Determinism at Standard Library Level in Transactional Memory Based Applications

Vesna Smiljković[12], Osman Ünsal[1], Adrián Cristal[13], and Mateo Valero[12]

[1] Barcelona Supercomputing Center, Spain
[2] Universitat Politcnica de Catalunya, Spain
[3] IIIA - Artificial Intelligence Research Institute CSIC - Spanish National Research Council
{first}.{last}@bsc.es

Abstract. Deterministic execution of a multi-threaded application guarantees that threads access shared memory in the same order and the application gives the same output when it runs with the same input parameters. Determinism provides repeatability, which helps a programmer in testing and debugging. Additionally, Transactional Memory (TM) simplifies development of applications that use transactions (instead of locks) to synchronize accesses to shared memory. However, transactions that call standard library functions have to be serialized, and the serialization causes a deadlock when applications run with the deterministic systems proposed so far.
In this paper, we present DeTrans-lib, the first standard C library that provides deterministic execution of TM-based applications at user and standard-library level. DeTrans-lib avoids deadlocks by performing transaction serialization in deterministic order. We evaluate DeTrans-lib using benchmarks that perform I/O operations.

Keywords: transactional memory; determinism; standard library

1 Introduction

Transactional Memory (TM) [1][2] is a synchronization mechanism proposed as a replacement of traditional locking mechanism to simplify development of multi-threaded applications, allow threads to concurrently access shared memory, and avoid concurrency bugs. In the last two decades TM has been in the focus of researchers, who provided implementations in software [3][4][5] and hardware integrated in mainstream Intel [6] and IBM [7] processors.

Although TM increases productivity and developers might avoid the concurrency bugs common for lock-based applications, many bugs are still hard to reproduce, find and resolve. Like in any other multi-threaded application, in a TM-based application threads run concurrently and interleave non-deterministically when they access shared memory. This might cause a different output every time an application is executed, which makes testing difficult. What makes debugging difficult is that a bug might appear in one execution, but be hidden in another.

The final publication is available at Springer via http://dx.doi.org/10.1007/s10766-015-0383-4

To make every execution of a TM-based application repeatable, Smiljkovic et al. [8] and Ravichandran et al. [9] proposed systems for deterministic execution – DeTrans and DeSTM, respectively. They guarantee that threads always interleave in the same order and the application always gives the same output for the same input parameters. However, they ensure deterministic multithreading only at user level, and might cause a deadlock when transactions call standard library functions.

Standard libraries abstract and simplify the access to services of the operating system and encapsulate shared structures, e.g. file structures. In general, TM is not able to track accesses to shared structures in a standard library or kernel; therefore, a transaction that invokes a standard library call or a system call has to be serialized. If an application is executed deterministically, one thread might wait to be serialized and another to be executed in deterministic order. Since the deterministic system and TM try to enforce a different order of threads execution, they cause a deadlock.

In this work we present *DeTrans-lib*, a TM-based standard C library (*libc*) that ensures deterministic execution of multi-threaded applications at user and standard-library level. It is based on a runtime (DeTrans [8]) that provides deterministic execution even in the presence of data races, and a TM-based libc (TM-dietlibc [10]) that allows transactions to concurrently execute libc functions, and serializes them only if the libc function invokes a system call. DeTrans-lib ensures that transactions invoke system calls in deterministic order, which prevents from having deadlocks caused by serialization and deterministic execution.

The contributions of our work are:

- DeTrans-lib is the first TM-based libc that provides deterministic multithreading. We port a deterministic system (DeTrans) in a libc (TM-dietlibc) to ensure deterministic execution of TM-based applications at user and standard-library level.
- DeTrans-lib avoids deadlocks caused by busy-waiting in serialization (enforced by TM-dietlibc due to system calls) and busy-waiting in deterministic execution (enforced by DeTrans). With DeTrans-lib, threads invoke system calls in deterministic order.
- We discuss the importance of supporting serialization and other ad hoc synchronizations in state-of-the-art systems for deterministic multithreading.

We verify the correctness of the DeTrans-lib implementation by using stress test Racey [11][12]. For the evaluation, we use benchmarks that invoke libc calls with I/O operations: microbenchmarks from [10], and modified TioBench [13].

