# Synchronization Aware Conflict Resolution for Runtime Monitoring Using Transactional Memory

Chen Tian, Vijay Nagarajan, Rajiv Gupta
Dept. of Computer Science and Engineering
University of California at Riverside
{tianc,vijay,gupta}@cs.ucr.edu

## ABSTRACT

There has been significant research on performing runtime monitoring of programs using dynamic binary translation (DBT) techniques for a variety of purposes including program profiling, debugging, and security. However, such software monitoring frameworks currently handle only sequential programs efficiently. When handling multithreaded programs, such tools often encounter racing problems and require serialization of threads for correctness. The races arise when application data and corresponding meta data stored in DBT tools are updated concurrently. To address this problem, transactional memory (TM) was recently proposed to enforce atomicity of updating of application data and their corresponding meta data. However, enclosing legacy synchronization operations within transactions can cause livelocks, and thus degrade performance.

In this paper, we consider common forms of synchronizations and show how they can give rise to livelocks when TM is used to enable runtime monitoring. In order to avoid such livelocks, we implement a detector to dynamically detect synchronizations and use this information in performing conflict resolution. Our experiments show that our synchronization aware strategy can efficiently avoid livelocks for the SPLASH-2 benchmarks.

## General Terms

Monitoring, Transactional Memory, Conflict Resolution

## 1. INTRODUCTION

There has been significant research on performing runtime monitoring of programs using software techniques for a variety of purposes. For example, *LIFT* [12] is a software tool that perform dynamic information flow tracking to ensure that the execution of a program is not compromised by harmful inputs. In these monitoring applications, original instructions that manipulate application data are accompanied by instrumented instructions that manipulate meta data associated with the application data.

However, for multithreaded programs, it is essential that original application data and its corresponding meta data are manipulated atomically in order to correctly maintain the meta data values [11, 3]. Existing software monitoring schemes [11] ensure the atomicity by serializing the execution of threads. Naturally, this is not an efficient way to handle multithreaded programs. To overcome this inefficiency, the use of *transactional memory* support has been recently proposed to enforce atomicity [3]. Specifically, the original instructions that manipulate application data and the corresponding instructions that manipulate the meta data are put inside a transaction, so that TM mechanisms can automatically enforce atomicity. This obviates the need to *fully* serialize the execution of the threads and thus enables the efficient monitoring of multithreaded programs.

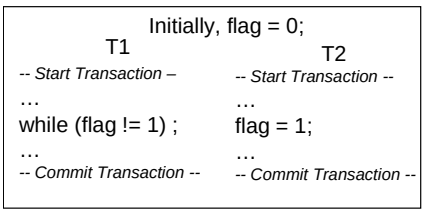


**Figure 1: A livelock caused by synchronization**

An important parameter that directly affects the efficiency of performing monitoring using TM is the length of the transactions [3]. This is because of the expensive bookkeeping tasks involved in starting and ending transactions. Thus for example, it will be inefficient to put an individual basic block of original code (and its accompanying instrumentation) within a single transaction, especially if that basic block is present in a hot loop. To alleviate this problem, two techniques are proposed in [3]: (a) creating transactions at trace granularity and (b) dynamic merging of transactions. However, employing these techniques may lead to livelocks if code that performs synchronization (from the original program) is put into a transaction. We illustrate this problem with a simple example shown in Fig. 1.

Let us assume that the *transaction start* and *transaction end* instructions are placed outside the spinning loop to prevent the creation of a transaction (T1) every loop iteration. Note that the write that sets the $flag$ (in processor 2) is also part of a transaction T2. Let us now analyze the sequence of events on a transactional memory system, more specifically the LogTM system [10]. First, T1 is created in processor 1 and is not committed until flag is eventually set. By the time processor 2 tries to set the flag, T2 in processor 2 would have

