In the Proceedings of the 20th International Conference on
Parallel Architectures and Compilation Techniques (PACT-2011)
Galveston Island, TX, USA, Oct. 2011.

Making STMs Cache Friendly with Compiler Transformations

Sandya Mannarswamy
IISc & Hewlett Packard India Ltd
Bangalore, India
sandya@hp.com

Ramaswamy Govindarajan
SERC, Indian Institute of Science,
Bangalore, India
govind@serc.iisc.ernet.in

Abstract

Software transactional memory (STM) is a promising programming paradigm for shared memory multithreaded programs. In order for STMs to be adopted widely for performance critical software, understanding and improving the cache performance of applications running on STM becomes increasingly crucial, as the performance gap between processor and memory continues to grow. In this paper, we present the most detailed experimental evaluation to date, of the cache behavior of STM applications and quantify the impact of the different STM factors on the cache misses experienced by the applications. We find that *STMs are not cache friendly,* with the data cache stall cycles contributing to more than 50% of the execution cycles in a majority of the benchmarks. *We find that on an average, misses occurring inside the STM account for 62% of total data cache miss latency cycles experienced by the applications and the cache performance is impacted adversely due to certain inherent characteristics of the STM itself.*

The above observations motivate us to propose a set of specific compiler transformations targeted at making the STMs cache friendly. We find that STM's fine grained and application unaware locking is a major contributor to its poor cache behavior. Hence we propose selective Lock Data co-location (LDC) and Redundant Lock Access Removal (RLAR) to address the lock access misses. We find that even transactions that are completely disjoint access parallel, suffer from costly coherence misses caused by the centralized global time stamp updates and hence we propose the Selective Per-Partition Time Stamp (SPTS) transformation to address this. We show that our transformations are effective in improving the cache behavior of STM applications by reducing the data cache miss latency by 20.15% to 37.14% and improving execution time by 18.32% to 33.12% in five of the 8 STAMP applications.

1. Introduction

Software transactional memory (STM) is a promising programming paradigm for shared memory multithreaded programs as an alternative to traditional lock-based synchronization [1, 2, 23]. Of late, there has been considerable work in supporting STMs for unmanaged environments [15, 23] like C/C++, since much of the performance-critical enterprise applications are still written in them. Thus, a high performance STM for unmanaged environments is crucial for deploying transactional memory as a programming mechanism. State of the art STM implementations for unmanaged environments have been developed and have been shown to support atomic sections efficiently over word-based transactional memory without requiring changes to the heap object layouts [3, 4, 15, 23].

Despite these developments, adoption of STM in mainstream software has been quite low. The overheads incurred by STMs are often quoted as the main reasons for the above view [7] and considerable research has focused on addressing this issue [1, 6, and 10]. Second, the memory/cache performance of STMs is another source of performance problem. In order for STM to be adopted widely for performance critical software, understanding and improving the memory performance of applications running on STM becomes increasingly crucial, as the performance gap between processor and memory continues to grow.

While there has been considerable work on cache performance characterization and compiler transformations to reduce data cache misses in the context of both single threaded and traditional lock-based multithreaded applications, this area has not received significant attention in the context of applications running on STMs. Table 1 shows the percentage of data cache (d-cache) miss stall cycles experienced on a state-of-the-art STM implementation [3] for a set of STAMP applications [8] and the corresponding numbers for a lock based[1] version of the same benchmarks run with 32 threads on a 32 core IA-64 server. In both cases, the applications were compiled using the same optimization level. We see that the STM runs experience significantly higher d-cache miss stall cycles than the lock-based implementation. Further, in the STM-32 implementation, more than 50% of total execution cycles in all the six benchmarks are spent on d-cache misses. Whereas in the Lock-32 version, the percentage stalls cycles is less than 35% in all but one benchmark.

What is the reason for the adverse cache performance of STM? While STMs support increased parallelism by their basic performance premise of optimistic concurrency control, supporting optimistic concurrency can have significant impact on the cache behavior of applications running on STM. Fine grained locking employed by the STM while improving concurrency, also impacts the cache behavior adversely due to the increased number of lock accesses required. An atomic section which accesses N

[1] Since the STAMP suite does not provide lock-based equivalents of the transactional benchmarks, we created equivalent lock based versions using the level of locking granularity that an average programmer would use in writing the application. We do note that the choice of locking granularity is an important factor on the cache miss behaviour. Since our intention is not to obtain/compare absolute numbers, but to get an estimate of the ratio of the total miss latency cycles taken on a STM implementation to a lock based version, we considered it as a reasonable approach.

words of shared data will contain an additional programmer invisible N lock word accesses on a word based STM, increasing the pressure on the cache. To support optimistic concurrency, STMs also need to perform considerable book-keeping activity such as maintenance of read/write sets, maintenance of the global version clock to enable validation of transactional data and redo/undo logs etc. Such activities being memory intensive can also impact the cache behavior adversely.

Understanding the impact of STM on cache behavior of applications running on it aids in developing targeted compiler transformations that can address the cache bottlenecks of STMs. Therefore in the first part of this paper, we perform a study of the cache behavior of STM applications in unmanaged environments using state of the art STMs [3, 4, 20] and quantify the impact of different STM factors on the cache misses experienced.

BM	STM-32 Implementation		Lock-32 Implementation	
	D-Cache Stall Cycles (In Billions of Cycles)	Stall cycles as % of Exec. Cycles	D-Cache Stall Cycles (In Billions of Cycles)	Stall cycles as % of Exec. Cycles
Kmeans	504.73	54.25%	2.87	34.25%
Genome	53.43	51.58%	10.22	18.45%
Intruder	2237.39	56.98%	10.29	15.71%
Ssca2	429.65	84.12%	155.06	43.04%
Vacation	758.65	56.85%	39.08	17.73%
Yada	26.38	53.58%	0.375	17.05%

Table 1 : D-Cache Performance in STM and Lock versions

We find that STM lock access misses (misses due to the locks/metadata associated by the STM with shared data) account for more than 50% of the total STM misses in majority of the applications. Our analysis also reveals that STM accesses to the Global Time Stamp (used for versioning) in time stamp based STMs [3, 4, 20] can incur significant overhead in certain cases, leading to a performance pathology known as Global Commit Convoy Bottleneck which we describe in Section 2.4.2. Further, we find that heap allocations inside transactions with high conflicts impact cache performance due to repeated allocations and releases as the transactions abort and restart. We also demonstrate that the cache behavior analysis is not an artifact of a particular STM implementation by presenting the cache behavior of STAMP applications on two other state-of-the-art STMs, namely TinySTM [4] and SwissTM [20]. To the best of our knowledge, this is the first detailed work on analyzing the effects of the STM layer on cache performance.

