Extending the Applicability of Deterministic Multithreading

Vesna Nowack (born Smiljković)

Department of Computer Architecture

Universitat Politècnica de Catalunya

A dissertation submitted in fulfillment of
the requirements for the degree of

Doctor of Philosophy / Doctor per la UPC

November 2015

Supervisor: Osman S. Ünsal
Co-Supervisors: Adrián Cristal, Mateo Valero

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola de Doctorat

Curso académico:

Acta de calificación de tesis doctoral

Nombre y apellidos

Programa de doctorado

Unidad estructural responsable del programa

Resolución del Tribunal

Reunido el Tribunal designado a tal efecto, el doctorando / la doctoranda expone el tema de la su tesis doctoral titulada _____

_____.

Acabada la lectura y después de dar respuesta a las cuestiones formuladas por los miembros titulares del tribunal, éste otorga la calificación:

☐ NO APTO ☐ APROBADO ☐ NOTABLE ☐ SOBRESALIENTE

(Nombre, apellidos y firma) Presidente/a		(Nombre, apellidos y firma) Secretario/a
(Nombre, apellidos y firma) Vocal	(Nombre, apellidos y firma) Vocal	(Nombre, apellidos y firma) Vocal

_____, _____ de _____ de _____

El resultado del escrutinio de los votos emitidos por los miembros titulares del tribunal, efectuado por la Escuela de Doctorado, a instancia de la Comisión de Doctorado de la UPC, otorga la MENCIÓN CUM LAUDE:

☐ SÍ ☐ NO

(Nombre, apellidos y firma) Presidente de la Comisión Permanente de la Escuela de Doctorado	(Nombre, apellidos y firma) Secretaria de la Comisión Permanente de la Escuela de Doctorado

Barcelona a _____ de _____ de _____

Acknowledgements

Fist of all, I would like to express my sincere gratitude to my supervisors Dr. Osman Ünsal, Dr. Adrián Cristal, and Prof. Mateo Valero for giving me the opportunity to enrol in a PhD program at Universitat Politècnica de Catalunya (UPC) and to perform research in Barcelona Supercomputing Center (BSC). I would like to thank Osman and Adrián for their ideas, guidance, motivation, and expertise. They encouraged my research and helped me making decisions important for my professional and personal life.

I would also like to thank Prof. Veljko Milutinović for believing in Nebojša Miletić, Milan Pavlović, and me, and for helping us to get to BSC in the fist place. Many thanks go to Nebojša and Milan for the great idea to move to Barcelona and enrol in this PhD program together.

During my PhD studies, I had a great opportunity to meet incredible and enthusiastic researchers. A few of them contributed and influenced my thesis significantly. I am very thankful to Dr. Tim Harris for his guidance, suggestions and all his help while working on a joint publication. I would also like to thank Prof. Christof Fetzer, who mentored me during my three-month stay in the System Engineering (SE) group at Technische Universität Dresden. He introduced me to deterministic multithreading and motivated me to continue my PhD research in that direction. I would like to acknowledge the colleagues from the SE group for being interested in my work, providing me thoughtful questions and useful suggestions, but also for making my stay comfortable and enjoyable in any possible way. I am also extremely grateful to Timothy Merrifield and Prof. Joseph Devietti for sharing their ideas and technical knowledge about deterministic multithreading with me.

I would like to thank pre-dissertation defence committee members Prof. Xavier Martorell, Dr. Adrià Armejach, and Dr. Miquel Moretó, and external reviewers Prof. Pascal Felber and Dr. Pramod Bhatotia for the time and effort they invested to read the thesis and to make suggestions for its improvement.

I would like to acknowledge the PhD students and the staff members who made my stay in BSC successful and pleasant. First, a huge thanks goes to Srđan Stipić for collaboration on several papers, long discussions, programming and

debugging tips and tricks, and, most of all, for being my friend. I was also lucky to share the ideas, doubts, thoughts, the good and bad times with the office mates: Adrià Armejach, Azam Seyedi, Cristian Perfumo, Daniel Nemirovsky, Damián Roca, Ferad Zyulkyarov, Gokcen Kestor, Gulay Yalcin, Ivan Ratković, Javier Arias, Nehir Sonmez, Nikola Marković, Milan Stanić, Milovan Djurić, Oriol Arcas, Oscar Palomar, Saša Tomić, Timothy Hayes, Vasileios Karakostas, Vladimir Gajinov, Gina Alioto, and many others. I am also thankful to other BSC people for being my colleagues and friends: Jelena Koldan, Ana Jokanović, Roberta Piscitelli, German Rodriguez, and Daniel Cabrera. I sincerely thank you all for the great moments we had together.

