Abstract—In the last few years, Transactional Memories (TMs) have been shown to be a parallel programming model that can effectively combine performance improvement with ease of programming. Moreover, the recent introduction of TM-based ISA extensions, by major microprocessor manufacturers, also seems to endorse TM as a parallel programming model for today’s parallel applications. One of the central issues in designing Software Transactional Memory (STM) systems is to identify mechanisms/heuristics that can minimize contention arising from conflicting transactions. Although a number of mechanisms have been proposed to tackle contention, such techniques have a limited scope, as conflict is avoided by either interrupting or serializing transaction execution, thus considerably impacting performance. To deal with this limitation, we have proposed a new effective transaction scheduler, along with a conflict-avoidance heuristic, that implements a fully cooperative scheduler that switches a conflicting transaction by another with a lower conflicting probability. This paper extends such framework and introduces a new heuristic, built from the combination of our previous conflict avoidance technique with the Contention Intensity heuristic proposed by Yoo and Lee. Experimental results, obtained using the STM Bench7 and STAMP benchmarks atop tinySTM, show that the proposed heuristic produces significant speedups when compared to other four solutions.

I. INTRODUCTION

The introduction of multicore architectures has renewed the search for programming models that can simplify the creation of efficient parallel applications. Among the various models proposed so far for parallel programming, Transactional Memory (TM) [1] has been considered a serious contender for the post of efficient yet simple-to-use parallel programming model. Evidence for its prominence in the processor industry is the recent series of announcements of Hardware TM (HTM) ISA extensions by IBM [2], AMD [3], [4], and Intel [5].

Hardware TMs (HTMs) are an interesting option because they integrate the support for transactions at the lowest, most efficient, architectural level. On the other hand, for some applications, HTMs can have their performance hindered by the lack of scalability and cache capacity overflow [2]. Software Transactional Memories (STMs) [6], in contrast to HTMs, tread in the opposite design direction. STMs are implemented atop the processor and, thus, allow more flexible designs as they do not limit the size of their critical data structures to the hardware constraints. Eventually, as research evolves, the combination of HTM/STM systems could lead to the best solution for a larger range of applications.

STM systems offer flexibility and potential for scalable performance, but they can have their performance considerably degraded under high-contention workloads, as the transaction conflict rates increase. Contention management mechanisms [7] try to minimize contention by selectively deciding, at conflict time, which of the conflicting transactions must be aborted and when it should be scheduled for re-execution. So, contention managers have their power to improve performance limited by the fact that they act after a conflict has been detected.

Recently, a new set of mechanisms have been proposed [8]–[10] to tackle contention by conflict avoidance as they try to predict which transactions are likely to conflict if executed concurrently, thus enabling the STM to avoid their simultaneous execution. Usually, conflict avoidance mechanisms are implemented by continuously monitoring the level of transaction content in the system and triggering the serialization of transactions while its value exceeds a given threshold [11]. Unfortunately, such techniques have a limited scope, as they do not try to replace the transaction whose execution has been delayed by another so as to sustain higher levels of transactional throughput. Instead, they avoid the conflict by simply serializing the execution of the potentially conflicting transactions.

To deal with this limitation, we have proposed a new transaction scheduling, along with a conflict-avoidance heuristic, which implements a fully cooperative transaction scheduler [12], [13]. This scheduling presents two main advantages when compared to state-of-the-art TM schedulers. First, it deals with the pseudo parallelism in an elegant way, by only spawning as many system-level threads as the number of available processor cores and handling the exceeding threads internally. Second, it allows the TM subsystem to efficiently access the runnable transaction queue and switch the execution to any of them. This allows the design of more precise proactive scheduling mechanisms, not possible with other know approaches, that are restricted to either serialization or yielding. This paper extends such framework and introduces a new heuristic, by integrating the conflict avoidance mechanism described in [12] with the Contention Intensity heuristic based on the work of Yoo and Lee [10]. By doing so, we show that this new heuristic effectively improves STM performance, mainly on high contention applications.
II. THE TRANSACTION SCHEDULER

The dynamic transaction scheduler (DTS) is the component that implements policies to condition the execution of transactions with the sole purpose of increasing the probability of a concerted, frictionless, transactional execution. DTS is in sharp contrast with traditional contention management (CM), as CMs tend to focus primarily on conflict resolution. The differences between DTS and traditional CMs can be summarized as:

- CMs work at the moment a conflict among already active transactions is detected. DTS works before transactions are started.
- CMs are usually implemented inside the TM. DTS is implemented as a separate component that is logically at the same level of the STM, below the application and on top of the operating system (OS).