Our cache analysis study is helpful in identifying the sources of the performance bottlenecks, which in turn motivates us to propose a set of specific compiler transformations targeted at reducing the cache overheads on STM. We propose (i) selective Lock Data Co-location (LDC) (ii) Removal of Redundant Lock Accesses (RLAR) to address the lock access misses and (iii) Selective Per-partition Time Stamp transformation (SPTS) to address the global time stamp misses. We show that they are effective in improving the cache behavior of STM applications by reducing the data cache miss latency by 20.15% to 37.14% in 5 of the 8 benchmarks.

The rest of the paper is organized as follows: Section 2 provides a detailed analysis of the cache miss behaviour of applications running on STM. Section 3 describes the compiler transformations. We discuss related work in Section 4 and provide concluding remarks in Section 5.

2. Data Cache Behavior of STM Applications

2.1 Experimental Methodology

We used STAMP benchmarks [8] with native input sets (except for Bayes, for which we used the smaller input set due to an issue with our memory allocator library) for our study with TL2 STM [3] as our underlying STM. TL2 being a lock based STM, with support for invisible reads and a global version number based validation represents many of the common design choices made by the state of the art STMs. We used a 32 core IA-64 BL-890c, with 8 Intel IA-64 Processors 9350 with 4 cores per socket, and a total of 127.71GB of physical memory. Each core operates at 1.73 GHz and has 6 MB non-shared L3 cache per core. The latencies of L1, L2, and L3 caches are 1, 5 and 20 cycles respectively. Cell local cache to cache (c2c) latency, cell local memory latency, one-hop remote memory latency, 2 hop remote memory and 1 or 2 hop remote c2c access times are 150, 300, 650, 750, 750+ cycles respectively. All our runs were with 32 threads unless otherwise mentioned. D-cache miss data was obtained using the performance profile tool caliper [9] by sampling the IA-64 hardware Performance Monitoring Unit counters and Event Address Registers (with a sampling period of 10000 events). We use the following terms in our discussion as given in Table 2.

CPU_ CYCLES (or) *Total Cycles*	Sum of elapsed CPU cycles over all the cores
Number of D-cache Misses (NDM)	Number of sampled data cache misses, attributed to the given program object
D-cache Miss Latency Cycles (DML)	Number of cycles expended on data cache misses summed across samples from all threads/cores for the given program objects
Total D-cache Miss Latency Cycles (TDML)	Sampled total data cache miss latency cycles, summed over the entire application
Average D-cache Miss Latency (ADML)	TDML/NDM

Table 2 – Terminology

However due to restriction in the number and combination of hardware counters that can be monitored simultaneously, we do not report the L1, L2 and L3 cache misses individually.

2.2 D-cache Misses in STM

As a first step towards getting a rough idea of how significant are d-cache misses in STM applications, we compare the Total Dcache Miss Latency Cycles incurred by each STAMP application run with 1 or 32 threads. We compare these numbers also with a lock-based version of the same benchmark to show that the STM versions incur much higher Total Dcache Miss Latency Cycles (TDML) compared to the lock based version of the benchmarks.

Benchmark	Total Data Cache Miss Latency (TDML)				Normalized TDML of STM-32	
	Lock-1	Lock-32	STM-1	STM-32	STM32/STM-1	STM-32/Lock-32
Kmeans	100,841	120,655	526,385	2,947,147	5.59	24.43
Genome	160,447	578,362	309,407	838,263	2.71	1.45
Vacation	980,264	1,074,142	7,350,212	13,044,748	1.77	12.14
Intruder	78,333	421,956	1,811,037	18,442,525	10.18	43.70
Ssca2	1,226,711	1,784,489	1,583,770	4,273,552	2.69	2.39
Yada	4,812	9,888	93,597	227,133	2.42	22.97
Labyrinth	2,107,845	2,114,307	2,510,809	2,578,440	1.03	1.22
Bayes	22,986	24,832	25,727	26,913	1.04	1.08

Table 3 Comparison of D-Cache Miss Latency in STM and Lock-Based Implementations

Benchmark	TDML	App_Ms		Other_Ms		STM_Ms		DynAlloc_Ms	
		DML	% of TDML	DML	% of TDML	DML	% of TDML	DML	% of TDML
Kmeans	2,947,147	514,479	17.4%	978	0.03%	2,431,690	82.50%	0	0
Genome	838,263	417,538	49.81%	6,820	0.01%	391,015	46.61%	30,239	3.61%
Vacation	13,044,748	550,416	4.21%	3,572	0.03%	12,106,359	92.82%	381,419	2.90%
Intruder	18,442,525	2,449,987	13.33%	12,032	0.06%	13,052,328	70.71%	2,926,836	15.81%
Ssca2	4,273,552	2,069,608	48.41%	1,345	0.03%	2,185,887	51.12%	53	0.001%
Yada	227,133	75,372	33.24%	780	0.3%	80,463	35.43%	70,436	31.11%
Labyrinth	2,578,440	1,292,546	50.12%	1,277,127	49.51%	8,626	0.31%	0	0%
Bayes	26,913	25,422	94.53%	1,254	4.64%	237	0.90%	0	0%

Table 4 Breakup of D-Cache Miss Latency in STM

The comparison of the sampled total data cache miss latency (TDML) cycles of STM-32 and STM-1 versions gives a rough measure of the impact of conflicts on the d-cache miss behavior. (Note that TDML reported here is from the sampled Data EAR events whereas the total d-cache miss stall cycles reported in Table-1 are from the CPU Stall counters. Hence these two values will be different). We use TDML for our analysis since it allows us to correlate the cache miss cycles to a particular code/data object which is not possible with stall cycles measurement of the tool.

We note that the d-cache miss behavior of STM with 32 threads will also include various other effects such as the NUMA effects, the resource requirements due to the increased number of threads and other secondary effects. We see from Table-3 that Intruder (10.18X) and Kmeans (5.59X) show the highest ratio difference for TDML for the STM-32 and STM-1. This corresponds well with the fact that Intruder has the highest conflicts to starts ratio over all STAMP benchmarks followed by Kmeans. Hence these two benchmarks suffer worse in the d-cache miss behavior due to conflicting transactions. Most other benchmarks show a 2X degradation in TDML due to conflicting transactions. Conflicts contribute to a major portion of TDML in STMs and need to be addressed to improve the memory performance. There has been extensive research on addressing STM conflicts [23] and hence we do not focus on it further in this paper.

2.3 Breakup of D-cache Misses in STM

Next, our focus is on analyzing the d-cache misses occurring in STM library itself. Therefore we classify the d-cache misses into the following four classes first: (a) D-cache misses occurring inside STM library which we term as *STM_Ms*, (b) Misses which occur in the application code (not in the STM) which we term as *App_Ms*, (c) Misses which occur in dynamic memory allocation routines such as malloc/free routines from the memory allocator library which we term as *DynAlloc_Ms*, (d) Misses which are not included in the categories above, termed as *Other_Ms* (a catch-all bucket). This includes misses in standard libc (memcmp, strcmp), stripped libraries etc.