I would also like to acknowledge Maja Rajić and Verònica Torras for believing in me, hearing my thoughts, discussing various topics, and especially for proving that the distance cannot hurt our friendship. Maja, thank you for being my best friend and for sharing many extraordinary moments with me in the last twenty years. Vero, thank you for being "mi mejora amiga" and for correcting my Spanish. With you, Barcelona has been and always will be my home.

I feel very lucky to be a part of a wonderful family in Serbia. My mum Slavica, dad Milan, sister Ana, and nephew Aki have always been there for me to share my success and to count on when times were rough. They, and many other family members and friends, have been the reason to spend many vacations in Serbia. Thank you all for your unconditional love, support, and care.

Last but not the least, I would like to give my special thanks to my husband, Martin, for his love, patience, and understanding over the past several years. His optimism, positive attitude, great ideas and the ability to solve problems efficiently have helped me in many situations. Thanks to him I have made steps in my PhD and personal life that I thought I was never able to do. Many thanks to you and our daughter Leona for every smile you show me, every hug you give me, and every dance we take together.

In conclusion, this research was financially supported by Barcelona Supercomputing Center and BSC-Microsoft Research Center, Universitat Politècnica de Catalunya (Barcelona Tech) through the FPI-UPC scholarship, the Spanish Ministry of Education, the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC), the VELOX FP7 project, and the Euro-TM COST Action.

Abstract

With the increased number of cores on a single processor chip, an application can achieve good performance if it splits the execution into multiple threads that run on multiple cores at the same time. To synchronize threads, Transactional Memory (TM) allows them to concurrently execute sections of code (*transactions*) with accesses to shared memory, and requires re-execution of one of the transactions in case of a conflicting access.

Even though parallel programming with TM is simpler and less error-prone than with the traditional locking mechanism, concurrent programming errors are hard to avoid in general. The reason is that threads run in parallel and might interleave in nondeterministic order. As a consequence, an error can occur in one execution but be hidden in another (which makes debugging hard), and the application output might vary (which makes testing and replica-based fault tolerance hard).

To help programmers in testing, debugging and providing fault tolerance, researchers have proposed *deterministic multithreading*, which guarantees that application threads access shared memory in the same order and the application gives the same output whenever it runs with the same input parameters.

In this thesis we present *DeTrans*, a system for deterministic multithreading in transactional applications. DeTrans ensures determinism even in the presence of data races, by executing non-transactional code serially and transactions in parallel. We compare DeTrans with Dthreads, a widely-used deterministic system for lock-based applications, and analyse sources of the overhead caused by deterministic execution. Instead of using memory protection hardware and operating system facilities, DeTrans exploits properties of TM implemented in software and outperforms Dthreads.

To allow transactions to invoke standard library functions while running deterministically and to increase parallelism, this thesis proposes *TM-dietlibc*, a TM-aware standard library. Our experience in modifying a lock-based standard library in order to integrate it in a TM system is applicable for any TM-aware software. TM-dietlibc provides concurrent execution of standard library functions and only in a few cases the execution switches to serial. In

comparison to completely serialized execution, TM-dietlibc shows high scalability and performance improvement for benchmarks with short transactions and low contention.

Serialization of transactions – which is still required for transactions in TM-dietlibc with non-reversible side effects – might enforce an order of threads execution different from the one enforced by a deterministic system, causing a deadlock. By porting deterministic system DeTrans in TM-dietlibc, we ensure deterministic multithreading at application and standard-library level, and avoid deadlocks by serializing transactions in deterministic order.

In this thesis we also discuss a common limitation of deterministic systems – ad hoc synchronization. Ad hoc synchronization is in general widely used, but similarly to transaction serialization, it might be prone to deadlocks during deterministic execution. We use hardware performance counters to identify synchronization loops at runtime and to avoid deadlocks by dynamically (but deterministically) changing the order of threads execution.

With the techniques mentioned above, this thesis shows how we extend the applicability of deterministic multithreading by supporting transactions, standard library calls, system calls, and ad hoc synchronization, which allows a programmer to apply deterministic multithreading on real-world applications with various synchronization mechanisms.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Parallel Programming

For more than a decade, hardware manufacturers have been integrating multiple cores on a single processor chip. Applications benefit from the parallel hardware if their execution is split into control flows (*threads*) that are executed on multiple cores at the same time [77]. In shared memory architecture, threads communicate by reading and writing to shared variables, and without proper synchronization accesses to shared memory might lead to a *data race*. A data race occurs when two or more threads access the same shared variable concurrently, and at least one of them is a write, which produces unwanted and unpredictable effects on application execution.

A traditional approach to synchronize shared memory accesses and avoid data races is a locking mechanism. A programmer identifies a part of application code (a *critical section*) with accesses to shared memory and uses a locking variable to ensure *mutual exclusion*, i.e. only one thread at a time enters and executes the critical section. Using locks that protect critical sections with many instructions (*coarse-grained* locks) makes

code development and maintenance easier, but extensive mutual exclusion hurts application performance. On the other hand, using locks that protect critical sections with a few instructions (*fine-grained* locks) does not affect application performance, but the programmer puts a lot of effort to avoid concurrency bugs.