DTS provides to the STM library and application a set of methods that allow the management of threads and transaction scheduling. As an example, there is a method to create a pool of working threads \( \text{sched\_init} \) and a method to instruct the threads in the pool to start the execution of a given transactional program \( \text{sched\_start} \). It is worth to note that DTS does not include any synchronization support, e.g., locks and conditional variables, because we assume that the synchronization support is provided by the associated STM library. However, there is one important exception: barrier synchronization. Given that barriers are very common in parallel programming, DTS supplies the \text{sched\_barrier} interface.

A. Thread Operation

The design of DTS is based on the assumption that the number of system-level threads (SLTs) should not exceed the number of available cores. In other words, if \( \text{sched\_init} \) is called with argument 16 and there are only 8 cores in the system, only 8 working threads are effectively created. It is important to note that, from the application point of view, 16 threads have been created. In order to deal with this scenario, DTS represents each thread internally using execution context records (ECRs). Each ECR encapsulates the state of one thread. When an ECR is created, it is assigned to a worker thread for execution. When \( \text{sched\_init} \) is invoked, the scheduler creates as many ECRs as necessary and performs a system call to create as many SLTs as the number of cores of the system. Each ECR is then queued into the DTS queue of runnable threads, becoming eligible for later execution. The DTS dispatcher is responsible for taking an ECR from the runnable queue and mapping it to a system-level thread, as shown in Fig. 1

As previously mentioned, each ECR stores the state of a single thread. When the number of ECRs is equal or less than the number of SLTs, the behavior of the system is similar to conventional systems: each ECR is continuously mapped to a single SLT. When the number of ECRs is greater than the number of available SLTs, the dispatcher selects an ECR and maps it into an SLT for execution. After an ECR is mapped and starts executing, there are only two ways for the corresponding SLT to become free again: either the work is finished or the ECR voluntarily relinquish control.

B. Transaction Scheduling

The most important function of the DTS is, of course, the scheduling of transactions. At the DTS-STM interface there is a method that allows an STM library to switch the execution from one ECR (thread) to another ECR (thread) by calling \text{sched\_switch}. Because there is only one transaction per ECR, these interfaces effectively allow the STM library to resume execution of another transaction based on a given heuristic. Note that this service is more powerful than simply yielding to an ECR, as implemented by the Adaptive Transaction Schedule (ATS) proposed by Yoo and Lee [10], because, in our case, the STM has finer control over which transaction is selected for execution.

In the next section, we detail the heuristic methods that DTS uses to implement a transaction scheduler that effectively reduces contention.

III. SCHEDULING HEURISTICS

In order to decrease contention, we have developed a new, proactive heuristic, named best alternative transaction (BAT), by integrating the conflict avoidance mechanism proposed in [12] with the Contention Intensity heuristic proposed by Yoo and Lee in the ATS system [10]. By doing so, we show that BAT effectively improves STM performance, mainly on high contention applications. Before introducing BAT, we will first outline the Contention Intensity (CI) heuristic.

A. The rationale behind CI

The effectiveness of a transaction, that is, the chance that once started it progresses and commits, is directly related to the competition for shared resources encountered during its
execution. In ATS, each transaction, materialized by a thread, maintains its own estimate of such competition, by measuring an index, called Contention Intensity (CI). Equation 1 below shows how ATS defines the contention intensity for the n-th activation of a transaction:

\[ CI_n = \alpha * CI_{n-1} + (1 - \alpha) * CC \]  (1)

Contention intensity is computed by combining, with different weights, two components of the contention experienced by the transaction. The \( CI_{n-1} \) component captures the contention experienced by the transaction from its first \((n = 1)\) activation up to its \((n-1)\)-th activation. The component \( CC \) measures the transaction contention, for its current \((n)\)-th activation. A weight \( \alpha \) is used to adjust the importance of the past and current contention components in the equation. Note that \( \alpha \) and its complement are used in the computation of \( CI_n \). By applying competitive learning between these two predictors, the \( \alpha \) can be adjusted automatically. However, throughout our experiments, we fixed \( \alpha \) to 0.30, giving a little more weight to \( CC \), because this value showed the best performance. Initially, CI is set to 0, and the equation is evaluated at each commit or abort operation, with \( CC \) set to 0 on commits and to 1 on aborts.