2 Related Work

Recent work on deterministic multithreading ensures that when a program runs with the same input parameters it always gives the same output. When a system guarantees deterministic execution of the user-level code, we call this *user-level determinism*, and in case of deterministic execution when threads concurrently

invoke standard library calls and system calls, we call it *standard-library-level determinism* and *OS-level determinism*, respectively. Further more, *strong determinism* is deterministic execution even in the presence of data races, and *weak determinism* is deterministic execution of data-race-free programs.

In the rest of this section we give a brief overview of the systems for deterministic multithreading that are related to our work.

Olszewski et al. [14] propose Kendo, a software implementation for weak determinism of lock-based applications. Each thread has a private logical clock that counts logical time when a thread acquires a lock. A thread can execute a critical section only if the lock is available in both logical and physical time.

Devietti et al. (DMP [15], RCDC [16]) and Bergan et al. (CoreDet [17]) propose several software-only, hardware-only and software-hardware implementations to ensure determinism. The basic implementation is deterministic serialization of lock-based applications – threads execute parts of code in round-robin order. Parallelism is improved by (i) using a shared memory ownership table (strong determinism); (ii) maintaining thread-local store buffers that each thread copies to shared memory when parallel execution is over (strong determinism), or (iii) tracking happens-before dependencies of lock acquisitions (weak determinism).

Grace (Berger et al. [18]) and Dthreads (Liu et al. [19]) are software implementations for strongly deterministic execution of lock-based applications. Threads run concurrently and work on private memory pages, which are copied to shared pages in round-robin order at synchronization points.

In contrast to the systems for deterministic execution of lock-based multithreaded programs (i.e. multi-threaded programs that use `pthread`[4] calls as synchronization points), DeTrans (Smiljkovic et al. [8]) and DeSTM (Ravichandran et al. [9]) propose deterministic execution of TM-based programs.

DeTrans is based on a double-barrier technique and guarantees strong determinism of TM applications. Two barriers ensure that application code outside of transactions is executed serially, and transactions are executed in parallel, respecting the round-robin commit order.

On the other hand, DeSTM relaxes the barriers by allowing some transactions to pass the barriers earlier than others, i.e. to start their execution earlier and to start with the commit phase earlier. With the relaxed double-barrier technique, DeSTM provides weak determinism.

Dobel et al. [20] propose RomainMT – an operating system service that ensures redundant multithreading of lock-based applications. An application thread and its redundant thread acquire and release the same locks, and perform the same externalization events, e.g. system calls. During execution, the architectural state of the threads is compared and checked for faults. RomainTM is a fault-tolerant technique and cannot be used for testing and debugging since it does not provide repeatability – different runs of an application with the same input parameters might give a different output.

[4] pthread_mutex_lock/unlock, pthread_barrier_wait, pthread_create, etc.

From the systems mentioned so far, only CoreDet and Dthreads handle libc and system calls during deterministic execution. CoreDet either serializes libc calls, or provides its own version of libc functions. Dthreads allows invoking some system calls, and each thread maintains its private copy of a shared libc structure. However, these systems cannot be used for TM-based applications due to differences in synchronization mechanisms [8].

Deterministic operating systems Determinator [21] and dOS [22] enforce determinism at user and OS level by using page protection hardware. Instead of significant changes of an OS, DeTrans-lib is implemented at user level, can be used with any OS, ensures libc-level determinism and guarantees deterministic order of invoking system calls in an application.

In the next section we discuss in detail about DeTrans[5] and the issues that appear when a TM-based application calls libc functions within transactions while running deterministically.

3 Standard library calls in deterministic execution

In a TM-based application (like in any other multi-threaded application) threads run concurrently and interleave non-deterministically when they access shared memory. As a result, even for the same input parameters the application might give a different output, which makes testing and debugging difficult. To avoid this, researchers have proposed DeTrans, the runtime system for deterministic execution of TM-based applications.

DeTrans implements a *double-barrier technique* and *deterministic-token passing* (Figure 1(a)). It runs transactions in parallel and code outside of transactions (non-transactional code) serially. In the parallel phase threads execute transactions concurrently and commit them in round-robin order, and the thread that holds the deterministic token commits first. In the serial phase threads are executed in round-robin order, and only the thread that holds the deterministic token is executed. However, invoking a standard library while threads are executing transactions (the parallel phase) might influence deterministic order enforced by DeTrans.