While d-cache misses in buckets such as *App_Ms* and *Other_Ms* need to be analyzed and addressed from an application perspective to improve the performance, we focus our attention on *STM_Ms* component since our interest is on the impact of STM layer on d-cache misses. We also look at the *DynAlloc_Ms* component since STM can impact the dynamic memory allocation costs due to aborting transactions as we discuss later in Section 2.5. Table-4 shows the contribution to TDML by *STM_Ms* and *DynAlloc_Ms* across all benchmarks. We find that *STM_Ms* component for the 2 benchmarks Labyrinth and Bayes are less than 1% as their TDML is dominated by *App_Ms* and *Other_Ms* components. Application and library routines dominate the cache misses in Labyrinth (*'memmove', 'PexpandToNeighbor'*) and Bayes (*'vector.c'*) and hence we do not discuss them further. From column 8 of Table-4 we see that the contribution of *STM_Ms* to TDML is significant, ranging from 35.42% (Yada) to 92.8% (Vacation). Hence we analyze the individual components of *STM_Ms* next.

2.4 Breakup of *STM_Ms*

In this section we examine the STM_Ms data in detail, giving a breakup of STM_Ms into the following 4 sub-components:

i. *Lock_Ms* are the misses occurring due to the STM lock (assigned by STM to protect the shared data) accesses.
ii. *Data_Ms* are the misses occurring due to the actual shared data accesses occurring in the STM Read and STM Write routines. We do not focus on this component further.
iii. *GTS_Ms* are the misses occurring due to the global time stamp (GTS) accesses. The Global timestamp mechanism (for maintaining consistency) uses centralized meta data that is modified by every committing transaction (except RO transactions) at least once, leading to costly coherence misses for the global timestamp counter [3, 4, 10, 20].
iv. *BK_Ms*: STMs typically have to maintain bookkeeping data, such as read/write sets, about every access, which can contribute to STM_Ms and we term them as *BK_Ms*. This is also a catch-all bucket for all other activities happening in the STM apart from *Lock_Ms*, *Data_Ms* and *GTS_Ms*.

BM	Lock Misses (Lock_Ms)			
	No. of Misses	Miss Latency	% Contrib. To STM_Ms Latency	Average D-cache Miss Latency
Kmeans	2,648	1,249,159	51.37%	471.36
Genome	6,641	197,997	50.63%	29.81
Intruder	137,846	7,192,897	55.12%	52.18
Ssca2	1,507	744,377	34.05%	493.94
Vacation	585,292	6,547,082	54.07%	11.18
Yada	508	35,419	44.02%	69.72

Table-5 STM Lock Miss (Lock_Ms)

2.4.1 STM Lock Access Misses (*Lock_Ms*)

Table-5 shows the breakup of *Lock_Ms* component for each of the STAMP benchmarks. We see from column 4 of Table 5 that a significant fraction of *STM_Ms* latency cycles is due to the *Lock_Ms* component. Since our experiments were run on TL2 with STAMP, the transactional loads/stores are marked explicitly. In case of compiler based STMs, compiler over-instrumentation can result in a much larger superset getting marked as transactional. Hence the impact of *Lock_Ms* component is likely to be higher in such cases. Since TL2 is word based, the contribution due to *Lock_Ms* observed here can be higher than what would be observed in an STM with cache line level granularity. We experimented using different sizes of 16, 32 and 64 bytes locking granularity and found that *Lock_Ms* component remains a significant contributor to overall *STM_Ms*.

BM	GTS_Ms			
	No. of Misses	Miss Latency	% Contrib. To STM_Ms Latency	Average D-cache Miss Latency
Kmeans	582	392,794	16.14%	674.90
Genome	113	91,450	23.38%	809.29
Intruder	2,059	1,270,780	9.70%	617.18
Ssca2	583	832,767	38.09%	1,428.41
Vacation	394	146,685	1.21%	372.29
Yada	4	2,162	2.62%	540.5

Table-6 Global Timestamp Miss (GTS Miss)

2.4.2 Global Time Stamp Misses (*GTS_Ms*)

Many STM implementations use a global time stamp (GTS) for validating shared data [3, 4, 20]. In the original TL2 implementation, all transactions read the GTS at the beginning of the transaction and GTS is updated once by transactions that write to shared data, leading to costly cache coherent misses. In its default mode, called GV4, TL2 mitigates this problem by using a pass-on time stamp approach [3]. When concurrently executing transactions with little or no data conflicts commit, GTS experiences ping pong across the caches, leading to costly coherence misses. This is because each committing transaction tries to update GTS at least once. We term this as GTS Commit Convoy Bottleneck. With GV4, at least one attempt will be made by each committing transaction to update the GTS.

GTS_Ms contribution is high in the two benchmarks, Ssca2 and Genome as shown in Table 6. In Ssca2 and genome, most of the atomic sections are relatively conflict-free and hence exhibit the GTS commit convoy bottleneck. *The very fact that disjoint access parallel transactions suffer worse due to this bottleneck makes it a serious concern from the point of view of STM performance.* Further, *Lock_Ms* and *GTS_Ms* together contribute to more than 65% of STM_Ms stall cycles in most of the benchmarks. This suggests that (compiler or other) transformations that target to reduce these components would help improve the STM performance.

BM	Book-Keeping Misses (BK_Ms)			
	No. of Misses	Miss Latency	% Contrib. To STM_Ms Latency	Average D-cache Miss Latency
Kmeans	4,387	138,491	5.69%	31.56
Genome	1,365	37,305	9.54%	27.33
Intruder	277,417	2,181,469	16.71%	7.863
Ssca2	1,784	207,293	9.48%	116.2
Vacation	191,870	1,107,724	9.15%	5.773
Yada	3,894	18,661	23.19%	4.79

Table-7 Bookkeeping Miss (BK_Ms)

2.4.3 STM Book-Keeping Misses (BK_Ms)

Next we consider the contribution of BK_Ms component to the *STM_Ms*. Table 7 gives the contribution of this component. This component is high in Intruder since Intruder exhibits significant amount of conflicts and restarts, leading to repeated book keeping activities. In Yada, it is high due to the large transactional read/write sets. Unlike the study on Haskell STM [22], we did not find the STM book keeping functions for maintaining read/write sets to be a main contributor to d-cache misses.

2.5 DynAlloc_Ms Component

STM can lead to an increase in the dynamic memory allocation costs due to aborting transactions having to release/re-allocate memory dynamically allocated inside the transactions. Dynamic memory allocation in multithreaded applications is expensive due to the need to protect shared meta-data structures of the dynamic memory allocation routines. Increased number of dynamic memory allocations due to repeating transactions (due to aborts) can further exacerbate this overhead. We see from column 10 of Table 4 that the contribution of *DynAlloc_Ms* to the STM stall cycles is significant in two of the benchmarks namely intruder (15.8%) and Yada (31%). Both suffer from high conflicts and hence the dynamic allocation overhead inside transactions is increased due to aborting transactions.