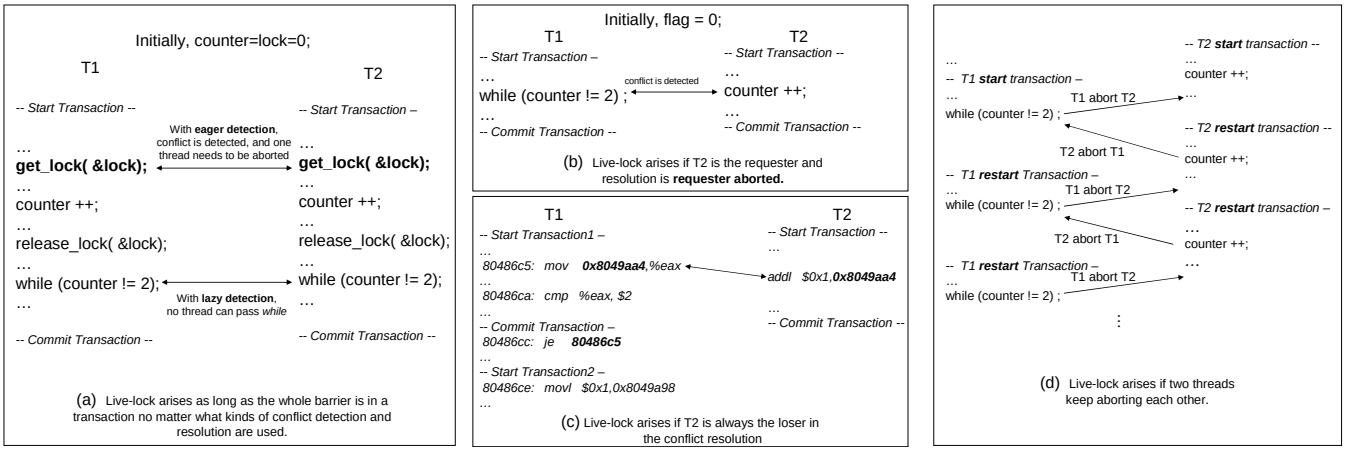


**Figure 2: Livelocks due to synchronization in Transactional Memory**

been started. Clearly T2 will conflict with the ongoing T1 as it tries to write to the location ($flag$) that has already been read by T1. Since logTM follows the policy of the requester aborting, T2, which is the requester, is aborted. When it is restarted, it still remains the requester (T1 has not committed yet), and so ends up getting aborted repeatedly, causing a livelock situation.

In this paper, we first analyze the effect of putting various synchronizations (including locks, barriers and flag synchronizations) within transactions and find that in several scenarios, a livelock can arise for a variety of TM policies (presented in section 2). We then present a synchronization aware conflict resolution strategy called **SCR**: *Synchronization aware Conflict Resolution*, where synchronization operations are dynamically identified by our synchronization detector (presented in section 3) and used to perform conflict resolution (presented in section 4). In section 5, we present experimental results that show that SCR is able to efficiently avoid livelocks and thus enables the use of TM to perform software based monitoring for multithreaded programs. We conclude in section 6.

# 2. LIVELOCKS IN TM BASED MONITORING

In this section, we will discuss livelocks scenarios for different transaction sizes.

## 2.1 TM Policies

Transactional memory systems [5] enable atomic execution of blocks of code. The three major functions of TM are *conflict detection*, *version management* and *conflict resolution*. Previously proposed TM systems can be separated into three different categories [2]: *LL* lazy conflict detection, lazy version management and committer wins; *EL*: eager conflict detection, lazy version management and requester wins; and *EE*: eager conflict detection, eager version management and requester loses.

## 2.2 Livelock Scenarios

Recall that in software based monitoring, operations in the original program are accompanied by instrumentation that manipulates the meta data, both of which need to be atomically executed. A naive way of creating transactions would be to put each original operation (that modifies application data) along with its instrumentation (that modifies meta data) into a unique transaction. However, according to Chung et al.'s recent work in [3], smaller the transaction size, more the bookkeeping overhead involved in the creation and the committing of transactions. To reduce this overhead, one possible strategy would be to create a transaction for every basic block of original program. To further increase the transaction size without introducing too many conflicts, Chung et al. proposed two techniques in [3], putting *traces* of frequently executed basic blocks into transactions and dynamically merging transactions.

