Dynamic Transaction Coalescing

Srđan Stipić, Vasileios Karakostas, Vesna Smiljković,
Vladimir Gajinov, Osman Unsal, Adrián Cristal, Mateo Valero
Barcelona Supercomputing Center, Spain
{srdjan.stipic, vasilis.karakostas, vesna.smiljkovic
vladimir.gajinov, osman.unsal, adrian.cristal, mateo.valero}@bsc.es

ABSTRACT

Prior work in Software Transactional Memory has identified high overheads related to starting and committing transactions that may degrade the application performance. To amortize these overheads, transaction coalescing techniques have been proposed that coalesce two or more small transactions into one large transaction. However, these techniques either coalesce transactions statically at compile time, or lack on-line profiling mechanisms that allow coalescing transactions dynamically. Thus, such approaches lead to sub-optimal execution or they may even degrade the performance.

In this paper, we introduce *Dynamic Transaction Coalescing* (DTC), a compile-time and run-time technique that improves transactional throughput. DTC reduces the overheads of starting and committing a transaction. At compile-time, DTC generates several code paths with a different number of coalesced transactions. At runtime, DTC implements low overhead online profiling and dynamically selects the corresponding code path that improves throughput. Compared to coalescing transactions statically, DTC provides two main improvements. First, DTC implements online profiling which removes the dependency on a pre-compilation profiling step. Second, DTC dynamically selects the best transaction granularity to improve the transaction throughput taking into consideration the abort rate. We evaluate DTC using common TM benchmarks and micro-benchmarks. Our findings show that: (i) DTC performs like static transaction coalescing in the common case, (ii) DTC does not suffer from performance degradation, and (iii) DTC outperforms static transaction coalescing when an application exposes phased behavior.

1. INTRODUCTION

Transactional Memory (TM) [9, 8] is a concurrency mechanism that simplifies the development of multi-threaded applications. It has been shown that TM is easier to use than locks [12] because it frees the programmer from having to implement the synchronization to access shared data. Programming with locks is known to be error-prone and can cause deadlocks or erroneous execution due to multiple releases of the locks. Additionally, protecting critical sections with locks is non-composable because the code using lock can not be treated as a "black box".

Most of the current TM systems are implemented as a library in software [7, 5] or as compiler extensions [11]. In order to support concurrent execution of transactions, a software TM (STM) system manages transactional metadata during transactional execution. This creates overheads related to transaction initialization, versioning, read/write instrumentation, etc. At the beginning of each transaction, the STM system initializes transactional metadata (read and write sets) and creates a checkpoint of the current thread state (the register file and local variables). If the transaction aborts due to a conflict with another transaction, the checkpoint is used to restore the state from the beginning of the transaction. Otherwise, it commits the changes in shared data and associated metadata.

Prior work in STM has identified high overheads related to starting and committing transactions [15, 18, 4, 14]. In case the executed transactions in an application are numerous and small, the overheads accumulate rapidly – even more than 70% of the total execution time in extreme cases [4] – and may degrade the application performance. To mitigate these overheads, researchers proposed various techniques to coalesce transactions in order to amortize the cost of creating and committing transactions [4, 18, 14]. However, these techniques either coalesce transactions statically at compile time, or lack on-line profiling mechanisms that allow coalescing transactions dynamically.

For example, Stat-TC [14] is the state-of-the art transaction coalescing technique[1]. Stat-TC is a profile-guided compiler optimization technique that coalesces two or more small transactions into one large transaction. However, Stat-TC coalesces transactions *statically* at compile-time. Such an approach suffers from two important drawbacks: (i) the number of coalesced transactions may be sub-optimal, and (ii) the application performance may degrade when the number of aborts increases due to longer transactions. Thus, Stat-TC cannot adapt to dynamic program behavior, such as when the application exposes phased behavior with different abort rates or when the thread count changes.

In this paper we introduce *Dynamic Transaction Coalescing* (DTC), a compile-time and run-time technique that

[1]In this paper we refer to [14] as Stat-TC for the sake of readability and clarity.

CF'14, May 20 - 22 2014, Cagliari, Italy

http://dx.doi.org/10.1145/2597917.2597930

improves the application performance. At compile-time, DTC generates several code paths with a different number of coalesced transactions. At runtime, DTC implements low overhead online profiling and dynamically selects the corresponding code path that improves throughput. In this way, DTC improves the transaction throughput of the loops executing transactions. To evaluate the effectiveness of DTC, we used various benchmarks (CLOMP-TM, SSCA2, Vacation) that are widely used in TM research, and micro-benchmarks (hash-table and red-black tree). Our findings show that DTC improves the performance of SSCA2, Vacation, CLOMP-TM, and hash-table by 44.4%, 45.8%, 66.9%, and 62.9% respectively when running with 12 threads. Also, we show that the DTC's online profiling mechanism has low overhead (11% in the worst case).

The main contributions of this paper are:

- We show that statically coalescing transactions performs sub-optimally – even degrading the performance – when the running conditions of the program change (e.g. thread count, abort rate).
- We introduce *Dynamic Transaction Coalescing* (DTC), a compile-time and run-time technique that improves transactional throughput of the loops executing transactions.
- We show that DTC dynamically selects the best transaction granularity (to improve transactional throughput) and adapts it accordingly to the program behavior.

The remainder of this paper is organized as follows: in Section 2 we motivate our work; in Section 3 we explain the design and implementation of the *dynamic transaction coalescing* technique; in Section 4 and Section 5 we show experimental results; in Section 6 we compare our work to previous studies; finally, we conclude in Section 7.

2. BACKGROUND AND MOTIVATION

Prior works in TM has shown that starting and committing transactions can incur high overheads [15, 18, 4, 14]. These overheads may account for even more than 70% of the total execution time in extreme cases [4]. In response, researchers proposed mechanisms to coalesce transactions in order to minimize the associated overheads of starting and committing transactions. In this section we provide background information about Stat-TC [14], the state-of-the-art static transaction coalescing technique, and we demonstrate the short-comings of this approach that motivate our work.

2.1 Background - Transaction Coalescing

Static Transaction Coalescing (Stat-TC) is a profile-guided compiler optimization that aims to reduce the overheads of starting and committing a transaction [14]. Stat-TC coalesces transactions statically at compile-time. First, the compiler profiles the single-threaded version of the application and collects profiling information (transaction length, the transaction frequency, and the transaction distance). Based on this information, the compiler *statically* coalesces two or more transaction instances into one larger transaction. The number of transactions that are included in the final coalesced transaction define the *transaction coalesce factor* (TC factor)[2].

```
for (i = 0; i < N; i++) {
  TX_START();
  if (rand() < 0.5)
    ht.update();
  else
    ht.lookup();
  TX_COMMIT();
}
```

(a) Hash-table main loop

```
for (i = 0; i < N/2; i+=2) {
  TX_START();
  if (rand() < 0.5)
    ht.update();
  else
    ht.lookup()
  // no TX_COMMIT();

  // no TX_START();
  if (rand() < 0.5)
    ht.update();
  else
    ht.lookup();
  TX_COMMIT();
}
```

(b) Hash-table with unrolled main loop and coalesced transaction

Figure 1: **Hash-table main loop:** (a) For-loop with one transaction in the loop body; (b) For-loop unrolled with the unroll factor 2. The body of the unrolled loop contains the coalesced transaction, which contain the two initial transactions.