2.6 Average D-cache Miss Latency

Column 5 of Tables 5, 6 and 7 shows the average d-cache miss latency for each component respectively. We see that *GTS_Ms* has the highest ADML since they are typically c2c coherence misses. ADML (*BK_Ms*) is less than the IA-64 L3 cache latency in most benchmarks and in others, it is within the local c2c latency. ADML (*Lock_Ms*) is high for Kmeans and Ssca2, with the latency values indicating that lock words ping-pong across the caches, indicating possible hash conflicts on the STM's lock table. We used the IA-64 performance monitoring unit's trace measurement of data access pattern to determine that these are due to false conflicts in case of Kmeans. Hence transformations which can help address false conflicts should improve the ADML for Kmeans. High ADML (*Lock_Ms*) in Ssca2 happens because of true conflicts, due to the application's data access patterns. ADML (*Lock_Ms*) in Intruder, Yada and Genome are higher than the L3 cache latency reflective of the lack of spatial locality among the shared data accesses.

2.7 Applicability of our Analysis to other STMs

In order to verify that our cache behavior analysis of STM applications is applicable in general to other STMS as well, we studied the d-cache miss behavior with two other state-of-the-art STMs namely Swiss TM [20] and TinySTM [4].

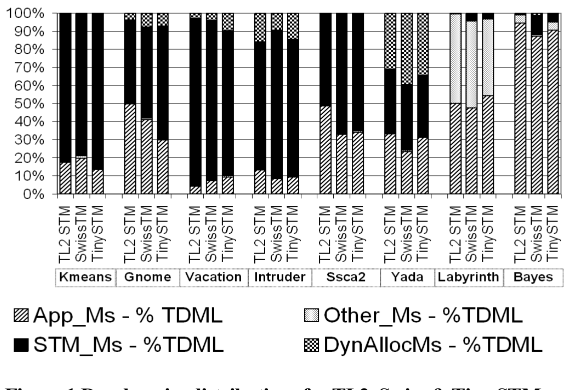

Figure 1 Dcache miss distributions for TL2, Swiss & Tiny STM

Figure 1 shows the breakup of the (sampled) total d-cache miss latencies in terms of DML of different misses in Swiss TM and TinySTM. We see that the contributions of different classes of misses are more or less similar across all three STMs. Further, STM_Ms component continues to dominate (more than 50%) in 5 of the 8 STAMP applications. In both Swiss TM and TinySTM as in TL2, lock misses are a major contributor to STM misses, contributing nearly 40% or more misses in all the six benchmarks and book keeping misses are not significant.

2.8 Applicability to different configurations

For our experiments, we collected the d-cache miss data for all configurations 4, 8, 16 and 32 threads. Due to space constraints, we only present the *STM_Ms* sub components of interest for 4T and 32T configurations in Table-8. We find that the cache miss distribution remains similar across varying number of threads. We found that *Lock_Ms* remains the main contributor to STM_ misses. However the contribution due to *GTS_Ms* is more with larger number of threads (due to increased cache line ping-pong effect of the Global Time Stamp).

BM	Lock_Ms % contribution to STM_Ms		GTS_Ms % contribution to STM_Ms		BK_Ms % contribution to STM_Ms	
	4T	32T	4T	32T	4T	32T
Kmeans	58.91%	51.37%	4.61%	16.14%	7.42%	5.69%
Genome	57.84%	50.63%	10.26%	23.38%	8.41%	9.54%
Intruder	61.83%	55.12%	3.08%	9.70%	18.04%	16.71%
Ssca2	48.81%	34.05%	11.97%	38.09%	7.84%	9.48%
Vacation	60.41%	54.07%	0.21%	1.21%	7.32%	9.15%
Yada	49.16%	44.02%	0.61%	2.62%	19.64%	23.19%

Table-8 STM_Ms distribution for 4T and 32T

3. Compiler Transformations to Address D-cache Performance Bottlenecks in STM

To address the cache bottlenecks experienced due to STM layer, we propose 3 transformations namely (i) selective Lock Data Colocation (LDC), (ii) Redundant Lock Access Removal (RLAR) and (iii) Selective Per-partition Time Stamp Transformation (SPTS). LDC and RLAR are intended to address the lock access misses and SPTS addresses the *GTS_Ms* component. We also propose Dynamic Allocation Optimization (DAO) to reduce the dynamic memory allocation overheads in transactions. Due

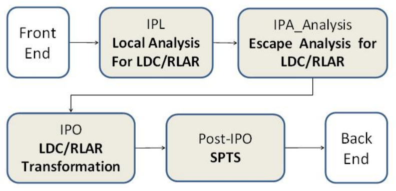

Figure 2 Open64 compiler phases

to space constraints, we only briefly touch upon the legality and profitability aspects of each of our transformations.

We implemented our transformations using the Open64 compiler [13] and with TL2 [3] as our underlying STM. Fig 2 shows the Open64 compiler phases where our transformations were implemented. We used the same machine configuration as described in Section 2.1. The baseline for our experiments is to compile the benchmarks at the same optimization level (O3) with inter-procedural analysis enabled, but without our transformations and run them on the unmodified TL2 implementation. We refer to this baseline as TL2. Our optimized version is TL2-Opt. In all cases, we report the execution of the application with 32 threads. For each transformation, we first briefly motivate its applicability, describe our approach and then present our experimental evaluation of the same.

3.1 Data and Lock Layout Issues

One major source of cache performance bottlenecks is accesses to locks assigned by the STM to protect the shared data accessed inside a transaction. Consider the code snippet in Figure 3 which is simplified from the STAMP benchmark 'Ssca2'.

```
for (i = i_start; i < i_stop; i++) {
    for (j = GPtr->outVertexIndex[i];.. ; j++) {
      int v = GPtr->outVertexList[j];
      TM_BEGIN();

       long inDegree = GPtr->inDegree[v];
       GPtr->inDegree[v] = (inDegree + 1);

       TM_END();
  }
}//end for i
```

Figure 3: Example Showing Data and Lock Layout Issues

In the above example, the array '*GPtr->inDegree[]*' is accessed using '*v*' which is the out vertex of vertex '*i*'. Each iteration of loop '*j*' has the programmer visible access to '*GPtr->inDegree[v]*'. Each transactional data access, also has associated with it, an access to its corresponding STM assigned lock which is invisible to the programmer but adds to the total data accessed within the loop. Thus each transactional execution of the atomic section AS inside loop '*j*' brings in two cache lines of data, one for the access of the data '*GPtr->inDegree[v]*' and other for the lock associated with it by the STM. Since there is no spatial locality between the values of '*v*' in the loop '*j*', the data access '*inDegree[v]*' in this transactional region already suffers from poor cache performance. Further, since STMs typically use a hash function to map the address of the shared data item to a lock, a shared data access pattern which does not exhibit spatial locality as in above, will also result in poor spatial locality among the STM lock accesses corresponding to those shared data accesses. While the poor cache behavior of '*inDegree[v]*' is an application artifact, this also leads to a similar bad behavior in the STM lock accesses corresponding to these shared data accesses, worsening the data utilization in this region. Therefore we propose a compiler driven lock-data co-location (LDC) transformation for addressing this bottleneck.