Having large transactions has its own share of problems, particularly if code for performing synchronization fully fits within an individual transaction. To see why, let us consider the counter based barrier shown in Fig. 2(a), which is fully enclosed within a transaction. Let us assume that *processor 1* reaches the barrier first and thus wait for the counter to be 2 after incrementing it (transaction T1). When transaction T2 is subsequently started (when *processor 2* reaches the barrier), it conflicts with T1 since it tries to acquire the same lock that was earlier acquired by T1. If EE policy is followed, T2 being the requester, is forced to abort and this situation keeps repeating since it is always the requester. Thus a livelock situation arises. If EL policy is instead used, T2, being the requester, aborts T1. However, T1 becomes the requester now, and it consequently aborts T2. Thus, in this situation, transactions keep aborting the other repeatedly and cause a livelock. With lazy conflict detection (LL), neither of the processors can see the value of the updated value of the counter and hence land into a livelock. Thus, irrespective of the policy followed, putting barrier entirely into a transaction, causes a livelock. However, a livelock can still arise for some policies, if we put only parts of the barrier code into a transaction.

Let us consider Fig. 2(b) in which the *while loop* which spins on the variable *counter* (and its associated instrumentation) is put inside a transaction T1, and the increment of counter is in another transaction T2. Let us assume that processor 1 reaches the while loop first where it spins until the value of counter is subsequently incremented by processor 2. Thus T1 is created first and is not committed until T2 commits. However, if the EE (requester aborts) policy is followed, T2 is always aborted since it is the requester and this leads to a livelock. It is worth noting that the chances of a livelock are

significantly decreased if we had a transaction for every basic block as shown in Fig. 2(c). (In this scenario, a T1 is created every iteration of the spinning read.) This is because, to cause a livelock T2 has to be the requester every time, which is probabilistically not possible.

Let us now consider the sequence of events if EL (requester wins) policy is followed. Let us assume as before that processor 1 reaches the spin loop first and waits for the value of the *counter* to be incremented by processor 2. When processor 2 eventually tries to increment *counter*, it is a part of T2. Since T2 is a requester, and we follow the *requester wins* policy T1 is aborted and T2 is allowed to committed. However, if T1 restarts before T2 commits, then T1 now becomes the requester and consequently aborts T2. In the worst case, this situation can be repeated infinitely and thus cause a livelock. The pattern of aborts is illustrated in Fig. 2(d) and this pattern of aborts is also called FRIENDLYFIRE in [2]. It is worth noting that if we had used a strategy of one basic block per transaction as shown in Fig. 2(c), the chances of T1 restarting before T2 commits is very low, since the size of transaction T2 is small, only containing the code for incrementing *count* and its associated instrumentation.

If *lazy conflict detection* is used and the resolution is *committer wins*, the livelock can be avoided. This is because T2 will finish its transaction first and will therefore commit. When T2 is being committed, a conflict on variable *counter* will be detected and thus T1 will be aborted. Therefore, when T1 is restarted, the *counter* is already incremented, and no livelock is caused.

The various livelocks scenarios for different TM policies, and for various transaction sizes are summarized in Table 1.

| TM Policies | Barrier | Spinning-Read | Basic Block |
|---|---|---|---|
| EE | Yes | Yes | Not Probable |
| EL | Yes | Possible | Not Probable |
| LL | Yes | No | No |

**Table 1: Livelock scenarios for different TM policies and Transaction sizes**

# 3. SYNCHRONIZATION DETECTION

To solve livelock problem, HTM systems need to be aware of the synchronization operations so that the correct conflict resolution can be performed. In this section we will present a technique that can dynamically detect busy-waiting synchronizations.

## 3.1 Pattern of Busy-waiting Synchronization

The simplest mechanism to synchronize two threads is *flag synchronization*, because it does not need any special instructions such as *Test-and-Set* and *Compare-and-Swap*. Fig. 2(b), which was seen previously, shows an example. It is very clear that processor 1 performs a spinning read (*while* statement) and processor 2 performs a write on a same shared location *flag*, and these two statements are those that cause the synchronization.

Lock is another common synchronization mechanism in multithreaded programs. A classic **Test and Test-and-Set** algorithm, is shown in Fig. 3(a). To acquire the lock, each thread executes an atomic instruction *Test-and-Set* (line 2), which reads and saves the value of the lock. If the lock is available, then the *Test-and-Set* instruction returns false, which makes the winner enter the critical section. Other threads have to spin on the lock (line 3) until there is a possibility that the *Test-and-Set* instruction can succeed. The reason for the spinning on line 3 is to avoid executing the *Test-and-Set* instruction repeatedly which causes the cache invalidation overhead. From this implementation, we can see line 3 is a spinning read and line 5 is the remote write, which race with each other.

```
1 shared bool lock := false;

acquire_lock:
2       while (TS(lock)) {
3               while(lock);
4       }

release_lock:
5       lock := false;
```

