Discovering and Understanding Performance Bottlenecks in Transactional 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

[†]{ferad.zyulkyarov, srdjan.stipic, osman.unsal, adrian.cristal, ibrahim.hur, mateo.valero}@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 tuning programs which use transactions.

In this paper we introduce a series of profiling techniques for TM applications that provide in-depth and comprehensive information about the wasted work caused by aborting transactions. We explore three directions: (*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.

To examine the effectiveness of the profiling techniques, we provide a series of illustrations from the STAMP TM benchmark suite and from the synthetic WormBench workload. We show how to use our profiling techniques to optimize the performance of the Bayes, Labyrinth and Intruder applications.

We discuss the design and implementation of our 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.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: [Parallel programming]; D.2.8 [**Software Engineering**]: Metrics—*Performance measures*; C.4 [**Performance of Systems**]: [Measurement techniques]

General Terms

Performance, Measurement

PACT'10, September 11–15, 2010, Vienna, Austria.

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 [14]. 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 [11].

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 [2]. Several studies report that the initial versions of transactional applications can have very high abort rates [9, 18, 20]—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 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 [12] (Section 3). 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.

In Section 4 we present a series of case studies to illustrate the use of our profiling 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 [3]). Profiling revealed that our version of Bayes experienced false conflicts due to Bartok-STM's object-level conflict detection; it scaled well after modifying the data structures involved. 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 very badly when using 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 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 effect on the performance of programs using 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 [11]). 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.

```
   int taskResult = 0;

 1: while (!taskQueue.IsEmpty) {
 2:    atomic {
 3:       Task task = taskQueue.Pop();
 4:       taskResult = task.Execute();
 5:       for (int i < 0; i < n; i++) {
 6:          if (x[i] < taskResult) {
 7:             x[i]++;
 8:          } else if (x[i] > taskResult) {
 9:             x[i]--;
10:          }
11:       }
12:    }
13: }
```

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

Our paper focuses on this second kind of problem. The rationale behind this 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.

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 by 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 [29]. However, after using this technique to profile applications from 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

```
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`.

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