Standard libraries encapsulate shared structures, and abstract the access to services of the operating system. In general, TM is not able to track accesses to shared structures in a libc or kernel. Therefore, a transaction that calls a standard library function has to be serialized, i.e. to be executed as the only running transaction in the system.

To provide more parallelism in TM-based applications and reduce the number of serialized transactions, researchers proposed TM-dietlibc [10], i.e. a TM-aware libc that allows transactions to call libc functions and to execute them concurrently. Only when a libc function invokes a system call, the transaction has to be serialized because of with non-reversible side effects in kernel.

[5] In our work we focus on this system for deterministic multithreading since it guarantees determinism even in the presence of data races, which are considered to be bugs that are hard to reproduce and resolve [23].

Listing 1.1 shows a TM-dietlibc's implementation of the I/O function for writing data to a file. The function either performs the I/O completely internally, (it buffers the data if the buffer is not full – line 5), or it goes to kernel (it invokes a system call `write` to write the data to the file – line 11). In the latter case, a transaction first has to wait for all other transactions to finish their execution (Listing 1.2 line 8), then it gets restarted (Listing 1.1 line 9) as the only transaction in the system, and invokes the system call.

```
1  size_t fwrite(void *ptr, size_t size, size_t n, FILE *fp) {
2    size_t len = size*n;
3    tx_atomic { // transaction starts here
4      if(!fp->buf.full()) {
5        memcpy(fp->buf, ptr, len);
6      }
7      else {
8       if(!serialized())
9         tx_restart(); // go back to line 3
10      else
11        len = write(fp->fd, ptr, len); // system call
12     }
13   } // transaction commits here
14   return len;
15 }
```

Listing 1.1. The implementation of `fwrite` in TM-dietlibc.

```
1  int serialized() {
2
3
4   //check if tx is already serialized
5   if (this->tx->isSerialized)
6    return 1;
7   //wait until others finish
8   while (others.areExecuting()){}
9   this->tx->isSerialized = 1;
10  return 0;
11 }
```

Listing 1.2. Serialization in TM-dietlibc

```
1  int serialized() {
2   // wait for the token
3   while (this != token.owner) {}
4   //check if tx is already serialized
5   if (this->tx->isSerialized)
6    return 1;
7   //kill other txs
8   kill(others);
9   this->tx->isSerialized = 1;
10  return 0;
11 }
```

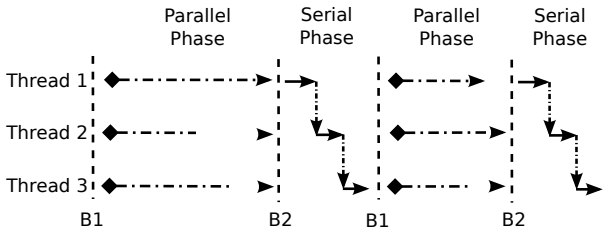
Listing 1.3. Serialization in DeTrans-lib

Figure 1(b) shows an example of a thread (`Thread1`) calling a libc function that invokes a system call within a transaction while running deterministically with DeTrans. Before the call, the transaction has to be serialized – it waits for other transactions to commit their changes and then to run as the only transaction in the system. However, it holds the deterministic token and the other transactions (executed by `Thread2` and `Thread3`) wait for `Thread1` to commit its transactions and pass the token.

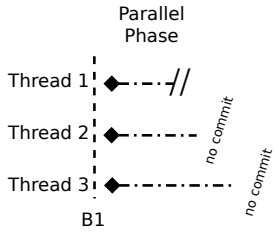
Our observation is that deterministic multithreading makes deadlocks out of busy-waiting in transaction serialization (Listing 1.2 line 8) and that a deterministic system has to be aware of synchronizations that are necessary to invoke libc calls within transactions in TM-based applications.

4 Implementation of DeTrans-lib

In this work we propose DeTrans-lib, i.e. the first TM-aware libc that provides deterministic execution of TM-based applications. We port deterministic system



(a) The double-barrier technique and token passing.