(a) Test and Test-and-Set Lock

```
1  shared int counter := P;
2  shared bool sense := true;
3  private local_sense := true;

barrier:
4       local_sense := NOT sense
5       LOCK();
6               counter--;
7               if (counter = 0) {
8                       counter :=P;
9                       sense := local_sense;
10              }
11      UNLOCK();
12      while (sense != local_sense) ;
```

(b) sense-reversing counter barrier

**Figure 3: Lock and Barrier Examples**

In Fig. 2(a), we had shown the **centralized barrier**, where all threads except the last one are delayed by a spinning read on variable *counter*. In this implementation, every thread also decrements variable *counter*, causing a remote write to all earlier-arrived threads.

To make the centralized counter barrier reusable, a **sense-reversing barrier**, described in [9], is shown in Fig. 3(b). Each arriving processor decrements *count* by exclusively executing line 6 and then waits for the value of variable *sense* to be changed by the last arriving processor (line 9). Similar to the simple counter barrier, line 12 is a spinning read and line 9 is a write on variable *sense*, which is the cause of synchronization races produced due to this barrier.

Having studied the different implementations of various synchronization operations including CLH [7] and MCS lock [9] and tree barrier, we find that the *spinning read and its corresponding remote write* is a common pattern among the synchronization operations.

## 3.2 Detection of the General Pattern

Li et al. have proposed a method for spin detection [13]. However, their scheme does not identify the remote write, and thus cannot be used for HTM systems to break livelocks. In this section, we will discuss another dynamic technique to identify both the spinning read and a remote write.

**Spinning Read.** To find the spinning read, we first introduce a *load table* to maintain the information of each load instruction executed in each thread. The information includes the *PC*, the previous address accessed by the load instruction, the previous value in this address and a variable *counter*, which essentially maintains the current count of spin loop. Then, we determine if a load instruction is a spinning read in a synchronization pattern by checking the following conditions:

- this load instruction has been repeatedly executed *threshold1* number of times with the same values and addresses;

- two consecutive executions of this load are never interrupted by more than *threshold2* other load instructions;

- after the previous two conditions are satisfied, the value accessed by this load is changed by another different thread.

As we can see, the first two conditions depend on two different heuristic value respectively. If we set them too small or too big, the detection result will become inaccurate. Especially for *threshold2*, we need to consider the extra loads introduced by DBT tools. To determine these two values, we profiled the benchmarks, and found that 10 and 12 are good enough for *threshold1* and *threshold2* respectively. Thus, we chose them in our implementation.

**Remote Write.** If a thread performs a store that causes a spinning read is detected in another thread, then we identify this store as the remote write.

## 3.3 Software Implementation

The software based detector can be implemented by DBT tools. For each thread, we use a global struct variable to implement the *load table*, which contains the information about the most recent 12 loads. When a store is executed, the current thread will check other threads' *load table* to see if a synchronization pattern is detected. Specifically, if this store accesses the same location as some load, whose spin counter has reached 10, in a *load table*, and the value in this location is being changed by this store, then a synchronization pattern is identified.

The disadvantage of software implementation is the overhead. According to our experiments, the overhead on average is over 45x. Even with the optimized implementation where the loads and stores accessing stack variables are not monitored, the overhead on average is still over 9x.

## 3.4 Hardware Implementation

To reduce the overhead, we implement a hardware based detector illustrated in Fig. 4. Light shaded parts are hardware changes we need to make.

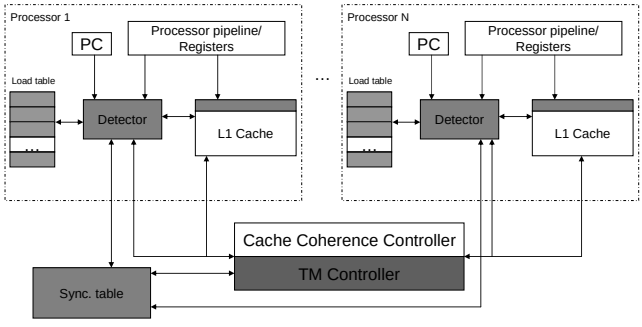


**Figure 4: Hardware Implementation**

The idea is to use a on-chip buffer to implement the *load table* for each processor, and leverage the cache coherence protocol to determine the synchronization pattern. Specifically, when a write is performed by a processor, the processor will store the *PC* into a cache coherence message, which will invalidate all copies of the updated shared variable in the caches of all other processors. When this message is received by a processor where a spinning read has been discovered, we can identify this synchronization and store the *PC* of the load and store into a synchronization table such that they can be exploited by HTM system later.