In general, programming with locks can cause data races (due to shared memory accesses executed out of critical sections), deadlocks (due to incorrect order of lock acquisitions) or erroneous execution (due to multiple releases of locks) [77]. Additionally, locks are not composable [41], and combining smaller critical sections into bigger ones requires programmer's knowledge and understanding of locks' ordering and their hierarchy in an application.

1.2 Transactional Memory

To avoid the issues of programming with locks and to increase concurrency in application execution, researchers have proposed Transactional Memory (TM) [42, 43]. TM is easier to use than locks (as shown by Rossbach et al. [83]) because programmers do not have to be greatly involved in synchronization of shared memory accesses. They mark sections of code (similar to critical sections), called *transactions*. The underlying TM library allows transactions to be executed optimistically and concurrently. For each transaction, it tracks shared memory accesses, and either makes the updates to shared memory (if any) globally visible, or – in case of conflicting memory accesses – discards changes, and re-executes the transaction.

Since TM simplifies parallel programming, it has been an attractive area for many researchers in the last two decades, and they have proposed various implementations in software [31, 88], hardware [6, 36] or as a combination of software and hardware [21, 81]. Furthermore, TM is nowadays supported in mainstream Intel Haswell [35] and IBM POWER8 [16] processors.

1.3 Deterministic Multithreading

Even though TM makes programming of multithreaded applications simpler and less error-prone than programming with locks, some bugs are hard to avoid in general. The difficulty of finding and fixing a bug lies in the underlying (nondeterministic) nature of multithreaded applications, where application threads run in parallel and interleave in arbitrary order, i.e. nondeterministically. For a single input set, the number of possible thread interleavings grows exponentially with the number of running threads and the number of critical sections (or transactions) [104].

Nondeterminism is common and prevalent in modern multithreaded applications [71]. Sources of nondeterminism are various: process scheduling, concurrent memory accesses, caches, and microarchitecture structures [23], and they might affect debugging, testing and providing fault tolerance in applications.

Depending on thread interleavings, a bug can occur in one execution, but be hidden in another. Furthermore, nondeterministic execution of an application might produce a different output even when it starts with the same input. When multiple replicas of the same application run in parallel to provide fault tolerance, it is hard to say if different outputs are due to a faulty execution or a fault-free nondeterministic execution.

On the other hand, determinism provides repeatable execution: for the same input, threads interleave in the same order, a program behaves in the same way and gives the same output, which is important for testing. In the presence of a bug, determinism helps a programmer to reproduce, find and fix the bug. Importantly for fault-tolerant systems, with determinism any mismatch in replicas' outputs is only due to a faulty execution of one of the replicas.

Although programmers would benefit from deterministic execution of TM-based (*transactional*) applications, the systems proposed so far have limited applicability and do not provide proper support for transactional applications with data races, standard library function calls and ad hoc synchronization.

1.4 Problem Statement

This thesis addresses issues in: (i) deterministic multithreading of transactional applications, and (ii) invocation of standard library functions in transactions.

1.4.1 Issues in Deterministic Execution of Transactions

Researchers proposed deterministic multithreading to simplify developing, testing and debugging of applications. However, previous systems for deterministic multithreading are suitable for lock-based applications, and they execute transactional applications incorrectly. Programmers would greatly benefit from a deterministic system that recognizes transactions, allows their concurrent execution and guarantees determinism even in the presence of data races.

1.4.2 Issues in Invocation of Standard Library Functions in Transactions

In general, TM is not able to track accesses to shared variables and data structures in the standard library or the kernel. Therefore, a transaction that invokes a standard library function has to be serialized, i.e. it has to be executed as the only running transaction in the application. Serialization causes two main problems in application execution.

First, serialization hurts scalability and performance. To reduce the number of serialized transactions and to provide more concurrency in transactional applications researchers need a TM-aware standard library that would allow transactions to call standard library functions and to execute them concurrently.

Second, serialization might induce deadlocks. It enforces the order of threads execution that might be different from the order enforced by a deterministic system. Therefore, deterministic execution with serialized transactions might be deadlock-prone. To prevent this, a deterministic system should support serialization enforced by a TM implementation.

1.4.3 Issues in Deterministic Execution of Applications with Ad hoc Synchronization

Ad hoc synchronization is a commonly-used synchronization method although it is considered to be error-prone. Programmers usually use it as a replacement for a standard barrier. Similarly to serialization of transactions, ad hoc synchronization enforces an order of threads execution that might be different from the order enforced by a deterministic system, and might cause a deadlock. To avoid this, a deterministic system should identify ad hoc synchronization and guarantee the progress of threads execution.

