whether to open an object for read pessimistically and when to switch between pessimistic and optimistic reads. Static decisions can be used to exclude objects from dynamic decisions. This would reduce the runtime overhead of identifying conflicting objects and switching between two logging approaches. On the other side, dynamic decisions would increase the parallelism by switching between pessimistic and optimistic logging earlier than the static specification.

Several researchers have examined various methods for scheduling transactions dynamically [11, 12, 25, 39]. Typically transactions are continuously monitored how frequently they abort. Whenever the abort rate exceeds a certain threshold transactions are serialized to reduce contention. Other approaches go step further by keeping history of the read and write sets of the transactions and try to predict weather two `atomic` blocks will conflict if they are executed concurrently. When possible the TM system may schedule two `atomic` blocks that are likely to conflict to execute on the same core. Unlike, dynamic scheduling, static scheduling cannot be flexible and adapt to the changing behavior of transactions. However, static scheduling does not have runtime overheads and might perform better in cases when the transactional characteristics of `atomic` blocks are constant. In addition, these two approaches can be combined to complement each others deficiencies – static scheduling can be used for the `atomic` blocks with predictive behavior and dynamic scheduling for those with non-predictive behavior.

6 Conclusion

In this paper we have introduced new techniques for profiling and optimizing transactional applications. The goal of our work is to provide the programmers means for discovering and resolving the TM bottlenecks in their applications. Our profiling techniques produce comprehensive information about transactions' aborts, wasted work and conflicts. The detailed profiling information can be subsequently used to optimize the transaction execution in a way to reduce the conflicts.

To examine the effectiveness of the proposed techniques we have profiled applications from STAMP TM benchmark suite and WormBench. Based on the profiling results we could successfully optimize Bayes, Intruder and Labyrinth.

Acknowledgments

We would like to thank Adria Armejach for running the Intruder experiments on the simulator, Timothy Hayes, Torvald Riegel, Vesna Smiljkovic, Nehir Sonmez for their comments and feedback.

This work is supported by the cooperation agreement between the Barcelona Supercomputing Center – National Supercomputer Facility and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2007-60625, by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC) and by the European Commission FP7 project VELOX (216852). Ferad Zyulkyarov is also supported by a scholarship from the Government of Catalonia.

References

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proc. 2006 ACM SIGPLAN conference on programming language design and implementation*, pages 26–37, June 2006.

[2] M. Ansari, K. Jarvis, C. Kotselidis, M. Lujan, C. Kirkham, and I. Watson. Profiling transactional memory applications. In *PDP '09: Proc. 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 11–20, 2009.

[3] C. Blundell, A. Raghavan, and M. M. Martin. Retcon: transactional repair without replay. In *ISCA '10: Proc. 37th International Symposium on Computer Architecture*, ISCA '10, pages 258–269, June 2010.

[4] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. In *ISCA '07: Proc. 34th International Symposium on Computer Architecture*, pages 81–91, June 2007.

[5] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proc. 11th IEEE International Symposium on Workload Characterization*, pages 35–46, September 2008.

[6] H. Chafi, C. Cao Minh, A. McDonald, B. D. Carlstrom, J. Chung, L. Hammond, C. Kozyrakis, and K. Olukotun. TAPE: A transactional application profiling environment. In *ICS '05: Proc. 19th International Conference on Supercomputing*, pages 199–208, June 2005.

[7] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. Cao Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A scalable, non-blocking approach to transactional memory. In *HPCA '07: Proc. 13th IEEE International Symposium on High Performance Computer Architecture*, pages 97–108, Feb. 2007.

[8] D. R. Chakrabarti. New abstractions for effective performance analysis of STM programs. In *PPoPP '10: Proc. 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 333–334, 2010.

[9] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS '09: Proc. 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–168, 2009.

[10] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC '06: Proc. 20th ACM International Symposium on Distributed Computing*, pages 194–208, Sept. 2006.

[11] S. Dolev, D. Hendler, and A. Suissa. Car-stm: scheduling-based collision avoidance and resolution for software transactional memory. In *PODC '08: Proc. of 27th ACM Symposium on Principles of Distributed Computing*, pages 125–134, August 2008.

[12] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh. Preventing versus curing: avoiding conflicts in transactional memories. In *PODC '09: Proc. of the 28th ACM symposium on Principles of Distributed Computing*, pages 7–16, 2009.

[13] R. Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR–06–052, Intel, 2006.

[14] P. Felber, C. Fetzer, U. Mller, T. Riegel, M. Skraut, and H. Sturzrehm. Transactifying applications using an open compiler framework. In *TRANSACT '07: 2nd workshop on transactional computing*, 2007.

[15] V. Gajinov, F. Zyulkyarov, A. Cristal, O. S. Unsal, E. Ayguadé, T. Harris, and M. Valero. QuakeTM: Parallelizing a complex serial application using transactional memory. In *ICS '09: Proc. 23rd International Conference on Supercomputing*, pages 126–135, June 2009.

[16] R. Guerraoui and M. Kapalka. On obstruction-free transactions. In *SPAA '08: Proc. of the 20th Symposium on Parallelism in Algorithms and Architectures*, pages 304–313, June 2008.