Note that in a hardware-based TM with lazy conflict detection configuration, there is no cache coherence message until the transaction is being committed. Thus, we cannot detect the synchronization pattern in time. To tackle this problem, when a spinning read is detected, we check if the variable being spun upon by the current processor is modified, i.e. in the write set, by another processor. If we find a match, we have identified a synchronization.

# 4. SYNCHRONIZATION-AWARE CONFLICT RESOLUTION

Enclosing synchronizations within transactions is the root cause of livelocks as shown in Section 2. Now that we are able to find synchronization instructions by using the detector, we need to make the HTM system be *synchronization-aware* such that a conflict due to synchronization can be correctly resolved at runtime. More specifically, the HTM system should commit the synchronization write as soon as the write instruction is detected and ensure that the spinning read from the other processor can see the updated *shared variable* quickly. In order to do that, we add the following two rules into the synchronization detector and the HTM system respectively:

- If a store instruction in a transaction is determined as a synchronization write, the detector signals the HTM system to commit this transaction immediately and start a new transaction; the detector also notifies the HTM system of the transaction containing a spinning read instruction so that it can be aborted;

- If the HTM system uses eager detection and detects a conflict, the synchronization table is checked to see if the conflict is caused by synchronization. If so, the transaction that has the synchronization write will be committed, and a new transaction will be started for this thread; the transaction that has the spinning read will be aborted. If lazy detection is used, the HTM system checks synchronization table for each store. Same action will be taken if there is a match;

The first rule ensures that if a livelock due to synchronization has already occurred or will occur, it can be broken or avoided. The second rule ensures that discovered synchronization can never cause any livelocks in the future. Intuitively, these two rules dynamically split a transaction containing a synchronization write into two smaller transactions, and give the priority to a transaction containing the synchronization write. Note that the splitting process is done by hardware, no re-instrumentation is needed.

To see how this new resolution works, let's again consider the example shown in Fig. 1(a). We assume eager detection is used and livelock will still occur because T2 keeps being aborted. However, our detector can quickly find the flag synchronization. Thus, the detector can signal the HTM system to immediately commit T2 (which performs a remote write to the variable *flag*) and abort T1. Therefore, in its next trial, T1 is able to see the updated value of *flag* and then go ahead (Fig. 5).

Similarly, our solution can also solve livelocks caused by barriers as shown in Fig. 2(a) if eager detection is used. Once

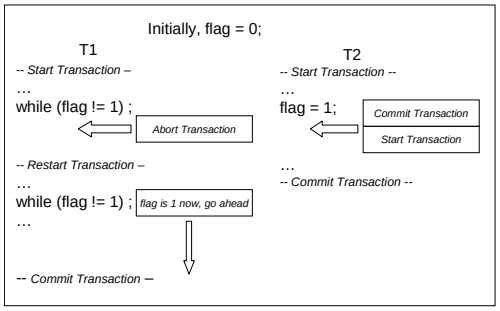


**Figure 5: Avoiding a livelock by committing a transaction containing the remote write**

the lock synchronization is detected, the transaction which released the lock will be committed. Thus, the other transaction can enter the critical section and will be committed after it updates *counter* as our detector determined that the update is a remote write. By aborting both transactions in turn, the livelocks can be eliminated permanently.

If lazy detection is used, and the spinning read on *counter* is discovered by T1's detector, the detector can periodically send a inquiry message to other processor. Once the detector finds that T2 has performed a write to *counter*, synchronization is identified and thus, HTM system can be notified and take the correct action, which is to commit T2 immediately, to break the livelock. Next time if the same store comes again, the HTM system can avoid the livelock automatically.

It is worth noting that if FRIENDLYFIRE occurs, our detector can still work correctly. This is because the implementation of *lock()* needs to perform both read and write to the *lock*. Thus, after certain number of abortions, the lock synchronization can be identified and the HTM system can enforce the rules to avoid the livelock.