3.1.1 Compiler Driven Lock Data Co-location

There has been considerable work on co-locating contemporaneously accessed data in the context of single threaded applications for improving locality [19]. In multithreaded code, lock-data co-location has been used by programmers manually to improve the performance of hot shared variables in lock based programs. Since typical lock based programs use only a few locks, it is feasible for the programmer to use his knowledge of the application to place lock and data together to improve performance. With extremely fine grained locking as employed by an STM, such a manual approach is not feasible. Nor is it possible for the programmer to evaluate manually the trade-offs in employing lock-data co-location for data structures which can be accessed both transactionally and non-transactionally. The reason is that co-locating the lock with data can have an adverse impact on spatial locality in non-transactional regions where the shared data is accessed contiguously. Hence a whole program compiler analysis is needed to drive the transformation.

LDC consists of two major steps namely candidate identification and code transformation. LDC results in modification of shared data layout. Hence we perform extensive legality checks to filter ineligible candidates as described in Section 3.1.2. Each candidate, which passes our set of legality checks is then analyzed to see if applying LDC is profitable for it, as we discuss below in Section 3.1.3. If so, it is then marked as eligible for LDC. For each LDC candidate, the code transformation step involves modifying the data layout of the LDC candidate and modifying the shared data references throughout the whole program accordingly.

3.1.2 Legality Checks

For each shared data item accessed inside atomic sections, the compiler performs certain legality checks to verify that it is eligible for lock-data co-location. LDC changes the data layout of the program. Changing the data layout can violate certain assumptions made by the programmer. This is because a programmer often uses knowledge of data layout in a structure definition for writing an application. For instance, if a programmer uses address arithmetic from the address of a shared datum to access another data, changing the data layout of the shared datum by co-locating the lock with it, can violate the programmer's assumptions. Hence extensive legality checks are performed to ensure safety of LDC. Our set of legality checks consists of the following:

a. **Dangerous type casts:** This test looks for any programmer specified explicit type casts (other than the standard casts applied for malloc/calloc) applied to the data address of LDC candidates, which can indicate type unsafe usage of the shared data.
b. **Escape to an external opaque function:** If LDC candidates address escape to an external function not visible during the compiler's whole program analysis and if the function is not a standard library function whose semantics is known, then the candidate is marked as ineligible for LDC.
c. **Address Arithmetic**: If the compiler detects address arithmetic operations on the data address for LDC candidates, it marks them as ineligible.

Other than the ones mentioned above, our legality checks also include a few other corner case checks which are similar to those applied for any data layout transformation eligibility [19, 24]. Our legality check implementation is split into 2 phases, with type cast and pointer arithmetic checks being done in the optimizer front end summary phase while the address escape checks are being done in the inter-procedural whole program analysis of the compiler.

3.1.3 Profitability Analysis for Lock-Data Co-location

Lock data co-location is intended to reduce the volume of data fetched inside the atomic blocks by bringing in the corresponding lock when the shared data is accessed. However co-locating the lock with data can reduce the spatial locality for contiguous accesses of that shared data, requiring a larger number of cache lines to be fetched for the data accessed over non-transactional code regions. Therefore we estimate the profitability of performing LDC for each shared data candidate as explained below.

Accessed Data Volume (ADV) is the total volume of data actually accessed/used by the application over a local code region whereas Fetched Data Volume (FDV) is the total volume of data fetched to satisfy the data accesses over that code region. For instance if we access only every 8th byte of a character array, the fetched data is 64 bytes (1 cache line) for each array element access but the accessed data is only 8 bytes. The ratio of ADV to FDV gives a measure of the utilization of fetched cache lines.

A local code region can be a loop, function or basic block and can either be transactional or non-transactional. As mentioned earlier, for a shared data item A, accesses to A in a transactional region would include the implicit lock accesses. Further, ADV (r, A) of a region *r* (transactional or non-transactional) would remain the same with or without the LDC. Whereas, with LDC, the FDV (r, A) would decrease for a transactional region and may increase for a non-transactional region. Hence the benefits of LDC is calculated as the difference in FDV without LDC and with LDC, over all regions (transactional and non-transactional), taking into account the frequency of execution of individual regions. Thus

$$LDC\,\mathrm{GAIN(A)} = \sum_{r} (\mathrm{FDV_{NOLDC}(r, A) * f(r)} - \mathrm{FDV_{LDC}\ (r, A) * f(r)})$$

BM	Execution Time (in Seconds)		% improvement in Execution Time	% Improvement in Average D-cache Miss Latency Cycles
	TL2	TL2-opt		
Kmeans	20.21	18.53	8.27%	12.13%
Genome	15.51	14.26	8.06%	10.27%
Intruder	87.27	85.64	1.86%	3.53%
Ssca2	73.81	68.75	6.86%	9.21%
Vacation	64.04	62.98	1.65%	3.34%

Table-9 Performance Improvements due to LDC

where f(r) is the frequency of execution of region r, and the summation is taken over all transactional and non-transactional regions. We perform LDC only if the gain is positive.

Note that computation of FDV(r,A) over each code region requires knowledge of the execution frequencies of the different code regions. For estimating the frequency of execution of code regions, either static or dynamic profile data can be used. We use the static profile heuristics [25] to estimate the execution frequencies in our analysis. Further we note that LDC can impact the cache performance of accesses of other data items as well. However, we do not consider them in this simplistic model.

3.1.4 Code Transformation for Lock-Data Co-location

LDC is performed in the inter-procedural optimization phase of the compiler. We perform LDC both for statically allocated shared data and dynamically allocated shared data. For each LDC candidate, the compiler creates a new type which contains the original datum and the lock word protecting the datum. Compiler instantiates the co-located item with the new type in the case of statically allocated data. In case of dynamically allocated data, allocation and de-allocation points are transformed accordingly to create the new instance of shared data with the lock co-located. All the references to the original shared datum in the whole application are transformed to point to the new instance. Transactional loads and store library calls into STM library for the LDC candidate accesses are transformed to call modified TL2 STM library interfaces, which are passed the corresponding lock for the shared data item as an argument by the compiler and they use this lock instead of using a hash function to compute the lock. Other than this change, the modified interfaces are identical in functionality to default TL2 library transactional load/store interfaces.

Note that the compiler analysis and code transformation for LDC can impact the compile time. The compile time overheads due to LDC ranged from 1.2% to 4.7%. The application's memory requirements increase due to the additional memory allocated for co-locating the locks with data. Though the increase in overall memory consumption due to LDC was not high (less than 2%) for the benchmarks we studied, this factor needs to be taken into account. It is possible to add a heuristic to LDC to estimate the increased memory requirements and limit LDC accordingly.