(b) Deterministic execution when a transaction invokes a system call.

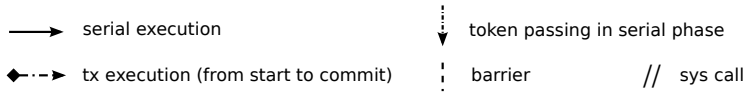


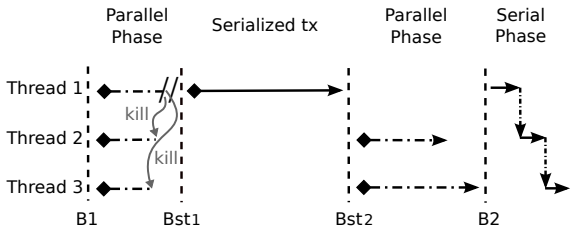
Fig. 1. DeTrans implements the double-barrier technique and passes the deterministic token in round-robin order to ensure deterministic execution (a). DeTrans causes a deadlock when a transaction has to be serialized (b).

DeTrans, which guarantees deterministic execution at user level, in TM-dietlibc to provide determinism at libc level. We extend DeTrans to support libc functions that internally invoke system calls.

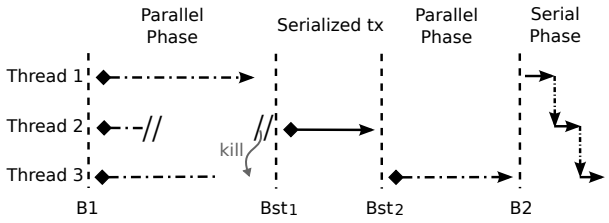
Listing 1.3 shows a modified version of the function for transaction serialization. To avoid a deadlock caused by busy-waiting in serialization (enforced by TM-dietlibc due to system calls) and busy-waiting for the deterministic token (enforced by DeTrans), with DeTrans-lib threads invoke system calls in deterministic order. Only the transaction which thread holds the deterministic token (the token owner) can invoke a system call (line 3). When it gets the token, the transaction kills other running transactions (line 8) instead of waiting for them to finish, which was the original implementation and a cause of deadlocks.

Figure 2 shows examples of threads invoking system calls within transactions while running deterministically with DeTrans-lib.

In Figure 2(a) the token-owner thread (`Thread1`) – before invoking a system call – kills other transactions, runs as the only transaction in the system, passes the deterministic token to the next thread and the other threads re-execute their transactions in parallel.



(a) The token owner invokes a system call.



(b) A non-token owner invokes a system call.

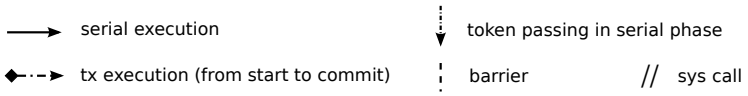


Fig. 2. DeTrans-lib serializes transactions and invokes system calls in deterministic order.

In Figure 2(b) one of the threads that is not the token owner (e.g. `Thread2`), before it invokes a system call, it waits for the deterministic token, kills the only remaining active transaction and re-executes its transaction. After the serial execution, the remaining transaction can be executed.

`Bst1` and `Bst2` are the barriers that separate serial execution of a transaction from the parallel phase. For one round of token passing, from 0 to N transactions can be serialized, where N is the number of running threads in the round.

With DeTrans-lib, any thread can invoke a system call. However, since transactions have to be serialized, the order of serialization and system call invocation is deterministic and repeatable in every execution of an application.

5 Evaluation

We evaluate DeTrans-lib with the benchmarks that call libc I/O functions, using 2 Intel Xeon E5405 processors each with 4 cores (8 cores in total) with 4GiB RAM, and running at 2.00GHz. We compile the benchmark with GCC 4.9 and link it with TinySTM [4] 1.0.5.

5.1 Methodology

First, we verify the correctness of the DeTrans-lib implementation by using the stress test Racey [11][12]. We took the modified version that supports transactions and was used in [8]. Additionally, we modified the test to invoke libc. The test calculates a signature which value depends on the order of threads accessing and updating a shared array. The test was executed 100 times with DeTrans-lib by multiple threads and gave the same signature for the same input parameters.