# 5. EXPERIMENTS

In this section, we first measure in the worst case how often livelock situations can arise in SPLASH-2 benchmarks where busy-waiting synchronizations are widely used. Then we evaluate the accuracy of hardware-based synchronization detection scheme. We also evaluate the performance overhead of dealing with livelocks using SCR strategies. Finally we estimate the impact of different transaction sizes on the performance.

## 5.1 Evaluation Methodology

Our software monitoring infrastructure is built on Intel's Pin Infrastructure [6]. We considered dynamic information flow tracking (DIFT) as our monitoring application [12] and implemented it in *Pin* for our experiments.

Our hardware synchronization detector is implemented on *Simics* [8] with 4 3GHz x86 processors, 64KB 4-way private L1 cache and 2MB 8-way unified L2 cache configuration. We assume the MESI *snoopy* cache coherence protocol. All the caches are assumed to be *write back* and *write allocate* caches.

For our transactional memory experiments, we consider a Hardware TM (HTM) system that uses an EE policy similar to the *logTM* system [10]. We used emulation (in Pin) to estimate the overhead of transactional memory. We used a *nop* instruction to indicate start of a transaction and the $mfence$ instruction to indicate the end of a transaction. A memory fence instruction enforces memory ordering of instructions before and after it, and since this functionality is required for *Transaction-end*, we use the $mfence$ instruction.

## 5.2 Characteristics of Benchmarks

We use SPLASH-2 benchmark [14] suite as our subject programs since it is widely used to facilitate the study of shared memory multiprocessors. and contains many different type of synchronizations. The SPLASH-2 benchmarks gives hooks for us to implement our own synchronization operations and we used *pthread-spin-lock* to implement our locks and used the *sense-reversing* counter based barrier. The SPLASH-2 programs also contain other busy-waiting synchronization operations hard coded into the source code.

| Programs | LOC | Input | Sync. Time (%) | # of Conflicts |
|---|---|---|---|---|
| BARNES | 2.0K | 8192 | 4.1 | 34490 |
| FMM | 3.2K | 256 | 7.1 | 998 |
| OCEAN-1 | 2.6K | $258 \times 258$ | 3.5 | 6893 |
| OCEAN-2 | 4.0K | $258 \times 258$ | 4.3 | 6953 |
| RADIOSITY | 8.2K | batch | 1.5 | 237960 |
| RAYTRACE | 6.1K | tea | 3.7 | 76706 |
| VOLREND | 2.5K | head -adpt | 12.5 | 2325 |
| WATER-NS | 1.2K | 512 | 8 | 4205 |
| WATER-SP | 1.6K | 512 | 3.7 | 4073 |

**Table 2: Characteristics of SPLASH-2 Benchmarks**

Table 2 shows the names, number of lines of source code and inputs, the percentage of time spent in synchronization and number of conflicts for each benchmark used. The time spent on performing synchronizations for each processor varies from 1.5% to 12.5%. The number of conflicts represents the potential number of situations which result in a livelock for the EE strategy. We measured the number of conflicts by instrumenting the original program and detecting the number of situations in which two processors synchronize with each other using shared memory accesses. As we can see, the number of conflicts are high since we used busy-waiting synchronization algorithms.

## 5.3 Accuracy of synchronization detection

Table 3 shows the accuracy of our synchronization detector. The actual number of synchronizations and the the number

| Programs | Actual No. | Detected No. |
|---|---|---|
| BARNES | 33 | 33 |
| FMM | 59 | 55 |
| OCEAN(non-contiguous) | 44 | 44 |
| OCEAN(contiguous) | 42 | 42 |
| RADIOSITY | 60 | 60 |
| RAYTRACE | 19 | 19 |
| VOLREND | 35 | 35 |
| WATER-NS | 24 | 24 |
| WATER-SPATIAL | 28 | 28 |

**Table 3: Accuracy of Synchronization Detection**

of synchronizations we detected by our detector are shown in the second and third column respectively. Note that only in one benchmark *FMM*, we miss 4 patterns, which are all flag synchronizations. This is because the instructions that set the flag always execute earlier than the instructions that check the flag. Since our synchronization is based on spin detection, we failed to detect these synchronizations in the

above situations as there is no spin. Although our detector failed to detect them, there is no transaction conflict, precisely for the same reason (no spin) and thus these synchronizations cannot lead to any livelocks.