```
// Thread 2
atomic {
   ...
   ...
   ...
   obj1.x = t1;
   obj2.x = t2;
   obj3.x = t3;
}
```

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.

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 allow low degrees of concurrent usage.

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, 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 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 6).

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 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

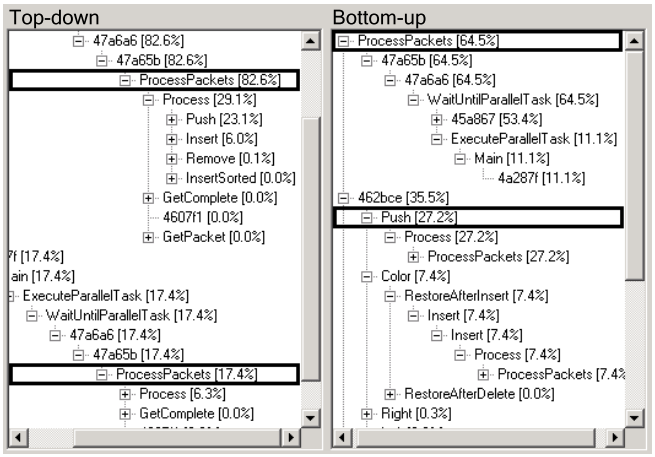

Figure 4: On the left is top-down tree view and on the right bottom-up tree view obtained from the 4-threaded execution of non-optimized Intruder application. The top-down view (left) shows that almost 100% (82.6%+17.4% summed from the two trees) of the total wasted work is accumulated at function `ProcessPackets`. 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.

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 5, 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 [6] in helping linking 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 [9, 28].

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

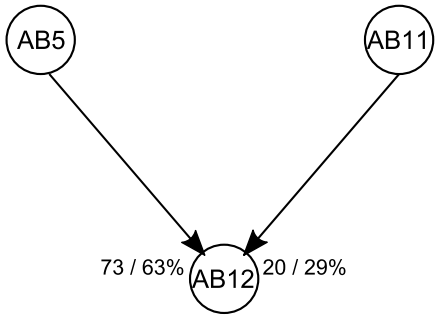

Figure 5: A part of the aborts graph drawn from the 4-threaded execution of non-optimized Bayes application. In this graph, AB5 is the `atomic` block which executes `Insert` operation, AB11 is the `atomic` which caches shared variables to thread local variables AB12 is the `atomic` block which executes function `FindBestInsertTask`. 73% of AB12's aborts were caused by AB5 amounting to 63% of wasted work and 20% of the AB12's aborts were caused by AB11 amounting to 29% of wasted work. AB5 and AB11 did not abort.

aborted less frequently and incurred less wasted work when they did abort [9, 15, 19].

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.* [26] used the `tm_waiver` keyword [17] to instruct the compiler to not instrument thread-private data structures with special calls to the STM library. In Haskell-STM [10] the user must explicitly identify which variables are transactional. To reduce the overhead of privatization safety, Spear *et al.* [22] have proposed that the programmer should explicitly tell which transactions privatize data [23]. 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 the subject of conflicts (Figure 6). 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 7 conflicts while executing `list[2]=33` (assigning a new value to the third element in a linked list). To describe the resulting conflict

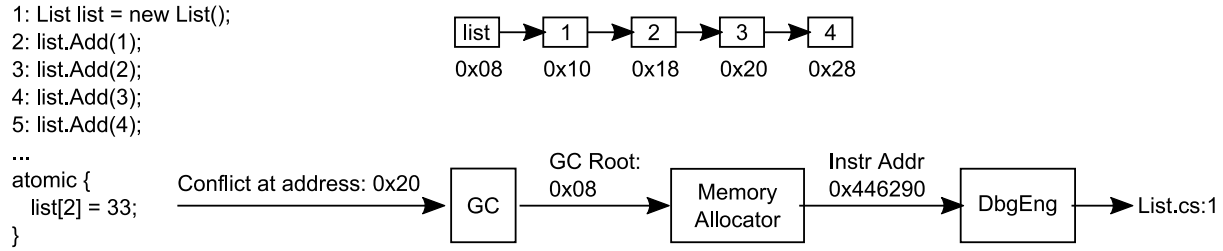

Figure 7: This figure demonstrates our method of identifying conflicting objects on the heap. The code fragment on the left creates a linked list with 4 elements. When the TM system detects a conflict in the `atomic` block, it logs the address of the contended object. During GC, the conflicting address 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.

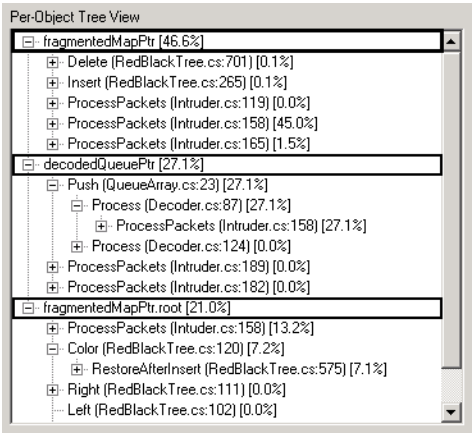

Figure 6: Per-object bottom-up abort tree. This view shows the contended objects and the different locations within the program where they have been involved in conflicts. Results shown are obtained from the 4 threaded execution of non-optimized Intruder application. For example, object `fragmentedMapPtr` 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.

to the programmer, 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 locations.

2.4 Visualizing Transaction Execution

The next aspect of our profiling system is a tool that plots a time line of the execution of all the transactions by the different threads (Figure 8). 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 8 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 [1, 4, 19, 21]. 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 8 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 8 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.

The global performance summary that our tool generates includes most of the statistics that are already used in the research literature.

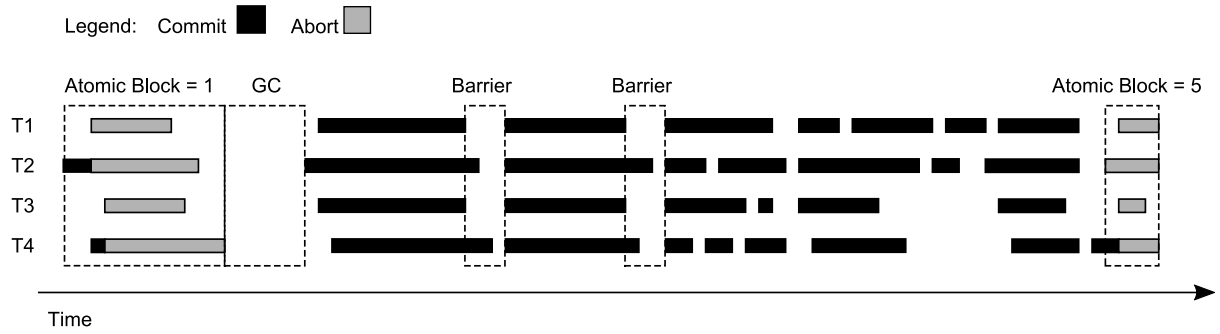

Figure 8: The transaction visualizer plots the execution of Genome with 4 threads. Successfully committed transactions are colored in black and aborted transactions are colored in gray. From this view, we can easily distinguish the different phases of the program execution such as regions with high aborts. 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.

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.

3. PROFILING FRAMEWORK

We have implemented our profiling framework for the Bartok-STM system [12]. 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 of 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 (Figure 9 and Figure 10). 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 as 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.

4. CASE STUDIES

In this section we describe our experience of analyzing the performance of applications from the STAMP TM benchmark suite [3] and from the synthetic WormBench workload [27]. The goal of this case study is to evaluate how effectively these techniques reveal the symptoms and causes of the performance lost due to conflicts in these applications.

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 Figure 11). By examining the data structures involved in conflicts, we identified that the most heavily contended object is the one used to wrap function arguments in a single object of type `FindBestTaskArg` (Figure 12(a)). Bartok-STM detects conflicts at object granularity, and so concurrent transactions cannot use the different fields of a given `FindBestTaskArg`. In this case, this form of false conflict resulted in 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 12(b)). After this small optimization Bayes scaled as expected (Figure 11).

Furthermore, the profiling results showed that the `atomic` block whose aborts caused the greatest wasted work (92%) is the one which executes the function `FindBestInsertTask`. The reason why this `atomic` block aborts repeatedly is because it involves long-running read-only transactions that are exposed to conflicts from other threads. The `atomic` block which performs the `Insert` operation caused 73% of the aborts in the `FindBestInsertTask` function and the `atomic` block which caches shared variables caused 20% of `FindBestInsertTask`'s aborts (Figure 5 illustrates this, with AB12 being the atomic block in the `FindBestInsertTask` function, AB5 being `Insert` and AB11 being `Cache`).

#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

Figure 9: 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

Figure 10: 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.