3.1.5 Performance Results of Lock-Data Co-location

In Table 9, we report performance results for 5 benchmarks, in which LDC identified candidates for transformation. LDC had no impact on the other STAMP benchmarks. In Table 9 we show the execution times for TL2 and TL2-opt in columns 2 and 3 and the performance improvement in terms of % improvement in execution time and the percentage reduction in average d-cache miss latency cycles in columns 4 and 5 respectively.

In the benchmark Kmeans, LDC identified 1 candidate accessed in the main transactional region of the benchmark in '*normal.c*'. By co-locating the lock and the data, the cache line utilization is doubled for this datum. In the benchmark Ssca2, LDC identified 2 candidates for LDC transformation from the atomic section in file '*computeGraph.c*' namely '*inDegree*' array and '*impliedEdgeList*' array, leading to a performance improvement of 6.86%. In the benchmark Genome, LDC is applied for the '*constructEntries'* and the table bucket arrays (file:*sequencer.c*). In Intruder, LDC is applied to '*DecodedQueuePtr'* and to '*reservation'* structure in Vacation.

3.2 Redundant Lock Access Removal Transformation

3.2.1 Issues with Fine-Grain Locks

STM implementations extract increased parallelism by providing fine-grained locks. However such fine-grained locking is not always required for all shared data items. For example, consider a data structure D in which a sub-set of fields S is always accessed together in the application. Hence an STM which supports the fine grained word level granularity locking, would incur a large number of STM lock accesses since each field of D is associated with a separate lock. Instead it is possible to assign the same lock to the sub-set S, thereby reducing the number of lock accesses and hence relieving the pressure on the memory system, without any loss of concurrency. We term this as Redundant Lock Access Removal (RLAR).

3.2.2 Redundant Lock Access Removal

If a group of shared data items $s_1, s_2 \ldots s_k$ are always accessed together in all atomic sections of the application, then the compiler can associate a single lock for them. The compiler performs an 'Always Accessed Together' (AAT) analysis to determine such AAT groups. AAT analysis requires whole program analysis by the compiler since it needs to examine all the atomic sections in the application. Consider the set of atomic sections in the application AS_1 to AS_n. For each atomic section,
compiler constructs the set of shared data items accessed in that atomic section SD (AS_i).

AAT analysis is similar to identifying cliques. Initially we consider each pair of shared items (d_i, d_j). For each atomic section AS_k 1<=k<=N, we set the AAT(d_i, d_j, AS_k) to be true if either of the following conditions are satisfied.

(i) Both d_i and d_j are accessed together in AS $_k$

(ii) Neither di nor dj are accessed in AS $_k$

$AAT(d_i, d_j) = AAT(d_i, d_j, AS_1)$ AND $AAT(d_i, d_j, AS_2)$ AND $\ldots AAT(d_i, d_j, AS_n)$

We first compute AAT(d_i, d_j) for all shared datum pairs. Then we construct a graph in which each node represents a datum. An edge exists between two data items d_i, and dj if and only if AAT(d_i,d_j) is true. Now we can find the AAT sets by finding the maximal cliques in the graph. Each maximal clique is a AAT candidate group. Compiler assigns a single lock to protect each AAT group. Transactional accesses to AAT data items are modified by the compiler to use this lock. Note that RLAR requires the same set of legality checks as LDC as the same restrictions apply.

3.2.3 Performance Results of RLAR Transformation

We show the performance results of RLAR transformation in Table 10. We find that AAT analysis identifies candidates in Kmeans, Genome, Ssca2, Vacation, Intruder, Labyrinth and Yada. RLAR yields 2.64% to 22.41% performance improvement in the first 5 benchmarks. However RLAR yielded negligible performance benefits for Labyrinth and Yada since the RLAR candidates occur in cold atomic sections in these benchmarks. Compile time overheads due to RLAR were from 0.8% to 3.4%.

BM	Execution Time (in Seconds)		% improvement in Execution Time	% Improvement in Average D-cache Miss Latency cycles
	TL2	TL2-opt		
Kmeans	20.21	15.68	22.41%	24.81%
Genome	15.51	15.10	2.64%	4.03%
Intruder	87.27	76.08	12.82%	14.04%
Ssca2	73.81	68.21	7.57%	8.91%
Vacation	64.04	53.09	17.10%	19.67%
Yada	21.56	21.53	0.14%	0.23%
Labyrinth	31.76	31.67	0.27%	0.31%

Table 10 RLAR Performance Improvements

3.3 Selective Per Partition Time Stamp Transformation

Though Global Time Stamp schemes scale well in smaller systems [10], Table 6 shows that '*GTS_Ms*' component is considerable in large NUMA systems. Even transactions which are Disjoint Access Parallel (DAP) still suffer from contention on the global timestamp counter. If each data group accessed by the non-conflicting transactions are associated with different partitions with a per partition time stamp, so that a transaction operating on a specific data group/partition updates only the time stamp associated with that partition, then the contention on the single global time stamp will be reduced, reducing the *GTS_Ms* component. Since our intention is to reduce the GTS misses, we focus our attention only on DAP atomic sections (i.e., those which have very few conflicts) and apply data partitions on them. We term this transformation as Selective Per-partition Time Stamp (SPTS) transformation.

3.3.1 Partitioning of Shared Data

For array based data structures, standard array section analysis can be employed to partition the arrays into different sub-partitions, with each partition having a unique GTS. For pointer based recursive data structures, there has been considerable work in identifying distinct logical heap regions of the application. Lattner and Adve proposed Automatic Pool Allocation (APA) to distinguish disjoint instances of logical data structures in a program [11]. This gives the compiler the information needed to segregate individual logical data structure instances in the heap into distinct pools. For example, each distinct instance of a list, tree or graph identified by the compiler would be allocated to a separate pool. Automatic pool allocation can be employed to partition disjoint instances of data structures into different pools such that each pool is associated with a different GTS. We implemented a prototype of the Data Structure Analysis framework [11] to support automatic pool allocation.

3.3.2 Micro-pool Partitioning

There exist many situations wherein concurrent mostly non-conflicting transactions update disjoint parts of the same data structure. In such a scenario, associating a single GTS with the pool will not suffice, since all concurrent transactions updating that data structure would update the same GTS, causing it to ping pong across processor caches. Hence we propose a variant of the automatic pool allocation, known as micro pool allocation, wherein parts of a single data structure which are updated in a disjoint manner from concurrent threads are allocated to different micro pools with a different time stamp per micro-pool. Thus, Automatic Micro Pool Allocation, in addition to associating a single instance of a logical data structure with a pool, partitions the data structure further into micro-pools.