Figure 1 shows the hash-table benchmark with statically coalesced transactions. The main loop of the benchmark has one transaction that repetitively executes either the `lookup()` or the `update()` function based on a probability. Assuming that the profiling step indicated as optimal a TC factor of 2, Stat-TC unrolls the loop once and removes the unnecessary extra calls to `TX_START()` and `TX_COMMIT()` functions, generating a single coalesced transaction. Hence, the resulting code has lower transactional overhead than the two original transactions.

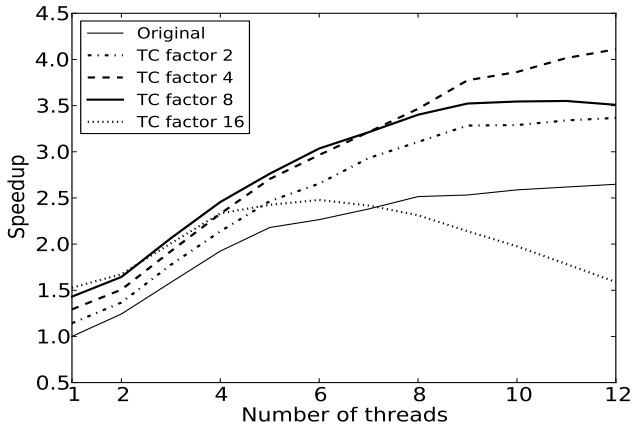
2.2 Motivation

Stat-TC generates profiling information based on *single-threaded* execution and uses this information to coalesce transactions at compile time. Such a static approach ignores the behavioral changes of the application as, for example, the number of threads changes. Thus, coalescing transactions statically may result in sub-optimal execution and even performance degradation.

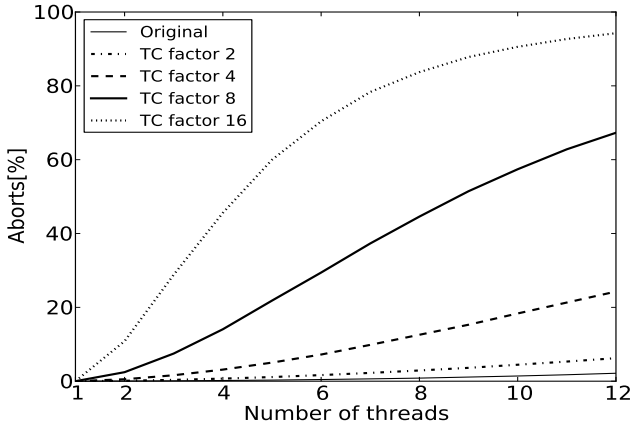
To better understand the limitations of coalescing transactions statically, we use the hash-table benchmark (Figure 1.a) executing the `update()` function with a 50% probability. We first run the 'original' benchmark where the main loop executes only one transaction. Then, we instruct the compiler to unroll the loop and to coalesce transactions for various TC factors generating different executables with TC factors equal to 2, 4, 8 and 16.

Figure 2a shows the speedup for the hash-table benchmark. We observe that the best TC factor changes with the number of threads. More specifically, when the number of threads is small (1 or 2) the best TC factor is 16, when the number of threads is modest (3-6) the best TC factor is 8, and when the number of threads is larger than 7 the most suitable TC factor is 4. Moreover, the results show that Stat-TC can increase the performance when the TC factor is small (2 or 4) but may also degrade performance when the TC factor is large (16) due to high abort rates as we explain

[2]Every coalesced transaction has its own TC factor. It is possible that different parts of the program contain transactions with different TC factor



(a) Speedup



(b) Abort rate (%)

Figure 2: **Hash-table speedup and aborts** with update rate 50%. Hash-table has 1024 entries with 2048 possible key values.

next.

Transaction Coalescing increases the length of the transaction, which in turn may increase the abort rate and decrease overall performance. Figure 2b shows the abort rate for the hash-table benchmark. For TC factor 2 and 4, the abort rate stays below 25% even when the number of threads increases up to 12. For TC factors 8 and 16 the number of aborts increase rapidly (by more than 50% when the thread number is higher than 8). Based on these results, we conclude that higher TC factors are likely to increase the abort rate. It is also difficult to predict the speedup from the abort rate and vice versa.

Summarizing, this simple example shows that there cannot be a single TC factor performing best for all thread counts. Thus, the approach of Stat-TC to statically generate code with fixed TC factors performs sub-optimally when the number of threads changes. Also, aggressively coalescing transactions based on profiling the single-threaded application may degrade the performance due to an increased abort rate. Since Stat-TC lacks a dynamic feedback mechanism, it cannot adapt to changing abort rates caused by a large number of running threads. In the following section we introduce DTC, a technique that dynamically adjusts the transaction coalesce factor in the presence of changing conditions of the application.

3. DYNAMIC TRANSACTION COALESCING

In this paper we introduce *Dynamic transaction coalescing* (DTC), a compile-time and run-time technique that improves transactional throughput of the loops executing transactions. At compile-time, DTC unrolls the loops containing the transaction and applies the transaction coalescing algorithm on the transactions contained in the loop body. Instead of generating only one unrolled loop instance, DTC generates several loop instances where each instance is unrolled with a different unroll factor. Consequently, each loop instance contains one coalesced transaction with a different 'TC factor'. At run-time, DTC profiles the application and dynamically selects which loop instance to execute. This way, DTC dynamically selects and executes coalesced transactions with the most appropriate TC factor.

Our implementation of DTC has three main components: (i) loop replacement and unrolling, (ii) transaction coalescing algorithm, and (iii) run-time transaction profiling. These components are related in the following way:

- **Loop replacement and Unrolling (LU):** At compile-time, LU replaces the loops that contain transactions with multiple loop instances where each loop instance is unrolled with a different unroll factor.
- **Transaction Coalescing Algorithm (TCA):** At compile-time, TCA coalesces transactions in each loop instance generated by the LU step. As a result, each loop instance contains one coalesced transaction that has its transaction coalesce factor equal to the loop unroll factor.
- **Run-time Transaction Profiling (RTP):** At compile-time, RTP inserts additional code that profiles transactions. At run-time, the profiling code measures transactional throughput and dynamically selects the TC factor in order to improve the throughput.