[17] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proc. 18th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 388–402, Oct. 2003.

[18] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. 2nd edition, 2010.

[19] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proc. 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48 – 60, Feb 2005.

[20] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI '06: Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 14–25, June 2006.

[21] T. Harris and S. Stipic. Abstract nested transactions. In *TRANSACT' 07: 2nd workshop on transactional computing*, August 2007.

[22] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93: Proc. 20th International Symposium on Computer Architecture*, pages 289–300, May 1993.

[23] G. Kestor, S. Stipic, O. S. Unsal, A. Cristal, and M. Valero. RMS-TM: A transactional memory benchmark for recognition, mining and synthesis applications. In *TRANSACT' 09: 4th Workshop on Transactional Computing*, Feb. 2009.

[24] J. Lourenço, R. Dias, J. Luís, M. Rebelo, and V. Pessanha. Understanding the behavior of transactional memory applications. In *PADTAD '09: Proc. 7th Workshop on Parallel and Distributed Systems*, pages 1–9, 2009.

[25] W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J. L. Lawall, and G. Muller. Scheduling support for transactional memory contention management. In *PPoPP '10: Proce. of the 15th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 79–90, 2010.

[26] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for C/C++. In *OOPSLA '08: Proc. 23rd ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 195–212, Oct. 2008.

[27] V. Pankratius, A.-R. Adl-Tabatabai, and F. Otto. Does transactional memory keep its promises? Results from an empirical study. Technical Report 2009-12, University of Karlsruhe, Sept. 2009.

[28] C. Perfumo, N. Sonmez, S. Stipic, A. Cristal, O. S. Unsal, T. Harris, and M. Valero. The limits of software transactional memory (STM): Dissecting Haskell STM applications on a many-core environment. In *CF '08: Proc. 5th International Conference on Computing Frontiers*, pages 67–78, May 2008.

[29] C. J. Rossbach, O. S. Hofmann, and E. Witchel. Is transactional programming actually easier? In *PPoPP '10: Proc. 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Jan. 2010.

[30] T. Skare and C. Kozyrakis. Early release: Friend or foe? In *WTMW '06: In Workshop of Transactional Memory Workloads*, 2006.

[31] N. Sonmez, A. Cristal, O. S. Unsal, T. Harris, and M. Valero. Profiling transactional memory applications on an atomic block basis: A haskell case study. In *MULTIPROG '09: Proc. 2nd Workshop on Programmability Issues for Multi-core Computers*, Jan. 2009.

[32] N. Sonmez, T. Harris, A. Cristal, O. Unsal, and M. Valero. Taking the heat off transactions: Dynamic selection of pessimistic concurrency control. In *IPDPS '09: Proc. 23rd IEEE International Symposium on Parallel and Distributed Processing*, pages 1–10, May 2009.

[33] N. Sonmez, C. Perfumo, S. Stipic, A. Cristal, O. S. Unsal, and M. Valero. Unreadtvar: Extending haskell software transactional memory for performance. In *TFP '07: In Proc. 8th Symposium on Trends in Functional Programming*, 2007.

[34] M. F. Spear, L. Dalessandro, V. Marathe, and M. L. Scott. Ordering-based semantics for software transactional memory. In *OPODIS '08: Proc. 12th International Conference on Principles of Distributed Systems*, pages 275–294, Dec. 2008.

[35] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. In *PODC'07: Proc. 26th ACM Symposium on Principles of Distributed Computing*, pages 338–339. 2007.

[36] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: eager-lazy hardware transactional memory. In *MICRO 42: Proc. 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 145–155, 2009.

[37] M. M. Waliullah and P. Stenstrom. Intermediate checkpointing with conflicting access prediction in transactional memory systems. Technical report, Chalmers University of Technology.

[38] I. Watson, C. Kirkham, and M. Lujan. A study of a transactional parallel routing algorithm. In *PACT '07: Proc. 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 388–398, 2007.

[39] R. M. Yoo and H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In *SPAA '08: Proc. of 20th Symposium on Parallelism in Algorithms and Architectures*, pages 169–178, 2008.

[40] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. S. Lee. Kicking the tires of software transactional memory: Why the going gets tough. In *SPAA '08: Proc. 20th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 265–274, June 2008.

[41] F. Zyulkyarov, S. Cvijic, O. S. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero. WormBench: A configurable workload for evaluating transactional memory systems. In *MEDEA '08: Proc. 9th Workshop on Memory Performance*, pages 61–68, Oct. 2008.

[42] F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero. Atomic Quake: Using transactional memory in an interactive multiplayer game server. In *PPoPP '09: Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 25–34, Feb. 2009.

[43] F. Zyulkyarov, T. Harris, O. S. Unsal, A. Cristal, and M. Valero. Debugging programs that use atomic blocks and transactional memory. In *PPoPP '10: Proc. 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 57–66, 2010.

[44] F. Zyulkyarov, S. Stipic, T. Harris, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. Discovering and understanding performance bottlenecks in transactional applications. In *PACT '10: Proc. 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 285–294, September 2010.