```
//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 12: 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.

```
   // Original
1: char[] data = new char[length];
2: Array.Copy(packetPtr.Data, data, length);
3: Decoded decodedPtr = new Decoded();
4: decodedPtr.flowId = flowId;
5: decodedPtr.data = data;
6:
7: Queue decodedQueuePtr = this.decodedQueuePtr;
8: decodedQueuePtr.Push(decodedPtr);
              (a)

   // Optimized
1: Decoded decodedPtr = new Decoded();
2: Queue decodedQueuePtr = this.decodedQueuePtr;
3: decodedQueuePtr.Push(decodedPtr);
4:
5: char[] data = new char[length];
6: Array.Copy(packetPtr.Data, data, length);
7: decodedPtr.flowId = flowId;
8: decodedPtr.data = data;
              (b)
```

Figure 13: Code fragment from Intruder: (a) is the original code fragment from Intruder – most of the conflicts happen in line 8 which result in substantial wasted work; (b) we moved the call to `Push` up in the code to line 2 – this optimization didn't reduce abort rate, but reduced the wasted work.

In Figure 10 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.

In this experiment, Bayes is a representative example of applications that require optimizations after being ported from a C-based environment to an object oriented environment.

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 [3]. Optimizing this application helped us determine the effectiveness of our profiling techniques.

The most contended objects in Intruder were `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 a lookup. On the other hand, the lookup was invoked while adding a new entry to check if it already exists. We optimized the program by replacing the red black tree with a chaining hashtable, which allows higher degree of parallelism. The performance improved significantly and the wasted work on `decodedQueuePtr` became the most dominant. Although the wasted work was the highest for this data structure, the contention was low. Unlike our approach, basic conflict point discovery would fail to detect this fact and direct the programmer's attention to other parts of the code that are not actually bottlenecks.

The `decodedQueuePtr` object is a queue data structure that stores a reference to already reassembled packets. All the conflicts were occurring when inserting the reassembled packets in a queue (i.e., in the longest `atomic` block executing function the `Process`). The underlying implementation of the queue was an array data structure. We thought that like in Bayes, because conflicts are detected at object granularity, the concurrent accesses to

#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

Figure 11: 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.