We use the term macro-pool to refer to the pool associated with the data structure instance and the term 'micro-pool' to the pool entity associated with the sub-part of a data structure. Each micro-pool is associated with per-pool specific time stamp. The sub-partitioning of a macro-pool into micro-pools is guided by a Micro-pool Partitioning Function (MPF). The partition function is chosen based on the update pattern of the data structure under consideration.

Micro-pool management costs increase with the number of micro-pools since each micro-pool updated by a committing transaction needs to have its GTS incremented at the time of commit and validation of the read-set data items belonging to a given micro-pool requires reading the corresponding micro-pool GTS. A fine-grained micro-pool, which avoids non-conflicting GTS updates thereby reducing GTS miss costs, incurs greater micro-pool management costs, whereas a coarse-grained micro-pool scheme while incurring low micro-pool management overheads, cannot help reduce GTS miss costs effectively.

In our current implementation we use a simple partitioning function based on *update patterns* in connected data structures such as lists and trees, which provides adequate performance. Typically a connected data structure is grown by attaching the newly allocated sub-part object to the pre-existing sub-part objects of a given data structure instance. Hence we propose a simple clustering algorithm to group a set of connected data items of a data structure into a micro-pool. In the proposed algorithm, a newly created sub-part which is getting connected to an existing connected data structure through the pointer fields of existing sub-parts is allocated the same micro-pool id as the referring pointers, subject to a size constraint which ensures that each micro-pool size is limited to k sub-parts where the value of k is an integer based on the choice of the micro-pool granularity desired. The intuition behind our simple partitioning scheme is that referrer and referent data are typically updated together in a connected data structure and *many of the data structure updates are localized updates involving only a few sub-parts*. We note that the selection of partition function is a major factor in determining the effectiveness of SPTS. We plan to study the selection of partitioning functions suitable for complex data structures as part of future work.

3.3.3 Implementation of SPTS Transformation

Remember that GTS is read at the beginning of all transactions and is updated at the end by a committing (write) transaction. If different time stamps are used to cover different groups of shared data items, the transaction needs to know at the entry to the atomic section the time stamps that need to be read. Since it is not possible to know exactly all the shared data that would be accessed inside an atomic section, we start by reading the set of time stamps associated with all live-in shared data. We refer to the set of time stamps read at the beginning of the atomic section as the starting pool set (SP).

If there is a shared data access '*a*' inside the atomic section whose time stamp TS(a) is not a member of SP, then we not only need to read TS(a) but we also need to re-validate the entire read set accumulated till now so as to maintain a consistent snapshot of our shared data accesses. However validation can be done at per pool level by checking that none of the values in SP have changed. If the validation succeeds, then the transaction proceeds normally after adding TS(a) to SP. Else, the transaction is aborted and the transaction is restarted including TS(a) in SP.

For any data written inside the transaction, its corresponding TS is added to a set WTS. At commit time, unlike having to update only one GTS as in case of TL2, the transaction increments the time stamp for each member of WTS and updates the version numbers of write set data with the corresponding WTS member time stamp.

BM	Execution Time		% improvement in Execution Time	% Improvement in Average dcache miss Latency cycles
	TL2	TL2-opt		
Kmeans	20.21	18.85	6.73%	6.92%
Genome	15.51	13.84	10.71%	11.34%
Intruder	87.27	83.06	4.82%	5.35%
Ssca2	73.81	60.65	17.82%	18.37%
Vacation	64.04	61.73	3.61%	3.43%

Table – 11 SPTS Performance Improvements

3.3.4 Performance Results for SPTS

Table-11 shows the performance results for SPTS only for those 5 benchmarks, in which SPTS identified candidates. Compile time overheads due to SPTS ranged from 1.44% to 7.67%. SPTS improved the execution time of Kmeans, Genome, Ssca2, Intruder and Vacation. In Kmeans and Ssca2, SPTS assigned different time stamps to disjoint array sections, reducing the global time stamp misses, whereas micro-pools were identified by automatic micro pool allocation technique and were assigned different time stamps in Genome, Intruder and Vacation. We did not observe any significant impact on abort rates due to SPTS revalidation phase for our experimental benchmarks.

BM	Execution Time		% improvement in Execution Time	% Improvement in Average dcache miss Latency cycles
	TL2	TL2-opt		
Kmeans	20.21	13.52	33.12%	37.14%
Genome	15.51	12.67	18.32%	20.15%
Intruder	87.27	69.49	20.37%	21.59%
Ssca2	73.81	54.50	26.16%	29.83%
Vacation	64.04	50.75	20.76%	21.74%

Table 12 Combined Performance Results

3.4 Combined Performance Results of our Transformations

Table 12 shows the combined performance benefits of all our transformations. We leave out Labyrinth and Bayes where our transformations were not applicable or didn't give significant benefits. We find that a 18% – 33% improvement in execution time and 20% – 37% reduction in d-cache miss latency cycles is achieved with our transformations. In Table 13, we show the impact of our transformations on the STM_Ms component. For Intruder and Vacation, STM_Ms still remains high overall due to their high data conflicts. We report Lock_Ms and GTS_Ms components as percentage of STM_Ms for TL2 and TL2-opt in columns 4, 5, 6 and 7 of Table-13 respectively. We find that for 5 of the benchmarks, Lock_Ms are less than 30% of the STM Misses showing that LDC and RLAR are effective in reducing the lock access misses. SPTS helps to reduce global time stamp misses for Ssca2 and Genome. GTS_Ms still remains high for Ssca2 and hence we are investigating it further.

BM	STM_Ms % of TDML		Lock_Ms % of STM_Ms		GTS_Ms % of STM_Ms	
	TL2	TL2-OPT	TL2	TL2-OPT	TL2	TL2-OPT
Kmeans	82.5	54.7	51.3	10.7	16.1	11.9
Genome	46.6	34.1	50.6	21.8	23.4	14.2
Intruder	70.7	58.6	55.1	29.3	9.7	7.6
Vacation	92.8	81.2	54.1	29.5	1.2	0.2
Ssca2	51.1	29.8	34.1	12.1	38.1	19.6

Table 13 Impact of our transformations on STM_Ms

In addition to the above transformations, we have also implemented a Dynamic Allocation Optimization (DAO) to address *Dyn_AllocMs* component as described next.

3.5 Dynamic Memory Allocation Optimization (DAO)

Transactions containing dynamic memory allocations need to free the speculatively allocated memory when they abort. Such repeated heap allocations inside transactions which abort and then restart, lead to application overheads due to cache misses in memory allocator library. While specialized STM-aware memory allocators can avoid these overheads, STMs which depend on the standard malloc/free dynamic allocation package suffer from this issue.

Dynamic Allocation Optimization (DAO) targets those dynamic allocations inside a transaction where the allocation occurs inside the transaction without a corresponding free. The allocation is speculative and must be rolled back if the transaction aborts. However, since the aborting transaction is likely to be restarted, the speculative allocation can be cached on a free list which is private to the thread executing the aborting transaction. When the transaction re-executes, the allocation request can be served from the cached list. Both intruder and Yada benefit from DAO. The performance results reported in the paper do not include the performance improvements due to DAO.