In the remaining of this section, we explain in more detail the LU, TCA, and RTP components of DTC and how they interact with each other.

3.1 Loop Replacement and Unrolling (LU)

At compile-time, LU replaces the loops containing transactions with multiple loop instances where each loop instance is unrolled with a different unroll factor. To better understand LU's code transformations, we use a simple while loop[3] example (Figure 3.a). The loop body contains one transaction surrounded with `<pre>` and `<post>` code that is not a part of the transaction. LU transforms the while loop into a switch statement (Figure 3.b) which contains several unrolled instances of the loop. Each `case 2, 4, 8, 16` has the loop body unrolled with unroll factors 2, 4, 8, 16, respectively, while the `default` case statement contains the original (non-unrolled) version of the loop body.

The resulting code has two main characteristics: (i) the case statements contain unrolled loop bodies that are suitable for transaction coalescing and (ii) the code can dynamically select which case statement to execute by changing the value of the `unroll_factor` variable.

[3]Every for-loop, while-loop, and do-while-loop can be transformed into an equivalent while-loop.

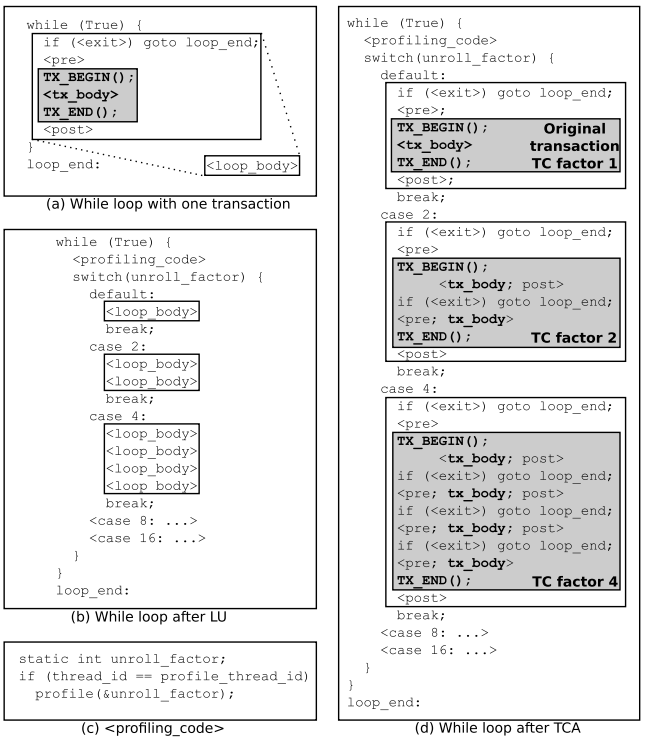


Figure 3: **While loop with transaction - Creating coalesced transactions.** Example of a while loop (a) with one transaction in the loop body. The compiler transforms the loop applying *loop replacement and unrolling* giving an expanded loop (b). The expanded loop contains several code paths (default, case 2, 4, 8, 16) containing while loop bodies unrolled with a different loop unroll factor (white rectangles). Next, the compiler transforms the expanded loop, applying the *transaction coalescing algorithm* on the code in each 'case block', giving several coalesced transactions (gray rectangles (d)). In the end, the compiler inserts <profiling_code> (c) at top of the while loop.

3.2 Transaction Coalescing Algorithm (TCA)

The compiler takes the transformed while loop from the LU step (Figure 3.b) and applies the Transaction Coalescing Algorithm (TCA) [14] on each case block (the code between `case` statement and the following `break` statement). TCA coalesces transactions into one larger transaction in the following way. TCA takes the Abstract Syntax Tree (AST) of each case block, finds the transactions nodes, finds the common parent node of all transactions, encloses the parent node in a transaction, and removes original transaction nodes from the AST. By design, each case statement consists of an unrolled loop body that contains a single coalesced transaction, where each loop unroll factor is equal to the TC factor of the corresponding coalesced transaction. The resulting code (Figure 3.d) contains several coalesced transactions where each transaction has a different TC factor. If the loop body contains two or more transactions, TCA coalesces all the transactions by finding the common parent node of all transactions.

It is important to stress that TCA correctly coalesces

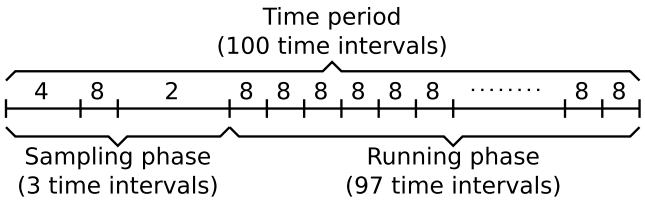


Figure 4: **Time period** consists of time intervals where every time interval lasts for 256 transaction commits. The sampling phase has 3 time intervals and the running phase has 97 time intervals. This example shows a sampling phase that uses unroll factors of 4, 8, and 2. In this example, the time interval with unroll factor 8 was measured to be the fastest, therefore, the unroll factor 8 is used for all the time intervals in the running phase.

smaller transactions into one larger transaction[4]. Despite the existence of the `goto` statements, the compiler handles the coalesced transaction correctly because it inserts additional code that commits the transaction on every code path that escapes out of the coalesced transaction.

3.3 Run-time Transaction Profiling (RTP)

At the beginning of the transformed loop, RTP inserts the `<profiling_code>` (Figure 3.c) that calls the `profile()` function. This function measures the transactional throughput of the loop for various TC factors, and selects which coalesced transaction to execute by updating the `unroll_factor` variable. Since all threads read the `unroll_factor`, all of them execute the same code path with the same TC factor. In order to reduce the profiling overhead, we only select one thread to be the profiling thread (`thread_id == profile_thread_id`).

To explain better the implementation of the `profile()` function, we introduce the following terms:

- **Time interval** is the time it takes to commit 256 transactions. The `unroll_factor` variable remains constant during the time interval.
- **Time period** consists of 100 consecutive time intervals (Figure 4).
- **Profiling phase** consists of the first 3 time intervals of a time period.
- **Running phase** consists of the remaining 97 time intervals of a time period. All 97 time intervals in the running phase use the same `unroll_factor` value.

The `profile()` function periodically estimates the transactional throughput by measuring the duration of the time intervals in the sampling phase. For each time interval of the profiling phase, `profile()` sets a different unroll factor (current unroll factor, next larger unroll factor, next smaller unroll factor), and selects the best unroll factor for the next running phase. Since the number of committed transactions is fixed, the throughput is inversely proportional to the interval length. So, the best unroll factor has the shortest time interval.