1.5 Thesis Contributions

In order to address the issues described in the previous section, this thesis makes the following contributions:

- **Deterministic execution of transactional applications.** We propose DeTrans, a runtime library that ensures deterministic execution of transactional applications. It provides strong determinism, meaning that even in the presence of a data race an application produces the same output if it runs with the same input parameters. DeTrans achieves this by executing transactions in parallel and committing them in round-robin order, and by executing non-transactional code serially, again in round-robin order. It relies on a software implementation of TM to preserve memory consistency and to ensure correct parallel execution of transactional code. (*This work was published at SBAC-PAD 2014*[1].)

- **Increased concurrency in standard library functions invoked in transactions.** We present TM-dietlibc, the first TM-aware standard library. It is based on an originally lock-based library (*diet libc*) and the API remains unchanged in order to allow software developers to maintain their programming habits and to ease modification of lock-based to TM-based software. We discuss general design choices related to (i) interaction between lock-based and transactional code in a

[1] Vesna Smiljković, Srđan Stipić, Christof Fetzer, Osman S. Ünsal, Adrián Cristal, and Mateo Valero, DeTrans: Deterministic and Parallel Execution of Transactions, In Proceedings of the 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2014).

standard library, (ii) standard library adaptation to a TM implementation, and (iii) standard library execution. We explain which design choices are suitable for standard libraries, and we detail our experience with diet libc and TM integration. We propose optimizations suitable for transactional code, and also an extension of the TM conflict detection mechanism that solves the problem of detecting conflicts in kernel space, which cannot be detected automatically by TM. With TM-dietlibc, serialization of transactions is needed only for a few system calls. (*This work was published at TRANSACT 2010*[1] *and IPDPS 2013*[2].)

- **Deterministic execution in a standard library.** We propose DeTrans-lib, the first TM-aware standard library that provides deterministic execution of transactional applications. We port deterministic system DeTrans in TM-dietlibc to ensure deterministic multithreading at application and standard-library level. DeTrans-lib supports serialization of transactions and avoids deadlocks caused by busy-waiting in serialization (enforced by the TM library due to a few system calls) and busy-waiting in deterministic execution (enforced by DeTrans). With DeTrans-lib, threads invoke and execute standard library functions and system calls in deterministic order. (*This work was published at NPC 2015 and IJPP 2015*[3].)

- **Support for ad hoc synchronization while running deterministically.** We present DeTrans-adhoc, the extended DeTrans-lib implementation, that supports ad hoc synchronization in transactional applications while running deterministically. We use hardware performance counters to identify synchronization loops at runtime and to avoid deadlocks by dynamically (but deterministically) changing the order of threads execution.

[1] Nebojša Miletić, Vesna Smiljković, Cristian Perfumo, Tim Harris, Adrián Cristal, Ibrahim Hur, Osman S. Ünsal and Mateo Valero, Transactification of a Real-world System Library, In the 5th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 2010).

[2] Vesna Smiljković, Martin Nowack, Nebojša Miletić, Tim Harris, Osman S. Ünsal, Adrián Cristal, and Mateo Valero, TM-dietlibc: A TM-aware Real-world System Library, In Proceedings of 27th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2013).

[3] Vesna Smiljković, Osman S. Ünsal, Adrián Cristal, and Mateo Valero, Determinism at Standard-Library Level in TM-Based Applications, In Proceedings of the 12th Annual IFIP International Conference on Network and Parallel Computing (NPC 2015), and International Journal of Parallel Programming (IJPP 2015), special edition for NPC.

1.6 Thesis Organization

Figure 1.1 shows the architectural diagram with the contributions of this thesis.

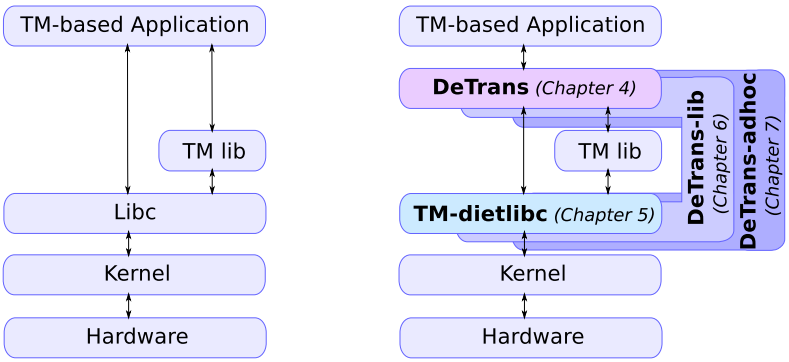


Figure 1.1: The architectural diagram without (left) and with (right) the contributions of this thesis (DeTrans, TM-dietlibc, DeTrans-lib, and DeTrans-adhoc).