When a transaction \( T \) is about to be started, the scheduler checks whether the \( CI \) of \( T \) is above a given threshold (again, throughout our experiments, we set the threshold at 0.70). If this is the case, the BAT heuristic is enabled to find an appropriate alternative transaction to take the place of \( T \) and a call to the method \texttt{sched_switch} is carried out to replace \( T \) with the newly selected transaction. By contrast, ATS simply serializes transactions whose CI has surpassed a given threshold. Of course, if the \( CI \) of \( T \) is below the threshold, then it is scheduled for execution (re-execution) without further delay.

**B. The rationale behind BAT**

BAT is based on a very simple rationale: maintain a global snapshot of the transactions active in the STM and select as the next transaction to be executed the transaction with the lowest probability of conflicting with those in the snapshot. The effectiveness of BAT depends on two factors:

- high accuracy: the transaction chosen by BAT should have a high probability of commit;
- low overhead: the execution of BAT must have a low impact on the overall execution of the transactional system.

The accuracy of BAT is highly dependent on the diversity of transactions available in the application. It is not difficult to see why by examining the extreme case where the application has only one transaction; the worst case scenario. At any moment, the STM will execute several instances of the same transaction. These, in their turn, will have the same structure, same expected duration and compete for the same shared resources. So, the probability of conflicts happening is going to be very high. In addition, at any moment, BAT will not be able to find a best alternative transaction to include in the set of active transactions because there is only one transaction available. In the best case scenario, BAT will be dealing with an application structured as many different transactions, possibly reading/writing different locations of the shared data space during their execution. Here, BAT is going to have at its disposal a diverse set of transactions from which it is going to pick the transaction the chance of conflict with the ones already in the snapshot, this is the best alternative transaction.

Suppose an activation and execution of BAT takes on average \( C \) cycles. The policy used to decide whether it is worth imposing the \( C \) overhead on the STM to minimize conflicts among transactions is the following: if the average duration of a transaction is shorter than \( k \times C \), where \( k \) has been found by actively monitoring the duration of transactions in the applications, then an heuristic for short transactions must be activated. Otherwise, the STM is considered as working mostly with long transactions and BAT for log transactions must be activated. In summary, the duration \( k \times C \) splits transactions into the classes short and long, and in each case BAT chooses an adequate heuristic to keep the overhead of DTS as low as possible. The factor \( k \) is computed using a sliding window average of the durations of the last window transactions with \( window \) set to one hundred (100); for the experiments the threshold for \( k \times C \) has been set to 100,000 cycles. During the profiling carried out to determine the threshold we have also found that an hysteresis should exist between the computation of the threshold and its use in the activation of each heuristic. DTS switches to the heuristic adequate for long transactions only after two successive values of the sliding window average are above the threshold, and it is switched back to the heuristic for short transactions as soon as the average falls below the threshold. Next, we discuss the behavior of DTS in each of these scenarios.

1) **Scenario for short transactions:** As said before, the heuristic chosen to select the next transaction to run in a short transactions scenario should have near-zero runtime overhead and a fair conflict prediction. So, in this case the best heuristic is to simply let DTS scheduler select as the next transaction to run any transaction different from the current transaction.

2) **Scenario for long transactions:** The goal of our heuristic in this scenario is to predict the best candidate transaction for execution given a set of active transactions. We say that a transaction is the best candidate when its conflict probability is the lowest one among the competing transactions. During execution time we record the past conflict history of a transaction in a table. In order to clarify the concepts of our higher-accuracy heuristic we provide now a detailed example:

Assume a system with 4 cores executing an application containing 3 transactions, represented by its unique identifiers (IDs) 1, 2 and 3. Besides the contention intensity (CI) maintained for each transaction, there are three more metadata employed by the higher-accuracy BAT heuristic: a vector with active transactions \((activeTx)\), a conflict table \((conflictTable)\), and a summary of the best candidate transactions \((bestTx)\) as illustrated in Fig. 2. The \(activeTx\) vector maintains, for each core, the identification number of the transaction that it is currently running on that core. When no transaction is assigned to a core we make use of the ID -1. The active transactions in the system illustrated in Fig. 2 is given by the set \{-1, 1, 3, 2\}. Therefore, core 0
is not executing any transaction, whereas cores 1, 2, and 3 are executing transactions 1, 3, and 2, respectively.