Our profiling approach introduces two sources of overhead: (i) instrumentation overhead and (ii) phase check

[4]For the implementation and the correctness of TCA, please refer to [14].

Benchmark	Input arguments
Hash-table	fixed update rates 1%, 20%, 50%
Red-black tree	fixed update rates 1%, 20%, 50%
Hash-table + phases	4 phases with update rates 1%, 20%, 50%, 100%
Red-black tree + phases	4 phases with update rates 1%, 20%, 50%, 100%
Vacation qpt=1,2,3	-n1,2,3 -q90 -u98 -r1048576 -t4194304
SSCA2	-s20 -i1.0 -u1.0 -l3 -p3
CLOMP-TM; No conflicts	-1 1 x1 d6144 128 Stride1 3 1 0 6 1000
CLOMP-TM; Rare conflicts	-1 1 x4 d6144 128 Adjacent 3 1 0 6 1000
CLOMP-TM; High conflicts	-1 1 64 100 128 firstParts 3 1 0 6 1000

Table 1: **Benchmark input parameters.**

overhead. The instrumentation overhead exists because the transformed code is larger than the original code but this overhead is negligible (less than 0.1%). The phase check overhead exists because after every time period (100 time intervals) the `profile()` function changes the unroll factor to check if there is a better one available using hill climbing technique. So, even if the benchmark executes with the best unroll factor, the profiling executes for two time intervals with "non optimal" unroll factors during the sampling phase. Changing the unroll factor in the sampling phase creates interference in the program execution since it affects the execution of all the threads running the program. To minimize the interference, we empirically selected profile parameters values (256 commits, 100 time intervals, and 3 profiling intervals) in order to provide a good balance between performance and interference.

3.4 Discussion

In this paper we propose DTC as the combination of compile-time and run-time techniques where we generate alternative code paths at compile-time and we select the code paths to execute at run-time. We follow this approach because we target the C/C++ programming environments that do not support run-time code generation. Thus we need to generate the different code paths at compile-time. However, implementing DTC for other programming environments that support run-time code generation, e.g. JVM and .Net, may allow the use of just-in-time (JIT) compilation for generating the alternative code paths. This is a potential direction for future work.

3.5 Summary

We have shown how dynamic transaction coalescing (DTC) transforms the loop containing transaction/s. DTC has three main components: (i) loop replacement and unrolling (LU), (ii) transaction coalescing algorithm (TCA), and (iii) runtime transaction profiling (RTP). LU generates several unrolled loop instances, TCA coalesces transactions in the unrolled loop instances, and RTP profiles the loop execution and updates the unroll factor. Because the loop unroll factor is equal to the transaction coalesce factor, the profiling code dynamically chooses the best TC factor to increase transactional throughput.

4. EVALUATION METHODOLOGY

We perform the experiments on a Sun Fire x4140 system equipped with two Six-Core AMD Opteron 2427 (12 cores in total), with 32GiB RAM, and running Linux 2.6.32-5. We compile the applications with GCC 4.7 which includes Transactional Memory support and link them with TinySTM[7] 1.0.3. We manually implement the DTC compiler pass and use GCC as a backend.

4.1 Benchmarks

We evaluate the effectiveness of DTC using 2 micro-benchmarks (hash-table and red-black tree) and 3 well-known benchmarks that are used in TM research (Vacation [10], SSCA2 [3], and CLOMP-TM [13]). We select these benchmarks because they cover different application domains: (i) hash-table and red-black tree are used ubiquitously in programs, (ii) Vacation mimics a travel reservation application powered by an in-memory database, (iii) SSCA2 mimics applications operating over large directed, weighted, multi-graphs (e.g. social network graphs and page-rank), and (iv) CLOMP-TM mimics large scale multi-physics applications used in high performance computing. Finally, we use the input parameters specified on each of the benchmark documentation (Table 1).

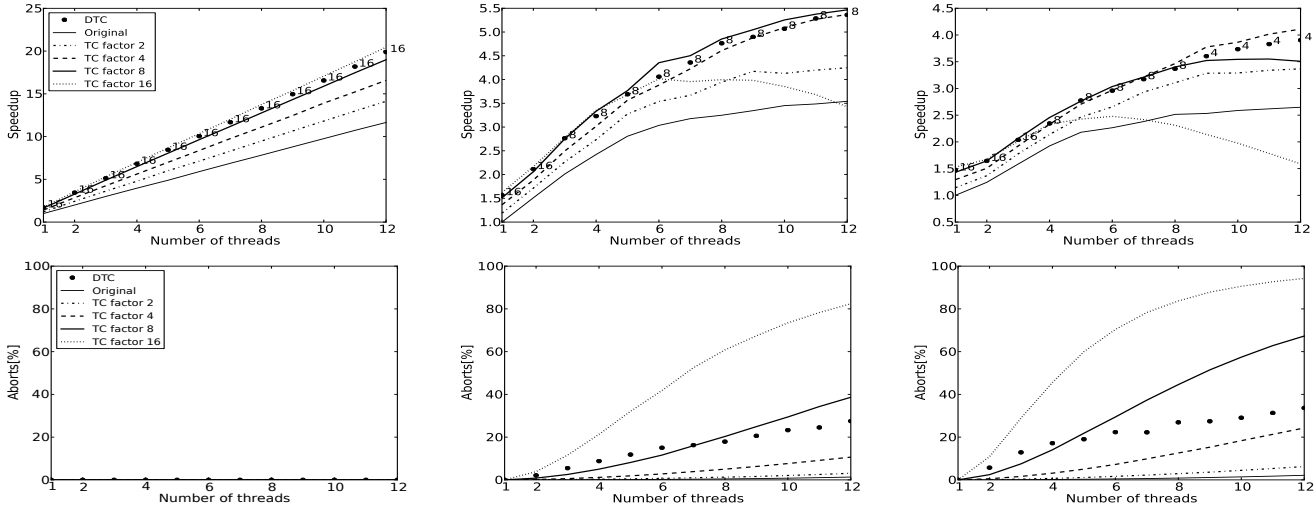
All of the benchmarks have a similar program structure which consist of an initialization section, an execution section, and a finalization section. The initialization sets up the data structures necessary for program execution, the execution section contains the parallel region that runs the transactions in benchmarks' main loop, and finalization section checks the results consistency. All transactions run in the benchmarks' main loop. Next we briefly describe the benchmarks used in this paper.

Hash-table and red-black tree execute repetitively the transactional `lookup()`, `add()`, or `remove()` functions. Each function operates on shared data stored in the hash-table or the red-black tree. We control the number of updates (`add()` and `remove()`) relative to the total number of operations (`lookup()`, `add()`, and `remove()`) with the *update rate* parameter.