TM benchmarks implemented for TM evaluations are either without standard library calls (STAMP [24] and Eigenbench [25]) or they postpone these calls until the serial phase of execution (Atomic Quake [26] and Memcached [27]). As a consequence, we cannot use any of these benchmarks to evaluate DeTrans-lib. Instead, we use microbenchmarks from [10] and modified TioBench [13], where transactions perform I/O on a single shared file and occasionally have to be serialized due to system calls.

The microbenchmarks perform (i) I/O functions `fgetc`, `fgets`, `fread`, `fputc`, and `fputs` within transactions, and (ii) calculations on thread-local variables out of transactions.

Our modified version of TioBench performs `fopen`, `fclose`, `fseek`, `fread` and `fwrite` instead of invoking system calls directly. Multiple threads execute: (i) sequential write (writes data to a file starting from the beginning), (ii) random write (writes data to a random position of a file), (iii) sequential read (reads data from a file starting from the beginning), and (iv) random read (reads data from a random position of a file).

To measure performance we run the benchmarks multiple times with 1, 2, 4 and 8 threads, and calculate the geometric mean of the slowdown of deterministic execution in comparison to the original (non-deterministic) single-threaded execution.

5.2 Results

Figure 3 shows the performance of the microbenchmarks and TioBench running non-deterministically (with TM-dietlibc), and deterministically (with DeTrans-libc).

The microbenchmarks execute short transactions with I/O operations (reading/writing a single character per transaction), and intensive calculations on thread-local variables out of transactions. Transactions conflict occasionally and the benchmarks scale while running non-deterministically. Since the benchmarks spend most of the execution time out of transactions, and that is the part that DeTrans-lib executes serially, DeTrans-lib slows down the single-threaded non-deterministic execution by 1.12x, 1.62x, 1.94x, and 4.53x for 1, 2, 4, and 8 threads, respectively.

TioBench executes long transactions (reading/writing 128 characters per transaction) and no other operations are performed out of transactions. The transactions conflict frequently and the benchmark does not scale while running non-deterministically. Since the benchmark spends most of the execution

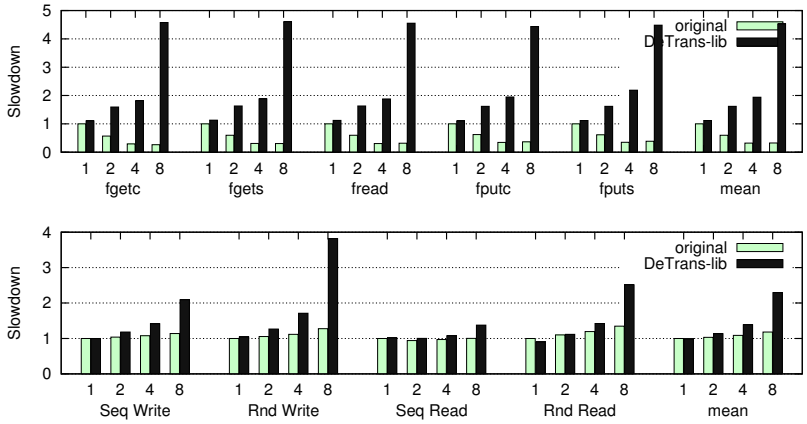


Fig. 3. Slowdown of deterministic execution of microbenchmarks and TioBench

time within transactions, and that is the part that DeTrans-lib executes in parallel, DeTrans-lib has low overhead and slows down the single-threaded nondeterministic execution by 0.99x, 1.14x, 1.39x, and 2.29x for 1, 2, 4, and 8 threads, respectively. Random write and read operations have higher overhead than the average because the benchmark invokes system calls to update the I/O buffer with the content from the random position of the file, and DeTrans-lib serializes transactions that invoke system calls.

The overhead of DeTrans-lib depends on the benchmark implementation. First, if threads perform time-intensive operations outside of transactions, which DeTrans-lib executes serially, than the overhead of deterministic execution is high. Second, if a benchmark spends most of the execution time within transactions, which DeTrans-lib executes in parallel, than the performance is close to the performance of the original execution. Third, increasing the number of serialized transactions causes performance degradation of deterministic execution.