The organization of this thesis is as follows:

- Chapter 2 describes the background of this thesis by providing more details about transactional memory and deterministic multithreading.
- Chapter 3 introduces the experimental setup we use in this thesis.
- Chapter 4 presents DeTrans, a runtime library that ensures deterministic execution of transactional applications even in the presence of data races.
- Chapter 5 introduces TM-dietlibc, a TM-aware standard library that increases concurrency in transactional applications when standard library functions are invoked in transactions.
- Chapter 6 presents DeTrans-lib, ported DeTrans in TM-dietlibc, that ensures deterministic multithreading at application and standard-library level, and avoids deadlocks caused by serialization required by TM.
- Chapter 7 extends DeTrans-lib to support ad hoc synchronization in an application or an external library, while running deterministically.
- Chapter 8 gives an overview of the work related to this thesis.
- Chapter 9 concludes this thesis and provides ideas for future work.

2 Background

2.1 Transactional Memory

The basic concepts of Transactional Memory (TM) [42, 43] originate from database management systems (DBMS), which successfully exploit parallel hardware by executing transactions – one or more queries that access and process data from a database – at the same time. Importantly for TM, a transaction satisfies the following properties: atomicity, consistency, isolation, and serializability. Operations in a transaction are either all performed or none of them (*atomicity*). Shared resources are always in a consistent state, whether a transaction finishes successfully or not (*consistency*). The changes made in a transaction are not globally visible until it finishes successfully (*isolation*). Updates of shared resources performed in transactions appear like they executed serially, one after another (*serializability*).

The concepts of transactions in databases and their parallel execution provide robustness and good performance, and they motivate researchers and developers to use these concepts in parallel programming.

2.1.1 How to Use Transactional Memory?

When using TM as a concurrency control mechanism, a developer writes a transactional application by identifying and marking sections of code (transactions) that a thread is executed atomically and isolated from other threads.

Figure 2.1 shows an example of transactional code. We use the keyword from the Draft C++ TM Specification [5] to mark a transaction as `__transaction_atomic {}`. The transaction performs one read from a shared variable (if `read_only` is 1), or one read and one write (if `read_only` is 0). In the former case, threads execute the transaction concurrently – they read the shared `counter` – and the transactions finish their execution. In the latter case, one or more transactions have to be aborted and re-executed due to the conflicting memory update.

The advantage of TM in comparison to the locking mechanism is that a developer does not have to keep track of locks and shared variables protected by these locks, and that application code might be executed in parallel by multiple threads. In case of mutual exclusion ensured by a lock, application code has to be serialized.

In a nutshell, TM provides the simplicity of parallel programming similar to the coarse-grained locking and the concurrency similar to the fine-grained locking.

```
int counter;   // a shared variable

int local_counter;
int read_only = rand() % 2;

__transaction_atomic
{
    if (read_only) {
        local_counter = counter;
    }
    else {
        counter  += local_counter;
    }
}
```

Figure 2.1: An example of a transaction: If flag `read_only` is set, the transaction is read-only and threads can execute it in parallel. Otherwise, one or more transactions have to be aborted and re-executed due to an update of `counter`.

2.1.2 Management of Transactions

To ensure atomicity and isolation of transactions, a compiler inserts function calls to start and commit transactions, and to instrument accesses to shared memory (Figure 2.2).

`_ITM_beginTransaction()`[1] initializes the transactional metadata and creates the snapshot of the register file and local variables. The compiler instruments all read and write operations that access shared variables, including operations in invoked functions. Instrumented reads and writes keep track of all speculative memory accesses. For each read (`_ITM_RU4()`[2]), a TM implementation saves the read address and the read value in a buffer called *read set*. For each write (`_ITM_WU4()`[3]), it saves the write address and the new value in a buffer called *write set*[4]. At the end of a transaction, the compiler inserts `_ITM_commitTransaction()` to make speculative writes globally visible, in case there was no conflict with any other transaction. In case of a conflict, the TM implementation aborts the transaction, rolls back the changes in the register file and local variables, and re-executes the transaction.

A conflict occurs when two or more threads access the same shared variable, and at least one of them is a write. A conflict is detected when TM realizes that a conflict has occurred. A conflict is resolved when TM performs an action to maintain the memory consistency, e.g. it aborts one of the conflicting transactions.

If TM tracks accesses to shared variables even outside transactions, then it provides *strong isolation* [20] (also called *strong atomicity* in Blundell et al. [13]), which is typical for TM implemented in hardware. On the other hand, if TM tracks accesses to shared variables only inside transactions, it provides *weak isolation*, which is typical for TM implemented in software. A program without data races behaves the same regardless of the isolation type.

Although we briefly explained how a compiler and a TM implementation manage transactions to provide the main properties of transactions, TM implementations differ in how they keep track of reads and writes, and how they cope with conflicting transactions.

[1] According to Intel's TM ABI [18], all functions that are used for managing transactions have fixed prefix `_ITM_`.

[2] Intel provides different ABI for read and write operations to allow optimizations in a TM implementation. RU4 stands for a Read operation of an Unsigned integer of 4 bytes.