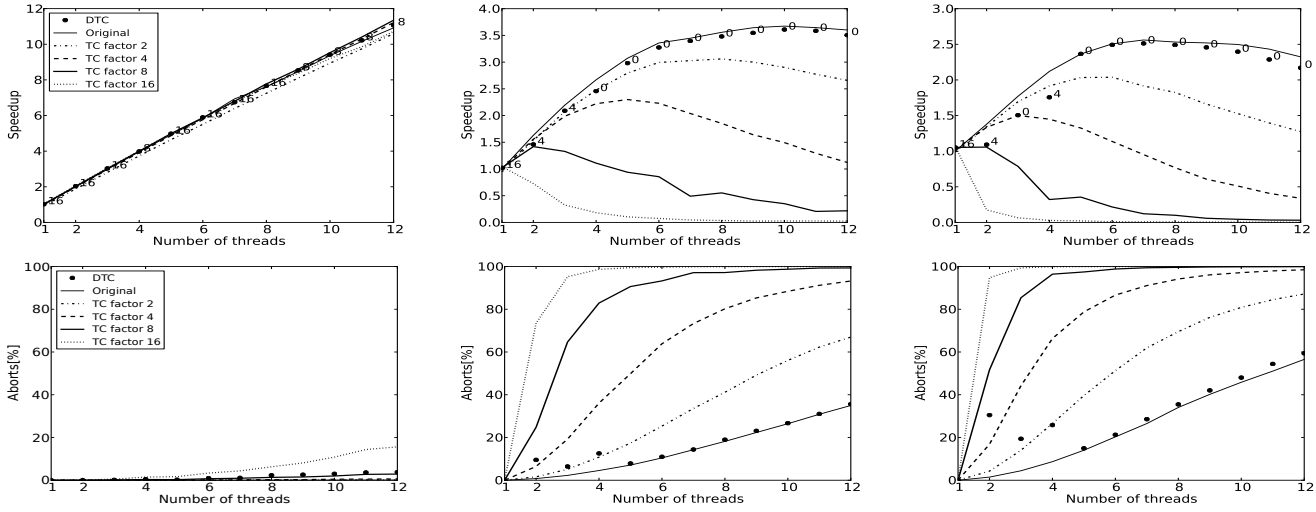
Vacation [10] implements a travel reservation system powered by an in-memory database. Several client threads interact with the database through transactional requests. The performance is measured as the number of served client requests per second. Zyulkyarov et al. [19] showed that the performance bottleneck in Vacation is the red-black tree that is used as database storage and suggested replacing it with a hash-table. Following their suggestion, Stipic et al. [14] replaced the red-black tree with a hash-table and showed that these modifications improve the performance. For our evaluation, we use this improved version of the benchmark.

SSCA2 [3] is a synthetic benchmark that operates on a large, directed, weighted multi-graph. The main loop of SSCA2 traverses all the edges of the graph. The graph can be traversed in any order and the final result of the benchmark is the same; thus, transactions in the main loop execute in arbitrary order. Stipic et al. [14] improved the performance of the benchmark by modifying the main loop. Their modifications – instead of executing the transactions immediately – buffer the values of the variables used in the transactions and when the buffer gets full, execute all the transactions in a tight loop. For our evaluation we use their improved version of the benchmark.

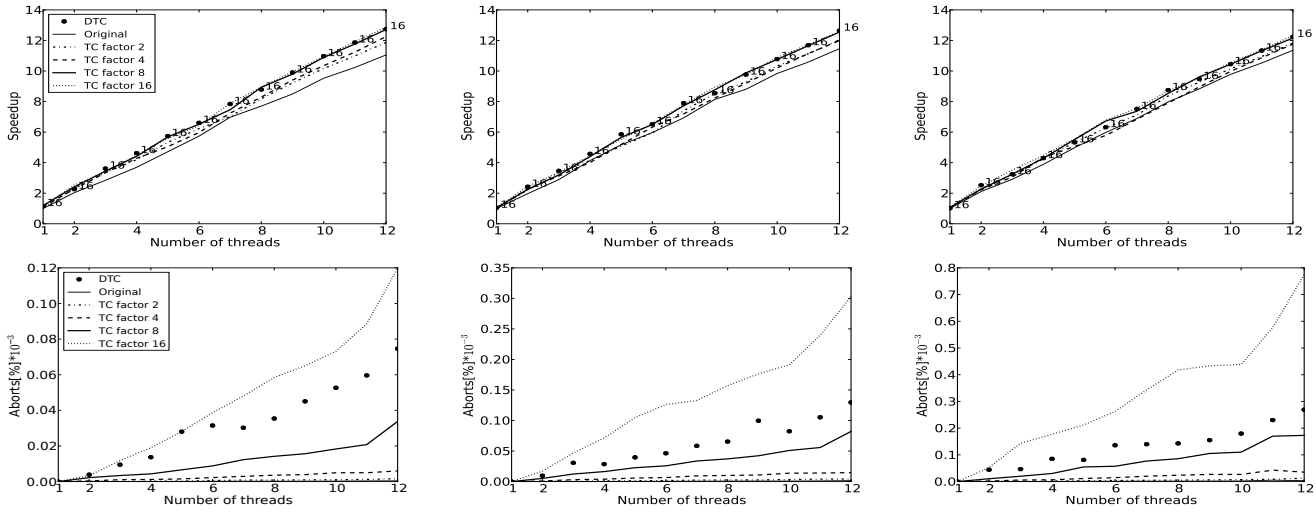
The **CLOMP-TM** [13] benchmark generates memory accesses that emulate the synchronization characteristics of HPC applications. An unstructured mesh is divided into partitions, where each partition is subdivided into zones. Threads concurrently modify these zones to update the mesh. Specifically, each zone is pre-wired to deposit a value to a set of other zones, called scatter zones, which involves (i) reading the coordinate of a scatter zone, (ii) doing some computation, and (iii) depositing the new value back to the scatter zone. Since threads may be updating the same zone,