## 5.4 Performance Overhead of Livelock Handling

The baseline strategy of eliminating livelocks is to create a transaction for each *basic block* (BB). However, creating and committing too many small transactions is not cheap. Based on our experiments, each SPLASH-2 benchmark execution on an average has 74.8 Million basic blocks. Creating and committing these number of transactions leads to 67% overhead on an average (not shown in the graph). It is worth noting that these are the performance overheads over and above the instrumentation overhead for performing the monitoring.

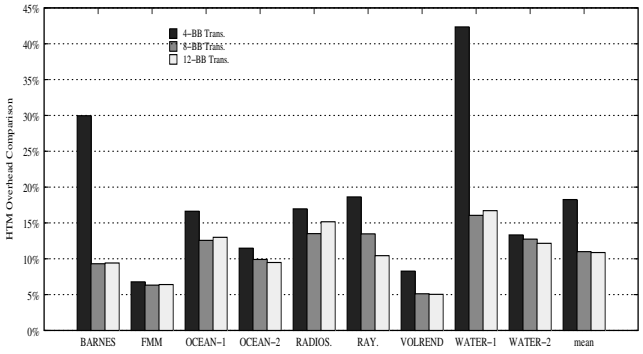


**Figure 6: SCR Strategy Overhead.**

Our SCR strategy makes it possible to put multiple basic blocks within a transaction while at the same time it safely avoids livelocks. We evaluated the performance overhead of our SCR strategy for different sizes of transactions namely, 4-BB-transaction, 8-BB-transaction and 12-BB-transaction, the results are shown in Fig. 6. As we can see, using 4-BB-transactions dramatically decreases the average overhead from 67% to around 18%. Using 8-BB-transactions decreases the overhead further to around 11%.

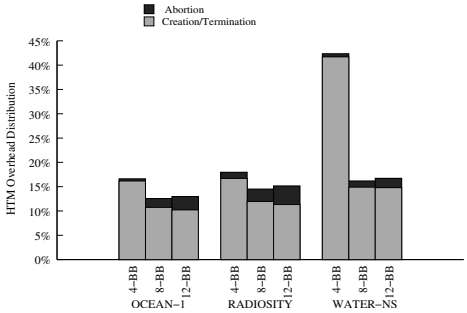


**Figure 7: HTM Overhead Distribution**

However, we observe that the 12-basic-block transaction strategy does not always perform better compared to 8-basic-block transaction. In fact, for somebenchmarks (OCEAN-2 and RADIOSITY) 8-BB-transactions perform better. Fig. 7 shows the distribution of the TM overhead between overhead for creation/termination and overhead for aborting the transactions. As we can see, while the overhead for creation/termination is lower for larger transactions, the overhead for aborting a transaction increases with the size of transactions.

# 6. CONCLUSION

Transactional memory has been recently proposed for performing software based monitoring of multithreaded programs. Unfortunately, enclosing legacy synchronizations into transactions leads to several livelock scenarios. To deal with these livelocks, we propose a synchronization aware conflict resolution strategy (SCR), that is able to efficiently avoid these livelock scenarios.

# 7. REFERENCE


[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06*, pages 26–37, New York, NY, USA, 2006. ACM.

[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*, pages 81–91, New York, NY, USA, 2007.

[3] J. Chung, M. Dalton, H. Kannan, and C. Kozyrakis. Thread-safe binary translation using transactional memory. In *HPCA '08*, 2008.

[4] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS '06*, pages 336–346, New York, NY, USA, 2006. ACM.

[5] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93*, 1993.

[6] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, 2005.

[7] P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. pages 165–171, citeseer.ist.psu.edu/magnusson94queue.html.

[8] P. S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner. Simics/sun4m: a virtual workstation. In *ATEC'98*, pages 10–10, Berkeley, CA, USA, 1998. USENIX Association.

[9] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[10] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in logtm. *SIGOPS Oper. Syst. Rev.*, 40(5):359–370, 2006.

[11] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[12] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO '06*, pages 135–148, 2006.

[13] A. R. L. Tong Li and D. J. Sorin. Spin detection hardware for improved management of multithreaded systems. In *IEEE TRANS. ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 17, No.6*, 2006.

[14] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA '95*, pages 24–36, 1995.