[3] WU4 stands for a Write operation to an Unsigned integer of 4 bytes.

[4] This is typical for lazy versioning, which will be explained in detail in Section 2.1.3.

```
    .LBB2:
 read_only = rand() % 2;
        call rand
        cdq
        shr edx, 31
        add eax, edx
        and eax, 1
        sub eax, edx
        mov DWORD PTR [rbp-4], eax
__transaction_atomic
{
        mov edi, 43
        mov eax, 0
        call _ITM_beginTransaction
        and eax, 2
        test eax, eax
        je  .L2
    .L2:
if (read_only)
        cmp DWORD PTR [rbp-4], 0
        je .L6
local_counter = counter;
        mov edi, OFFSET FLAT:counter
        call _ITM_RU4
        mov DWORD PTR [rbp-8], eax
        jmp .L7
    .L6:
counter += local_counter;
        mov edi, OFFSET FLAT:counter
        call _ITM_RU4
        mov edx, eax
        mov eax, DWORD PTR [rbp-12]
        add eax, edx
        mov esi, eax
        mov edi, OFFSET FLAT:counter
        call _ITM_WU4
    .L7:
}
        call _ITM_commitTransaction
        ret
```

Figure 2.2: Assembly code as output of compiled transactional code: To manage a transaction, a compiler (GCC, version 4.7 or newer) inserts function calls to start and commit the transaction and instruments reads and writes to shared variables.

2.1.3 Version Control

TM implementations manage concurrent reads and writes from multiple transactions by keeping track of the versions of shared variables. There are two data versioning approaches: eager and lazy.

- **Eager versioning.** A transaction buffers the previous version of shared data and writes the transactional (speculative) version directly in memory. To be able to update the memory, the transaction acquires a lock to access the shared data exclusively, and the lock is released when the transaction commits or aborts. In the meanwhile, other transactions that read the same shared data have to check for conflicts, and those that write to the same shared data have to acquire the same lock. If the transaction commits, it simply discards the buffered version, and if it aborts, the buffered version is copied back to memory.

- **Lazy versioning.** A transaction buffers the transactional version of shared data. If the transaction aborts it discards the buffered version and if it commits it copies the buffered version to memory. The updates to memory are also protected by locks.

The main overhead of eager versioning is at abort time due to restoring shared data in memory, while the main overhead of lazy versioning is at commit time due to new memory updates.

2.1.4 Conflict Detection

A TM implementation tracks read and write accesses in transactions, compares them with the accesses of other transactions, i.e. validates reads and writes, and prevents one of the conflicting transactions to commit its changes. In case of a conflict, a transaction discards its transactional reads and writes and has to be re-executed. Two different approaches of conflict detection are: eager and lazy.

- **Eager conflict detection.** This conflict detection checks for a conflict at every transactional memory access by comparing transactional reads and writes of one transaction with transactional reads and writes of other running transactions. Eager

conflict detection detects conflicts as early as possible and minimizes the amount of work that is wasted in case a transaction aborts.

- **Lazy conflict detection.** This conflict detection is optimistic since it allows transactional memory accesses to be executed without checking for conflicts until the transaction finishes its execution. To check for conflicting memory accesses, the transactional reads and writes are compared with other running transactions. Lazy conflict detection avoids the overhead of detecting conflicts at each memory access, but the amount of work that is wasted in a transaction that aborts might be significant.

Conflict detection performed for every memory location would induce high overhead in validation and application execution in general. Therefore, TM performs conflict detection at the granularity of objects [44, 60], words [31, 40] or cache lines [68]. Although the coarse-grained granularity reduces the overhead in validation of transactional memory accesses, it might cause transactions to have a conflict when they access different memory locations, but the same object, word or cache line. This type of a conflict is called a *false conflict*.

2.1.5 Additional Support in TM

The basic management of transactions (including conflict detection and version control) is sufficient for a TM implementation to provide atomicity and isolation of simple transactions like in Figure 2.1. However, developers usually write more complex benchmarks where they compose transactions and allow (*outer*) transactions to enclose one or more (*inner*) transactions. Therefore, a TM implementation has to provide support for nested transactions [42, 69]. Depending on the interaction between outer and inner transactions, nesting can be: flat, closed or open.

- **Flat nesting.** This is the simplest type of nesting, where inner transactions are coalesced with the rest of the outer transaction. The commit of an inner transaction is omitted, and transactional writes are committed to memory at the commit of the outer transaction. In case of a conflict in an inner transaction, the outer transaction gets aborted and re-executed.

- **Closed nesting.** Like in flat nesting, the outer transaction is executed as one big transaction, and transactional writes of an inner transaction are committed at the commit of the outer one. However, in case of a conflict in the inner transaction, only this transaction is aborted and re-executed.