(a) Hash-table; ur = 1% (b) Hash-table; ur = 20% (c) Hash-table; ur = 50%

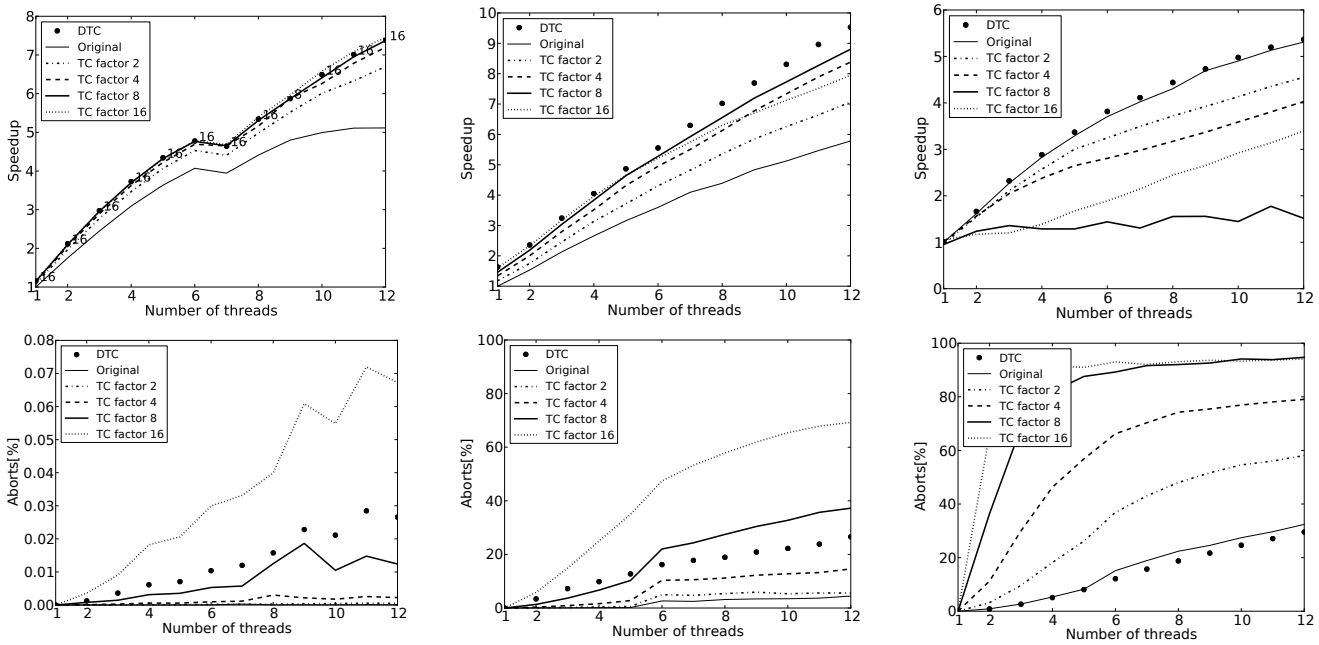


(d) Red-black tree; ur = 1% (e) Red-black tree; ur = 20% (f) Red-black tree; ur = 50%



(g) Vacation; qpt = 1 (h) Vacation; qpt = 2 (i) Vacation; qpt = 3

Figure 5: **Hash-table, red-black tree, and Vacation:** Throughput and abort rate for different numbers of *update rate* and *queries per transaction* (qpt).



(a) SSCA2 (b) Hash-table with phased execution (c) Red-black tree with phased execution

Figure 6: **SSCA2, hash-table, and red-black tree.** Hash-table and red-black tree with phased execution. Each phase last for a quarter of the total execution time and each phase has update rate (ur) of 1%, 20%, 50%, and 100%.

value deposits need to be synchronized. Conflict probability can be adjusted by controlling how the zones are wired, and by changing the number of scatters per zone; the amount of work done in a critical section can also be adjusted. For our evaluation we use the "Large TM" part of CLOMP-TM , that uses transactions to synchronize critical sections.

4.2 Metrics

Our results show the performance of applying DTC on the benchmarks' main loop. For all benchmarks we present two plots, one for the speedup and one for the abort rate. We normalize the speedup results to the benchmark running with one thread. In the speedup plots, we also show the dominant TC factor that DTC chooses for that benchmark run (a number next to the small circle in the figures). In the abort rate plots, the abort rate of transactions is plotted in terms of percentage of total executed transactions. Finally, we compare DTC with Stat-TC, and we show the results for different TC factors (2, 4, 8, and 16). We do not plot results for TC factors larger than 16 because they do not provide any performance improvements.

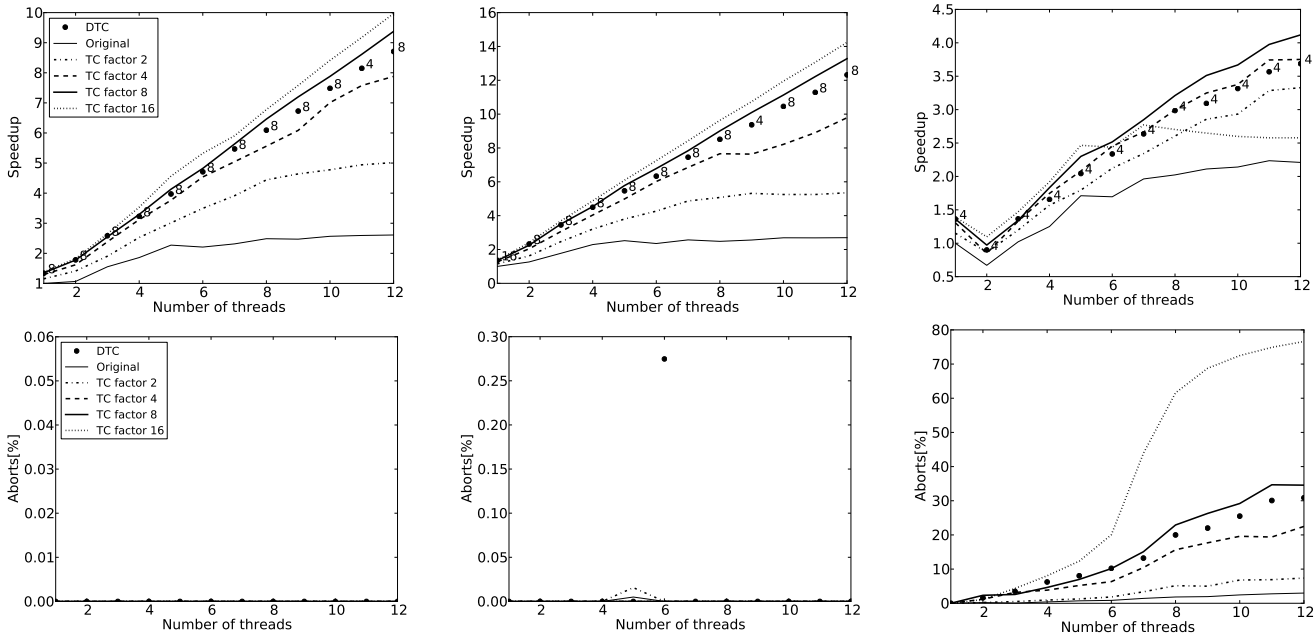
5. RESULTS

In this section we show how DTC affects the performance of each benchmark, we compare DTC with the performance of Stat-TC, and we also analyze the performance of both mechanisms in the presence of phased execution in the benchmarks.