3.6 Preserving the Semantics of the Atomic Sections

Any transformations introduced for STM applications needs to ensure that the original STM semantics are not violated. Our transformations are built on top of an existing word based STM (TL2). As TL2 preserves the atomic semantics, we should not introduce any transformations that violate the guarantees provided by the underlying STM. On this front, the underlying STM requires the guarantees that: (i) There is a locking discipline that maps each shared address accessed within atomic sections to a lock, and (ii) The locking discipline is consistent. Both these conditions are not violated by any of our transformations. LDC simply changes the lock location whereas RLAR only changes the locking granularity, while ensuring a consistent locking discipline. SPTS and DAO do not have any impact on the locking discipline. We omit a detailed correctness argument here due to space constraints.

4. Related Work

Due to space constraints, we mention closely related prior work here and cite the other prior art relevant to the broader theme addressed by this paper in Appendix. A detailed survey of TM can be found in [23]. There has been considerable work on improving STM performance [4, 5, 6, 10, 20]. However there has not been much work on investigating the cache performance of the applications on STMs. Some of the earlier works mention potential adverse impacts STM-specific activities may have on cache [3, 10, 22]. There has been work on compiler transformations to improve STM performance by reducing compiler over instrumentation and redundant barrier removal [16, 18, 21]. Techniques, such as the redundant barrier removal in a compiler driven STM, can indirectly reduce cache miss overheads by reducing the number of barriers. There has also been considerable work on reducing the STM overheads and effectively addressing the transactional conflicts [23]. However our compiler transformations being orthogonal to these schemes, are aimed at reducing the cache miss overheads directly and can co-exist with them.

5. Conclusion

In this paper, we studied the cache miss behavior of STM applications and identified the different components that contribute to the cache miss overheads in STM. We identified and evaluated a few specific compiler transformations to address the cache overheads in STM. Our proposed set of compiler transformations is a first step in the direction of improving the cache performance of STMs so that STMs can be adopted widely for performance critical mainstream software.

References

[1] M. Harley and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. ISCA-93.

[2] N. Shaved and D. Tutu. Software transactional memory. In PODC, Aug 1995.

[3] D. Dice, O. Shale, and N. Shaved. Transactional locking II. DISC 2006.

[4] P. Felber, C. Fetzer, and T. Riegel,. 2008. Dynamic performance tuning of word-based software transactional memory. In PPoPP '08. ACM, New York, 237-246.

[5] Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. PLDI 2006.

[6] R. Yoo, Y. Ni, A. Welc, B. Saha, A. Adl-Tabatabai, and H.S. Lee. 2008. Kicking the tires of software transactional memory: why the going gets tough. SPAA '08. ACM, NY, 265-274.

[7] C. Cascaval, C.Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. 2008. Software Transactional Memory: Why Is It Only a Research Toy?. Queue 6, 5 (Sep. 2008), 46-58.

[8] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In IISWC ’08, Sep 2008.

[9] R. Hundt, HP Caliper: A framework for performance analysis, IEEE Concurrency, pp 64–71, 2000

[10] D. Dice and N. Shavit. 2007. Understanding Tradeoffs in Software Transactional Memory. In CGO 2007.

[11] C. Lattner and V.Adve. 2005. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In PLDI 2005.

[12] J. Bobba, K.E.Moore, H.Volos, L.Yen, M.D.Hill, M.M.Swift,, and D.A.Wood. 2007. Performance pathologies in hardware transactional memory. In ISCA 2007,.

[13] http://www.open64.net/documentation

[14] A. R. Adl-Tabatabai, B. T. Lewis, V. S. Menon,et al. Compiler and runtime support for efficient software transactional memory. In PLDI 2006.

[15] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and Adl-Tabatabai, “Code Generation and Transformation for Transactional Memory Constructs in an Unmanaged Language,” in CGO 2007.

[16] T. Harris, M. Plesko, A. Shinar, and D. Tarditi, “Optimizing Memory Transactions,” in PLDI 2006,.

[17] B. Saha, A.-R. Adl-Tabatabai, et al, “McRT-STM: A High Performance Software Transactional Memory System For A Multi-Core Runtime,” in PPoPP 2006.

[18] M.Spear, M.Michael, M.L.Scott, and P.Wu. Reducing Memory Ordering Overheads in Software Transactional Memory. in CGO 2009.

[19] T.M. Chilimbi, M. D. Hill, and J. R. Larus. 1999. Cache-conscious structure layout, PLDI 1999.

[20] A.Dragojević, R.Guerraoui, and M.Kapalka. 2009. Stretching transactional memory. In PLDI '09.

[21] P.Wu, M.Michael, C.Von Praun, T.Nakaike, R.Bordawekar, H.Cain, C.Cascaval, S.Chatterjee, R.Chiras, M.Mergen, X.Shen, M.Spear, H.Y.Wang and K.Wang. 2009. Compiler and runtime techniques for STM transformation. Concurr. Comput. : Pract. Exper.

[22] C.Perfumo, N.Sönmez, S.Stipic, O.Unsal, A.Cristal, T.Harris, and M.Valero. 2008. The limits of software transactional memory (STM): dissecting Haskell STM applications on a many-core environment. CF-2008

[23] J.Larus, R.Rajwar. Transactional Memory. Morgan and Claypool Publishers

[24] G Chakrabarti and F.Chow, Structure layout transformation in open64, Open64 Workshop 2008

[25] Y.Wu and J.R.Larus. 1994. Static branch frequency and program profile analysis. In Proceedings of MICRO 2002, New York.

[26] T. M. Chilimbi, M. D. Hill, and J. R. Larus, “Cache-conscious structure layout,” in PLDI ’99, NY, USA, pp. 1–12, 1999.

[27] T. M. Chilimbi and R. Shaham, Cache-conscious coallocation of hot data streams, SIGPLAN Notices, pp. 252–262, 2006.

[28] Chilimbi, T. M. and Larus, J. R. 1998. Using generational garbage collection to implement cache-conscious data placement. ISMM '98. ACM, New York, NY, 37-48.

[29] M. F. Spear, M. M. Michael, and C. Praun. 2008. RingSTM: scalable transactions with a single atomic instruction. SPAA 2008, ACM, New York, NY, USA.

[30] Y Lev, V. Luchangco, V. Marathe, M. Moir, D.Nussbaum and M. Olszewski, Anatomy of a scalable software transactional memory. Transact 2009.

[31] T. Riegel, C. Fetzer, H. Sturzrehm, and P. Felber. 2007. From causal to z-linearizable transactional memory. PODC 2007. ACM, New York, NY, USA, 340-341.

[32] T. Riegel C. Fetzer, and P. Felber. Automatic data partitioning in software transactional memories. SPAA 2008