- **Open nesting.** This type provides more concurrency, because an inner transaction commits its changes to memory optimistically assuming that the outer will commit as well. If a conflict appears in the inner transaction, only this transaction gets aborted and re-executed.

Apart from nested transactions, a transactional benchmark might invoke standard library or system calls within transactions. These calls (called *real* or *unprotected* actions [8]) might make modifications that TM is not able to track, and they might cause side effects that TM cannot roll back. Therefore, TM has to provide support to defer these actions until commit time (in a *commit handler*), roll back their modifications at abort time (in an *abort handler*) or execute them when other transactions are not running (*serial execution*).

- **Commit handlers.** Some operations in a transaction should be deferred until the transaction commits its changes, and executed in a commit handler. This is very useful for single transactions. However, for nested transactions it can cause side effects that are not obvious at first glance. For instance, in flat nesting all inner transactions are combined into a single one; therefore, a deferred operation is executed when the outer transaction commits. The problem occurs when the result of the deferred operation is needed inside the outer transaction, but it still has not been performed. Therefore, developers should carefully choose operations for deferring. For example, freeing memory inside a transaction can be safely deferred until the commit time.

- **Abort handlers.** TM is not always able to discard changes made in a transaction. However, it is possible to write compensation code in an abort handler, and to execute it when the transaction aborts. For example, an abort handler is suitable for transactions that allocate memory, and the compensation code is the opposite action, i.e. freeing memory.

- **Serial execution.** Transactions with visible non-revertible side effects should be neither aborted nor re-executed. Instead, a TM implementation has to guarantee their commit. If the TM implementation cannot provide this while other transactions are running in parallel, it has to execute the transaction serially, as the only running transaction in the application. An example is a transaction that updates a file on the disk.

2.1.6 TM Implementations and Performance

The idea of supporting atomic operations in parallel programming was suggested by Lomet [55] in 1977, and the idea of parallel execution of transactions on a multiprocessor, supported by the cache coherence protocol, by Knight [52] in 1986. Finally, Herlihy and Moss proposed the first implementation of TM in hardware [43] in 1993, and Shavit and Touitou the first implementation in software [88] in 1995. Since then, researchers have provided various software, hardware, and hybrid implementations, and finally in the last few years, TM became integrated in the mainstream processors.

TM implementations in hardware (HTMs) provide the concurrency control in a modified cache and cache coherence protocol. The advantage of HTMs is that they do not require instrumentation of transactional memory accesses, but the disadvantage is that hardware transactions might exceed the cache capacity and have to be aborted, and then re-executed in a fall back path, e.g. protected by a global lock.

The first commercial processor with TM support was the Rock processor [66], developed in Sun Microsystems, but never released since the Rock project was cancelled. Fortunately for the TM community, other hardware manufacturers started supporting TM in their processors.

IBM implements HTM in BlueGene/Q [100], zEnterprise EC12 [37], and POWER8 [16] processors.

Furthermore, Intel implements HTM in Haswell processors [35, 79] as Transactional Synchronization eXtensions (TSX) to the x86 ISA. It provides two interfaces for developers to use transactions. Restricted Transactional Memory (RTM) allows developers to mark transactions that will be executed atomically. In case of a conflict, a transaction is aborted and execution takes a fall back path (e.g. the same code protected by a lock). Hardware Lock Elision (HLE) allows lock-based code to be executed optimistically

without locks, and the processor provides consistency. In case of a conflict, the memory updates are discarded and the code has to be re-executed as lock-based.

Another extension to the x86 ISA with the HTM support is Advanced Synchronization Facility (ASF) [17]. Unfortunately, AMD has never integrated it in the mainstream processors.

In general, applications with short transactions and a low conflict rate benefit from a hardware implementation of TM, and they are energy efficient and have good performance [26].

However, for applications with long and conflicting transactions software TM (STM) implementations are preferable. In STM, a compiler and a runtime library ensure atomicity and isolation of transactions. The compiler inserts function calls for managing transactions in the runtime library, and instruments reads and writes so that the runtime library keeps track of transactional memory accesses (as explained in Section 2.1.2).

Diegues et al. [26] point out that TinySTM [30, 31] is currently the best choice according to execution time and energy efficiency. In addition, TinySTM is compatible with the mainstream compilers.

The GNU GCC[1] and Intel CC[2] compilers rely on the Draft C++ TM Specification [5], and apart from the compatibility with other STM implementations, they also implement STM as an extension to the C/C++ programming language. Furthermore, IBM XL [47] provides support for HTM integrated in POWER8 processors.

Finally, hybrid TM (HyTM) implementations are proposed to take advantage of both HTM and STM. The hardware provides good performance and the software handles the transactions that cannot be successfully executed in hardware due to its limitations. In HyTM, hardware and software transactions run in parallel. Although HyTM performs well with the simulated HTM (as shown by Matveev et al. [61]), according to the study provided by Diegues et al. [26], HyTMs do not perform as well as expected when using a real HTM implementation.