5.1 Hash-table

The hash-table benchmark with 1% update rate (Figure 5.a) contains short transactions and exhibits a negligible abort rate. Such characteristics make the benchmark well suited for transaction coalescing techniques. The results show that DTC improves the application performance by 62.9% over the original version when running with 12 threads. Compared to Stat-TC, DTC performs slightly worse (7.7%) although DTC chooses correctly the dominant TC factor. This performance difference is due to: (i) the profiling overheads of DTC, and (ii) the suitability of the benchmark to static transaction coalescing since it exhibits well-expected behavior without any phases or abort rate changes. Still, we observe that DTC improves significantly the application performance and performs close to Stat-TC even for the case that dynamically coalescing transactions is not necessary.

Figures 5.b and 5.c show the results for the hash-table with update rates of 20% and 50%. We see that DTC consistently improves the application performance for all thread counts and always follows the best case of Stat-TC. We make the following important observations. First, in Stat-TC there is no single best TC factor that performs best for all thread counts. For example, Figure 5.c shows that the best TC factor is 16 for up to 3 threads, 8 for up to 8 threads, and 4 for up to 12 threads. Second, aggressively increasing the TC factor may degrade the application performance significantly. For the case of 50% update rate and with thread count larger than 8, Stat-TC with TC factor 16 performs worse than the original benchmark because of the high abort rate that reaches more than 80%. Unlike Stat-TC, DTC dynamically identifies the best TC factor taking the abort rate into consideration. Due to profiling overheads, DTC does not attain the best performance but



(a) CLOMP-TM; No conflicts

(b) CLOMP-TM; Rare conflicts

(c) CLOMP-TM; High conflicts

Figure 7: **CLOMP-TM**

is just 5.2% slower than the best TC factor for all thread counts.

5.2 Red-black tree

Figures 5.d, 5.e and 5.f show the results for the red-black tree benchmark with update rates 1%, 20% and 50% respectively. For update rates other than 1%, we observe that the benchmark is unsuitable for static transaction coalescing since the performance only degrades. For example, running the benchmark under Stat-TC with more than 4 threads and TC factors of 4 and more degrades performance significantly. This drastic performance degradation is due to the extremely high abort rate that reaches up to 99.9%. On the other hand, DTC does not degrade transaction throughput and follows the performance of the original version, having only 6.6% performance loss due to the profiling overhead (12 threads, update rate 50%). Hence, DTC does not suffer from Stat-TC's performance degradation when coalescing is not useful.

5.3 Vacation & SSCA2

Vacation (Figures 5.g, 5.h and 5.i) and SSCA2 (Figure 6.a) exhibit low abort rate (less than 1%) and demonstrate that coalescing techniques can improve performance. For Vacation with 12 threads and qpt=1, DTC and Stat-TC perform similarly well and improve the performance of the original benchmark by 15.3% and 16.4%. For SSCA2 with 12 threads, DTC and Stat-TC improve the performance of the original benchmark by 44.4% and 45.8%. Again, we conclude that DTC significantly improves performance of the applications, and closely follows the performance of Stat-TC in case there is no need for dynamically changing the TC factor, and meanwhile the profiling overhead remains low.

5.4 CLOMP-TM

Figure 7 shows the performance of the CLOMP-TM benchmark. TC factor 8 and 16 perform the best when the conflicts are rare (Figure 7.a and Figure7). DTC selects TC factors 4 and 8 for the execution but the best performing TC factors are 8 and 16. DTC is not able to 'find' the best TC factor due to profiling interference. Still, DTC does not suffer from TC's performance degradation (Figure 7.c with TC factor 16) and DTC performs just 11% worse than the best TC factor, while still being substantially better than the original version.

5.5 Phased execution

Up to this point, we analyzed the performance of DTC and we compared it to that of Stat-TC. However, we performed our evaluation on various benchmarks that do not have phased execution. In other words, the running conditions (thread count) or the program behavior (e.g. abort rate) did not change dynamically during the execution. In order to compare the performance of DTC and Stat-TC in changing conditions, we modify the hash-table and red-black tree benchmarks and introduce phased execution. We introduced four phases in the benchmark execution where each phase has the same duration and executes with different update rate parameters (ur = 1%, 20%, 50%, 100%).

Figure 6.b shows the speedup[5] and the abort rate for the hash-table benchmark with phases. We observe that DTC improves the performance by 64.7% for the hash-table benchmark, outperforming Stat-TC by 8.3%. The reason is

[5]We do not plot the dominant TC factor for DTC because it changes during the program execution.

that there is no single TC factor that performs best for all execution phases. Thus, Stat-TC executes various phases with sub-optimal TC factor. In contrast, DTC dynamically adapts to the program execution and selects the best TC factor improving application performance.

Figure 6.c shows the results for the red-black tree benchmark with phased execution. We observe that DTC performs equally or slightly better than the original version. DTC's profiling mechanism does not identify any potential benefits through coalescing, and executes most of the time without any coalesced transactions (TC factor 0). On the other hand, blindly increasing the TC factor – as Stat-TC does – would lead to increased abort rate and degrades the application performance by 3.5x with 12 threads compared to the original version.

5.6 Overview

We make the following observations to summarize this section:

- DTC is able to considerably increase the application performance over the original version (e.g. hash-table, Vacation, and SSCA2).
- DTC follows closely the performance of Stat-TC with the best TC factor for those applications that do not exhibit high abort rates. The performance gap is due to the profiling mechanism, but still remains lower than 11% in the worst case.
- DTC outperforms both the original version and Stat-TC with the best TC factor when programs exhibit phased execution (e.g., hash-table and red-black tree with phases).
- DTC does not introduce performance degradation when transactional coalescing is not beneficial, while Stat-TC may harm the performance. (e.g., red-black tree).

6. RELATED WORK

In this section we discuss the related work in the area of coalescing transactions, and explain how our approach differs from it. We also present briefly other optimization techniques for TM systems.

6.1 Coalescing Transactions

Chung et al. [4] leverage STM to provide thread-safe dynamic binary translation (DBT). The transactions aim to provide atomic access to the data and metadata during the program execution. To reduce the overheads of starting and committing a transaction, the authors generate transactions at the basic block level (instead of instruction level) and they further apply transaction coalescing. The decision of coalescing transactions depends only on the amount of work per transaction, i.e. the number of instructions. However, this approach applies only to DBT and does not take into account the transaction throughput and abort rate. Thus, aggressively coalescing transactions without such information may degrade the application performance [4]. In contrast, DTC differs in two ways: (i) DTC generates transactions of different size at compile-time, (ii) DTC uses run-time profiling information to choose dynamically the transaction size. In this way, DTC is able to increase or decrease the coalesce factor correspondingly and to improve application performance.