Recall from the illustration given in Fig. 2 that the conflict probabilities for transactions 1 and 2 are, respectively, 0.5 and 0.2. Therefore, bestTx must be updated because there are other transactions with lower or equal conflict probability than transaction 3 (in the example, transaction 2). This action is performed by the lines 23 to 32 in the pseudo-code.

Listing 1: Moderate-Overhead High-Accuracy BAT heuristic

We only need to check if a better transaction exists if the aborted transaction was previously in bestTx (line 23). In this case, we need to check every column of conflictTable in order to find the new best value (lines 24 to 31). When two transactions have the same conflict probability, as in the example, the algorithm makes use of the contention intensity of both transactions, as a tiebreaker. This strategy is based on the fact that the conflictTable indicates only the probability of conflict for a specific scenario but does not give a more precise idea of conflict history of the transaction itself, better shown by its contention intensity.

If the selected transaction commits instead of aborting we need to decrease its conflict probability (pseudo code lines 34 to 45). This operation is similar to abort, except that we do not need to check every column of conflictTable to find a better candidate transaction. The only way for bestTx to change is if the committing transaction was not previously the best choice and, as a result of decreasing its conflict probability (line 38), it became the best choice (lines 39 to 45). For instance, if we picked transaction 2 instead of 3 during the start operation (assuming there was no transaction 3 to be dispatched) then, at commit time, we would update its conflict probability to 0.1 and update bestTx to reflect that transaction 2 may be now the best candidate, just depending if its CI is less than or equal to the CI of the transaction 3.

Contrary to the low-overhead heuristic suitable for short transactions, where we might only use one variable by transac-
tion to hold its contention intensity, this high-accuracy heuristic is more elaborated and, consequently, results in a higher runtime overhead. This is because such a heuristic can capture information about which transactions conflict with others. Once this information is collected it can be viewed in the context of global data structures named conflictTable, activeTx and bestTx in order to predict what is the best transaction to execute given a set of running transactions. BAT is constructed by combining these heuristics into a transaction predictor that can be dynamically controlled.

IV. EXPERIMENTAL SETUP

The experiments were conducted on a machine with two 6-core Intel Xeon E-5645 processors (24 cores in total, hyper-threading enabled), clocked at 2.40GHz with 32GB of RAM. The machine runs a standard Ubuntu Server 10.04.3 LTS amd64. The evaluation used the STMBench7 [14] and STAMP 0.9.10 [15] benchmarks. All applications were compiled using gcc version 4.4.3.

The experiments aimed at comparing the BAT heuristic running under the DTS scheduler with four other systems that have been designed to improve the performance of STMs:

**Tiny** TinySTM, version 1.0.0. For the experiments, TinySTM was configured with the write-back and ETL strategy. The contention policy adopted was CM_SUICIDE, that immediately restarts a transaction on abort.

**ATS** An implementation of ATS scheduler proposed by Yoo and Lee [10], employing a single global queue for all transactions. The values used in the experiments that resulted in the best overall performance for ATS, are: 0.75 for $\alpha$ and 0.7 for threshold.

**Shrink** An implementation of the Shrink scheduler proposed by Dragojevic et al. [11]. The code was taken from the authors website for TinySTM version 0.9.5, and adapted to the current version (1.0.0). We did not change the parameters in the code: succ_threshold = 0.5, locality_window = 4, confidence_threshold = 3, $c_1 = 3$, $c_2 = 2$, $c_3 = 1$.

**LUTS** The LUTS scheduler with the conflict-avoidance heuristic introduced in [12]. In order to predict the conflict beforehand LUTS was adopted, for the Contention Police, $\alpha = 0.75$ and threshold = 0.5.

The configuration used for the STMBench7 is in table I below:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long traversals</td>
<td>False</td>
</tr>
<tr>
<td>Workload type</td>
<td>read dominated</td>
</tr>
<tr>
<td>Duration</td>
<td>5000ms</td>
</tr>
<tr>
<td>Size</td>
<td>Small, Medium, Big and Huge</td>
</tr>
</tbody>
</table>

For the experiments with the STAMP applications [15], the parameters listed in table II were adopted. They are the recommended configurations and data set for use in real machines. All speedup results showed here are normalized to the sequential execution. Results for the application Bayes were omitted since this application did not provide reproducible behavior and displayed a large variance. This very same behavior has already been reported in the literature [16]. For the applications Vacation and Kmeans only the results for configuration High have been included in the paper, because the results for the Low configuration are very similar.