6 Discussion

TM requires that transactions that invoke libc and system calls are serialized, and the serialization was implemented as ad hoc synchronization. In general, ad hoc synchronization is common and error-prone. According to [28], the studied benchmarks have tens of ad hoc synchronizations, and some of them cause incorrect execution and performance degradation.

State-of-the-art systems for deterministic multithreading do not support ad hoc synchronization mechanisms that might be used in external libraries and benchmarks. This causes deadlocks in the execution.

There are two main sources of ad hoc synchronization in the systems for deterministic multithreading that might cause deadlocks. First, some of the systems

for deterministic multithreading (e.g. DeTrans) serialize the code that programmers wrote as parallel and assumed that it will always be executed as parallel when running with multiple threads. In the serialized execution, only one thread executes at a time. Other threads wait for their turn (busy-waiting on the deterministic token in the serial phase). Second, some of the systems for deterministic multithreading (e.g. Dthreads) postpone writes to shared memory, i.e. the updates are buffered and copied to shared memory only when the thread holds the deterministic token (busy-waiting on the token at commit time of the parallel phase). Benchmarks run deterministically well only if there is no other busy-waiting during the execution, either in the external libraries or the benchmark itself.

```
1  main() {
2    pthread_create(thread1, foo1);
3    pthread_create(thread2, foo2);
4    pthread_join(thread1);
5    pthread_join(thread2);
6  }
7
8  foo1() {
9    while (!start) {}
10   // execute here something in parallel with foo2()
11 }
12
13 foo2() {
14   start = 1;
15   // execute here something in parallel with foo1()
16 }
```

Listing 1.4. A common example of an ad hoc synchronization mechanism.

Listing 1.4 shows a common example of an ad hoc synchronization used in benchmarks: a shared variable `start` synchronizes two threads to execute the code in `foo1()` and `foo2()` in parallel. If the code is executed deterministically and serially where `thread1` executes first, the program will never finish its execution. If the code is executed deterministically and in parallel where threads buffer their updates, and `thread1` holds the token, the program will never finish its execution since `thread2` cannot commit its update of the `start` variable without the token.

To allow complex programs to run deterministically, the state-of-the-art systems for deterministic multithreading have to support ad hoc synchronizations, and to extend their usage from the programs that exclusively use transactions, locks, conditions, and barriers, to the complex real-world applications with various synchronization mechanisms.

7 Conclusion and Future Work

In this paper, we presented DeTrans-lib – the first libc that provides deterministic execution of TM-based applications at user and standard-library level. Since libc functions occasionally invoke system calls and transactions that invoke system calls have to be serialized, DeTrans-lib provides a mechanism to support serialization and to avoid deadlocks caused by the ad hoc synchronization used

for the serialization. In future work we want to extend our system to support ad hoc synchronizations in general, and to use it for deterministic execution of complex real-world applications.

References

1. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the 20th annual international symposium on computer architecture. ISCA '93 (1993) 289–300
2. Harris, T., Larus, J., Rajwar, R.: Transactional memory. Synthesis Lectures on Computer Architecture **5**(1) (2010) 1–263
3. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing. PODC '95, New York, NY, USA, ACM (1995) 204–213
4. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. PPoPP '08, New York, NY, USA, ACM (2008) 237–246
5. Spear, M.F., Dalessandro, L., Marathe, V.J., Scott, M.L.: A comprehensive strategy for contention management in software transactional memory. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '09, New York, NY, USA, ACM (2009) 141–150
6. Karnagel, T., Dementiev, R., Rajwar, R., Lai, K., Legler, T., Schlegel, B., Lehner, W.: Improving in-memory database index performance with intel transactional synchronization extensions. In: High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on. (Feb 2014) 476–487
7. Cain, H.W., Michael, M.M., Frey, B., May, C., Williams, D., Le, H.: Robust architectural support for transactional memory in the power architecture. In: Proceedings of the 40th Annual International Symposium on Computer Architecture. ISCA '13, New York, NY, USA, ACM (2013) 225–236
8. Smiljkovic, V., Stipic, S., Fetzer, C., Unsal, O., Cristal, A., Valero, M.: Detrans: Deterministic and parallel execution of transactions. In: Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on. (Oct 2014) 152–159
9. Ravichandran, K., Gavrilovska, A., Pande, S.: DeSTM: Harnessing determinism in STMs for application development. In: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation. PACT '14, New York, NY, USA, ACM (2014) 213–224
10. Smiljkovic, V., Nowack, M., Miletic, N., Harris, T., Unsal, O., Cristal, A., Valero, M.: TM-dietlibc: A tm-aware real-world system library. In: Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on. (May 2013) 1266–1274
11. Hill, M.D., Xu, M.: Racey: A stress test for deterministic execution
12. Xu, M., Bodik, R., Hill, M.D.: A" flight data recorder" for enabling full-system multiprocessor deterministic replay. In: Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on, IEEE (2003) 122–133
13. Vianney, D.: Tiobench benchmark - ltc linux performance team. http://linuxperf.sourceforge.net/tiobench/tiobench.php