Yoo et al. [18] evaluate the performance of Intel Transaction Synchronization Extensions (TSX) that implement best effort hardware transactional memory. To quantify the transactional overheads, the authors analyze the performance benefits of static and dynamic transaction coalescing[6] and show that transaction coalescing improves performance. However, their implementation of transaction coalescing lacks the profiling mechanism that identifies the optimal transaction granularity at run-time. Their dynamic coalescing combines multiple dynamic instances of the same transactional region. In contrast, DTC generates several static instances of the same transactional region and uses online profiling to dynamically select the region that increases the throughput.

Stipic et al. [14] introduce a transaction coalescing technique which we call as Stat-TC in this paper for the sake of readability and clarity. Stat-TC is a profile-guided compiler optimization technique that reduces the overheads of starting and committing a transaction by coalescing two or more small transactions into one large transaction. Stat-TC consists of a pre-compilation profiling step that collects information about transactions (transaction length, transaction frequency, and transaction distance) and uses this information to coalesce transactions statically (at compile-time). The limitation of this approach is that coalescing transactions statically cannot adapt to dynamic program behavior. DTC improves Stat-TC by removing the dependency on the pre-compilation profiling step and by dynamically selecting the most appropriate number of coalesced transactions at run-time.

6.2 Other Optimization Techniques

Many optimizations have been proposed for compilers and runtime systems to reduce the overheads of STM [16, 17, 6, 2]. Wang et al. [16] provide compiler optimizations for eliminating unnecessary read/write barriers (read after write barrier, barriers on local variables, etc.) and register checkpointing. Afek et al. [2] propose static analysis and code motion to decrease the number of memory accesses inside transactions. Wu et al. [17] use compiler optimizations on statistics collected at run-time to eliminate redundant barriers and checkpointing. Dragojevic et al. [6] present run-time and compiler optimizations to identify transaction-local stack and heap allocation, and an API for annotating thread-local and read-only memory regions. DTC is an orthogonal mechanism to all these techniques and can be used to further improve the performance of STM systems.

Adl-Tabatabai et al. [1] use compiler and run-time optimizations for transactional memory language constructs. They use a JIT compiler to reduce the overheads of STM. The JITting mechanism could be used also to dynamically generate unrolled loops that benefit from dynamic transaction coalescing. Wang et al. [15] evaluate the performance of the BlueGene/Q machine that provides HTM support. BlueGene/Q maintains speculative states in the L2 cache and uses software register checkpointing with the `setjmp()` function. Even though BlueGene/Q has support for best effort hardware TM, the implementation has issues with small transactions. The authors admit that the software regis-

[6]Transaction coalescing is called coarsening in Intel's terminology.

ter checkpointing has significant overhead for small transactions. Thus we believe that DTC can act complementary in reducing these overheads.

7. CONCLUSIONS

In this paper we introduced *dynamic transaction coalescing* (DTC), a compile-time and run-time technique that improves transactional throughput of the loops executing transactions. We explained how DTC transforms the loops and how DTC generates coalesced transactions of different sizes. Also, we explained the implementation of DTC's online profiling and showed how profiling helps to find the best transaction granularity that increases the throughput. We evaluated DTC using 3 benchmarks(SSCA2, Vacation, and CLOMP-TM) and 2 micro-benchmarks (hash-table and red-black tree).

We show that DTC improves the performance of SSCA2, Vacation, CLOMP-TM, and hash-table by 44.4%, 45.8%, 66.9%, and 62.9% respectively (running with 12 threads and having a high conflict rate). We also show that DTC performs like the statically selected best transaction coalesce factor, and that DTC's online profiling has small performance overhead. The overhead is 6.6% in hash-table and red-black tree; 11% in CLOMP-TM; and less than 1% in SSCA2 and Vacation. Finally, we show that DTC performs better than Stat-TC when phases are present improving the performance of hash-table and red-black by 8.2% and 1.1% with respect to Stat-TC with the best TC factor.

8. REFERENCES

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

[2] Y. Afek, G. Korland, and A. Zilberstein. Lowering STM overhead with static analysis. *Languages and Compilers for Parallel Computing*, pages 31–45, 2011.

[3] D. Bader and K. Madduri. Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. *High Performance Computing–HiPC 2005*, pages 465–476, 2005.

[4] J. Chung, M. Dalton, H. Kannan, and C. Kozyrakis. Thread-safe dynamic binary translation using transactional memory. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 279–289. IEEE, 2008.

[5] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Distributed Computing*, pages 194–208. Springer, 2006.

[6] A. Dragojevic, Y. Ni, and A. Adl-Tabatabai. Optimizing transactions for captured memory. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 214–222. ACM, 2009.

[7] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246. ACM, 2008.

[8] T. Harris, J. Larus, and R. Rajwar. Transactional memory. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010.

[9] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, 1993.

[10] C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46. IEEE, 2008.

[11] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, et al. Design and implementation of transactional constructs for c/c++. In *ACM Sigplan Notices*, volume 43, pages 195–212. ACM, 2008.

[12] C. J. Rossbach, O. S. Hofmann, and E. Witchel. Is transactional programming actually easier? In *ACM Sigplan Notices*, volume 45, pages 47–56. ACM, 2010.

[13] M. Schindewolf, B. Biliari, J. Gyllenhaal, M. Schulz, A. Wang, and W. Karl. What scientific applications can benefit from hardware transactional memory? In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.

[14] S. Stipic, V. Smiljkovic, A. Cristal, O. Unsal, and M. Valero. Profile-guided transaction coalescing - lowering transactional overheads by merging transactions. *The ACM Transactions on Architecture and Code Optimization, TACO 2014*, 2014.

[15] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of blue gene/q hardware support for transactional memories. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 127–136. ACM, 2012.

[16] C. Wang, W. Chen, Y. Wu, B. Saha, and A. Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *Code Generation and Optimization, 2007. CGO'07.*, pages 34–48. IEEE, 2007.

[17] P. Wu, M. Michael, C. von Praun, T. Nakaike, R. Bordawekar, H. Cain, C. Cascaval, S. Chatterjee, S. Chiras, R. Hou, et al. Compiler and runtime techniques for software transactional memory optimization. *Concurrency and Computation: Practice and Experience*, 21(1):7–23, 2009.

[18] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of intel transactional synchronization extensions for high-performance computing. In *Supercomputing 2013.*, 2013.

[19] F. Zyulkyarov, S. Stipic, T. Harris, O. Unsal, A. Cristal, I. Hur, and M. Valero. Discovering and understanding performance bottlenecks in transactional applications. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 285–294. ACM, 2010.