<table>
<thead>
<tr>
<th>Application</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>genome</td>
<td>-s64 -n102400 -l1000000 -i ttimeu1000000</td>
</tr>
<tr>
<td>intruder</td>
<td>-s10 -n128 -m262144 -s1</td>
</tr>
<tr>
<td>kmeans-high</td>
<td>-m15 -n15 -n0.00001 -i random-n65536-d32-c16</td>
</tr>
<tr>
<td>labyrinth</td>
<td>-i random-x512-y512-z512-z7-n512</td>
</tr>
<tr>
<td>ssca2</td>
<td>-s20 -i1.0 -u1.0 -i3 -p3</td>
</tr>
<tr>
<td>vacation-high</td>
<td>-n4 -g60 -r60 -r1048576 -w4194304</td>
</tr>
<tr>
<td>yada</td>
<td>-a15 -i $timeu1000000.2</td>
</tr>
</tbody>
</table>

V. RESULTS AND ANALYSIS

All the results presented in this section were averaged over 30 executions. The histograms show the averages and a 95% confidence interval. Our presentation of the results and analysis assesses first the results obtained with STMBench7 and next STAMP.

DTS implements BAT heuristic that is expected to provide speedup gains when transaction diversity is reasonably large and contention is high. Higher contention is obtained by increasing the number of threads, thus transactions, simultaneously active.

A. STMBench7

STMBench7 uses a graph composed of objects (atomic and composite) as the basis for the implementation of over forty different transactional operations. Therefore, it is fair to say that this benchmark provides a reasonably large transaction diversity.

As shown in Figures 4(a), 4(b), 4(c) and 4(d), DTS performance is about the same as other systems in the interval that goes from 1 to 16 threads given that the number of threads is relatively small, generating moderate contention. This shows that DTS does not impose a significant overhead in workloads that do not favor the BAT heuristic.

The benefits of BAT heuristic becomes clear as the number of threads as well the sizes of the data sets increase. For configurations Small (Figure 4(d)) and Medium (Figure 4(c)) DTS gives better speedups from 64 threads and up. For configurations Big (Figure 4(b)) and Huge (Figure 4(a)), there is an improvement over the other systems, as soon as the number of threads reaches 32. In the STMBench7 Big and Huge scenarios the length of the average transaction considerably grows (as the transaction length increases with the data size). This confirms that BAT selects a best transaction alternative for the cases when the cost of an abort is larger due to long running transactions.
Fig. 3: STMBench7: The ops/secs and speedup comparisons

(a) Huge

(b) Big

(c) Medium

(d) Small
Fig. 4: STAMP. Time and speedup comparisons for class I.
Fig. 5: STAMP. Time and speedup comparisons for class II.
STAMP applications can be split in two major classes. The first one is composed of Genome (Fig. 5(a)), Ssca2 (Fig. 5(b)), Vacation (Fig. 5(c)) and Yada (Fig. 5(d)). The first three applications have short transactions and very low contention level (typically below 1). In Ssca2 transactions are extremely short (around 1 transactional read and 2 transactional writes), that amplifies the overhead of the heuristics. The other application, Yada, is composed of both short and long transactions. From the total of 5 active transactions in this application, 3 are extremely short (usually only a single transactional read or write). On the other hand, the contention level is very high (up to 250!), thereby providing many scheduling opportunities, mainly in the overloaded scenario (32 threads or higher), and hence an improvement in the results.

The other class consists of Kmeans (Fig. 6(a)), Labyrinth (Fig. 6(b)), and Intruder (Fig. 6(c)). For Kmeans and Intruder, the contention level provides a lot of opportunities. The short transactions contained in Kmeans and Intruder are not as small as in Ssca2 and, given the ideal contention level, they should allow DTS to improve the performance for scenarios above 32 threads. Albeit its low contention level, labyrinth has very long transactions and, thus, is very favorable for heuristics such as DTS’s BAT (and LUTS) whose main goal is to prevent aborts from happening.

VI. RELATED WORK

Researchers have long realized the importance of avoiding conflicts in transactional memory systems. Initial attempts to mitigate their negative effects relied on the so-called contention managers, introduced by Herlihy et al. in 2003 as a mean to guarantee progress in obstruction-free STM implementations [17]. The idea is very simple and elegant: when a transaction discovers a conflict, it consults a contention manager to determine how to proceed. Therefore, different resolution policies can be created and “plugged in” without affecting the correctness of the underlying transactional algorithm.