14. Olszewski, M., Ansel, J., Amarasinghe, S.: Kendo: efficient deterministic multithreading in software. SIGPLAN Not. **44**(3) (march 2009) 97–108
15. Devietti, J., Lucia, B., Ceze, L., Oskin, M.: DMP: deterministic shared memory multiprocessing. In: Proceedings of the 14th international conference on Architectural support for programming languages and operating systems. ASPLOS '09, New York, NY, USA, ACM (2009) 85–96
16. Devietti, J., Nelson, J., Bergan, T., Ceze, L., Grossman, D.: RCDC: a relaxed consistency deterministic computer. SIGPLAN Not. **47**(4) (2011) 67–78
17. Bergan, T., Anderson, O., Devietti, J., Ceze, L., Grossman, D.: CoreDet: a compiler and runtime system for deterministic multithreaded execution. In: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems. ASPLOS '10, New York, NY, USA, ACM (2010) 53–64
18. Berger, E., Yang, T., Liu, T., Novark, G.: Grace: Safe multithreaded programming for c/c++. In: ACM SIGPLAN Notices. Volume 44., ACM (2009) 81–96
19. Liu, T., Curtsinger, C., Berger, E.: Dthreads: Efficient deterministic multithreading. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, ACM (2011) 327–336
20. Döbel, B., Härtig, H.: Can we put concurrency back into redundant multithreading? In: Proceedings of the 14th International Conference on Embedded Software. EMSOFT '14, New York, NY, USA, ACM (2014) 19:1–19:10
21. Aviram, A., Weng, S.C., Hu, S., Ford, B.: Efficient system-enforced deterministic parallelism. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. OSDI'10, Berkeley, CA, USA, USENIX Association (2010) 1–16
22. Bergan, T., Hunt, N., Ceze, L., Gribble, S.D.: Deterministic process groups in dos. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. OSDI'10, Berkeley, CA, USA, USENIX Association (2010) 1–16
23. Serebryany, K., Iskhodzhanov, T.: Threadsanitizer: Data race detection in practice. In: Proceedings of the Workshop on Binary Instrumentation and Applications. WBIA '09, New York, NY, USA, ACM (2009) 62–71
24. Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on, IEEE 35–46
25. Hong, S., Oguntebi, T., Casper, J., Bronson, N., Kozyrakis, C., Olukotun, K.: Eigenbench: A simple exploration tool for orthogonal tm characteristics. In: Proceedings of the IEEE International Symposium on Workload Characterization. IISWC '10, Washington, DC, USA, IEEE Computer Society (2010) 1–11
26. Zyulkyarov, F., Gajinov, V., Unsal, O.S., Cristal, A., Ayguadé, E., Harris, T., Valero, M.: Atomic quake: Using transactional memory in an interactive multiplayer game server. SIGPLAN Not. **44**(4) (February 2009) 25–34
27. Ruan, W., Vyas, T., Liu, Y., Spear, M.: Transactionalizing legacy code: An experience report using gcc and memcached. In: Proceedings of the 19th international conference on Architectural support for programming languages and operating systems, ACM (2014) 399–412
28. Xiong, W., Park, S., Zhang, J., Zhou, Y., Ma, Z.: Ad hoc synchronization considered harmful. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. OSDI'10, Berkeley, CA, USA, USENIX Association (2010) 1–8