#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

Figure 14: 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.

the head and the tail of the queue were falsely causing conflicts. Therefore we changed the queue with one that uses linked list, but the performance did not improve. We then noticed that conflicts do not happen for both `Push` and `Pop` operations but only for `Push`. After, we have examined the code more carefully we noticed that the call to `Push` is the last statement of the `atomic` block. It was preceded with computation causing significant wasted work on abort. As shown in Figure 13, we moved the call to `Push` to be the first statement in the `atomic` block. This did not reduce the abort rate but reduced the wasted work as conflicts were being detected earlier in the transaction's execution, thus wasting little time in executing code that eventually will be rolled-back. After these two optimizations, Intruder scaled as expected (see Figure 11).

Needless to say that in our case the latter optimization was effective because Bartok-STM detects conflicts for write operations eagerly. In the case of TM systems with lazy conflict detection, this optimization would not be so effective because the TM system would still continue the speculative execution while the transaction is invalid.

To verify that these changes do not degrade performance on other TM systems, we made corresponding changes to the original version of Intruder. Then, we executed the resulting application using the TL2 STM library [8], and in a simulated environment using some state-of-the-art HTM systems (Scalable-TCC [5] and Eazy-HTM [24]). The results that we have obtained are consistent and shown in Figure 14.

In this experiment, Intruder is a representative example of applications which can be optimized by using different data structures such as hashtable that allow more parallelism. Also, another optimization that TM applications can benefit is to move code statements, a technique that optimizing compilers use pervasively.

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.

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.* [25]. 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. After this optimization Labyrinth's execution was similar to the one reported by the STAMP suite's authors [3] (see Figure 11).

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. Nevertheless, we still profiled them to see if we could obtain further improvements. In Genome most of the aborts occurred in the first and the last `atomic` blocks in the program order (see Figure 8). With 4 threads, the total wasted work in Genome amounted to 1.2% of the program execution. In Vacation, almost all aborts occurred in the `atomic` block which makes reservations. With 4 threads, the total wasted work in Vacation amounted to 2% of the program execution. In our setup, WormBench had almost no conflicts—in 4-threaded execution, from 100 000 transactions only 10 aborted.

In these applications, there was not any opportunity for further optimizations. However 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.

5. RELATED WORK

In previous work we [29] and Lev and Herlihy [13] introduced techniques for debugging transactional programs. These techniques fo-

cus on identifying correctness errors, rather than investigating performance.

Chafi *et al.* developed the Transactional Application Profiling Environment (TAPE) which is a profiling framework for HTMs [4]. 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 [7] (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 [6] introduced dynamic conflict graphs (DCG). A coarse grain DCG represents the abort relationship between the `atomic` blocks similar to aborts graph (see Figure 5). 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 [29]. 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.* [16] 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 [21]. 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 this earlier 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 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.

6. CONCLUSION AND FUTURE WORK

In this paper we have introduced new techniques for profiling transactional applications. The goal of these profiling techniques is to help programmers find the bottlenecks specific to the program rather than the bottlenecks specific to the underlying TM system. To generate more comprehensive results we have extended our previous work on conflict point discovery. The extensions include metrics such as WastedWork and ConflictWin, assigning context to conflict points, building abort graphs, visualizing the transactions and identifying conflicting objects and data structures. We report all results in source code level such as variable names and statements.

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, Labyrinth and Intruder. Bayes is an example where programs do not perform as expected when ported from non-object oriented environment such as C to object oriented environment such as C# or vice-versa. Labyrinth is an example where the programmer may give hints to the underlying TM system about the shared data structures and the operations applied on them. Intruder is an example of a program with poor performance which can be improved by using data structures with higher degree of parallelism and restructuring the code to reduce the wasted work.

In future work we plan to implement profiling frameworks for other TM systems and profile more complex applications that are implemented in other platforms. We also plan to make the C# versions of the STAMP applications from this work publicly available.

Acknowledgments

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

7. REFERENCES

[1] 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, Feb. 2009.

[2] 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.

[3] 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, Sept. 2008.

[4] 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.

[5] 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.

[6] 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, Jan. 2010.

[7] 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, March 2009.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] M. Herlihy and Y. Lev. tm_db: A generic debugging library for transactional programs. In *PACT '09: Proc. 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 136–145, Sept. 2009.

[14] 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.

[15] 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.

[16] 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, July 2009.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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. Aug. 2007.

[24] 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, Dec. 2009.

[25] 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, Sept. 2007.

[26] 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.

[27] 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.

[28] 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.

[29] 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, Jan. 2010.