A host of resolution policies for contention managers have been proposed during the past 10 years [18]–[22]. Despite all advances achieved by the community, contention managers have an important drawback that limits its applicability: they only take action after a conflict has already happened. To counter this limitation, scheduling-based contention management approaches have emerged [8]–[11], [23], [24]. The basic idea of this new strategy is to use some information about the past history to decide whether a given transaction should be scheduled for execution or not.

To the best of our knowledge, the first scheduling-based proposal is due to Yoo and Lee [10]. Their approach, named Adaptive Transaction Scheduling (ATS), keeps a per-thread measurement of contention intensity as a predictor to decide if a transaction should be allowed to start. In case the contention is too high, transactions are inserted into a queue and serialized, appropriately avoiding repeated aborts. ATS is very attractive since it lends itself to simple implementations. CARSTM [9] is a transactional scheduler that maintains a queue of transactions for each core. When a conflict between two transactions arises, the scheduler inserts the aborted transaction into the aborter’s queue. Therefore, the likelihood of repeated aborts is decreased because the aborted transaction is serialized after the aborter. The same approach is used by Steal-on-Abort [8] with slightly variations regarding the location the aborted transaction is inserted in the aborter’s queue (e.g., at the tail or at the head). Our approach is similar to CARSTM in the sense that we also restrict the number of running transactions to the available cores. However, we rely on DTS support to find a candidate transaction instead of serializing the execution.

Shrink [11] makes use of past access patterns of a thread to predict whether a transaction should be allowed to start in the future, thus preventing conflicts if the prediction is accurate. When a transaction is about to start, Shrink checks if any address in the predicted read and write sets is being written by an active transaction. If that is the case, the transaction is serialized. To avoid unnecessary serialization, Shrink also measures the contention intensity and serializes only if it is above a certain threshold. Maldonado et al. [24] investigate kernel-level support for serializing contention management, guided by the motivation that user-level implementations are costly since they require system calls. An inconvenience of this approach is that the OS kernel needs to be modified. Blake et al. [25] propose a prediction mechanism based on the past conflict history of transactions. The system keeps a conflict table that stores the likelihood of a conflict between any pair of transactions in the system. The scheduler relies on this information to decide whether a transaction should proceed or be serialized. Our heuristic to avoid conflicts among long running transactions is similar to Blake’s work with an important difference: when a conflict is encountered, we use the information stored in the conflictTable to choose another transaction to execute instead of serializing the execution.

Recent works have focused on providing more accurate prediction and resorting to serialization only as the last option [26], [27]. In special, the work by Attiya and Milani [23] provides a formal description of schedulers targeted at read-dominated workloads. The approach provided by DTS is in sharp contrast with the majority of the related work, since it aims at finding another suitable transaction instead of serializing the execution.

VII. CONCLUSION

The main contribution of our work is to show that implementing good heuristics integrated with a dynamic transaction scheduler can yield significant performance benefits. In this article we have described BAT, a heuristic that has its roots in our observation that a global view of the status of the active transactions, implemented by a conflicting table, can improve accuracy of the scheduler’s decision, thus minimizing aborts, specially for scenarios with longer transactions. BAT combines and improves on two other heuristics: conflict avoidance from [12] and contention intensity from [10].

We have tested the BAT heuristic using two well-known and widely accepted benchmarks: STMBench7 and STAMP with tinySTM as the baseline. We also plan to include SwissTM in the tests. We already have preliminary results for BAT upon SwissTM and so far they are well aligned with the results presented here.
The results obtained with STMBench7, with emphasis on the Big and Huge data set sizes, show that BAT indeed improves the speedup of STMBench7 in setups that generated high contention (heavy workload), large number of threads, and with diverse but long duration transactions. The results obtained with STAMP also confirm that our heuristic points in the right direction, as DTS does well for applications with higher transaction diversity, longer durations, and heavier workloads. Overall, the results confirm that it is possible to improve the performance of STMs using a transaction scheduler implemented outside the STM, and a combination of well-designed heuristics. For STMs the overhead of any associated mechanism is critical. In the case of DTS, the results show that it is possible to have modularity at a reasonable cost; this is the second contribution of our work.

Potential future research based on this work is focused on exploring new heuristics. Ideally such heuristics should select the best transactions to execute and minimize the overhead imposed by high-accuracy heuristics, while avoiding the loss of information when switching from a high-accuracy heuristic and a low-overhead (low-accuracy) one.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for the insightful comments. This work is supported by FAPESP under grant number 2011/19373-6.

REFERENCES