[1] https://gcc.gnu.org/
[2] https://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition

2.2 Deterministic Multithreading

Even though TM makes parallel programming easier and less error-prone, concurrency bugs are still present in multithreaded applications because application threads run in parallel and interleave in arbitrary order, i.e. nondeterministically. For a single input set, the number of possible thread interleavings grows exponentially with the number of running threads and the number of critical sections (or transactions) [104]. Furthermore, this number gets larger with every shared memory access performed outside critical sections (or transactions) and with different input parameters.

Sources of nondeterminism are the state of: caches, memory pages, OS global data structures, I/O buffers, and branch predictor tables [23]. Additional sources of nondeterminism are concurrency bugs and library calls: `malloc` (dynamic memory allocation might return a different address in every execution), `gettimeofday` (the function returns current time), `sleep` (the function suspends thread execution for a specified time), and `rand` (the function returns a different value each time it is called) [71].

In this thesis, we assume that program execution is not based on time, random values, or nondeterministic memory addresses[1], and we focus on concurrency bugs.

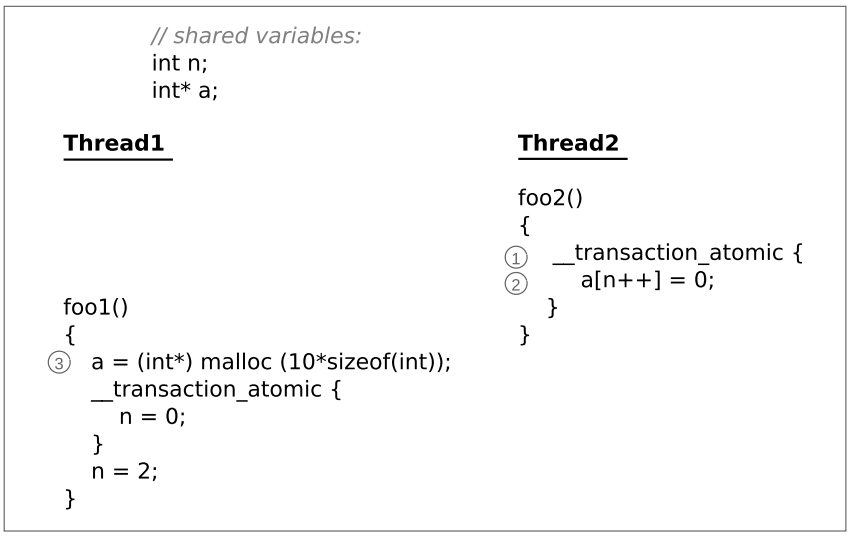
2.2.1 Why Do I Have Bugs in My Code?

Many different types of concurrency bugs can appear in multi-threaded applications, e.g. deadlocks, atomicity and order violations [58], due to the lack of proper synchronization among application threads.

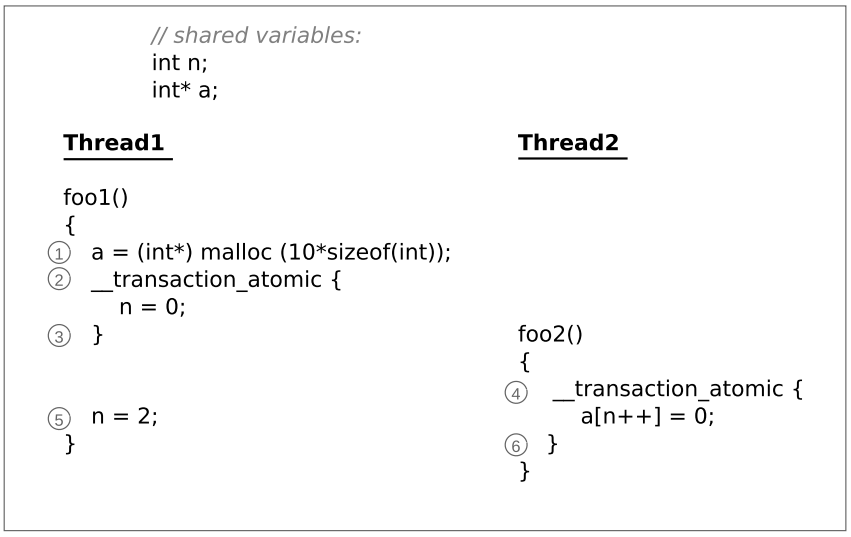
In Figure 2.3 we show an example of very simple code with two bugs (a use of an uninitialized pointer and a data race) and how they manifest in nondeterministic executions. Depending on the order of threads execution, one or none of these bugs might appear.

Threads `Thread1` and `Thread2` execute functions `foo1()` and `foo2()`, respectively, and access shared variable `n` and shared array `a`. If the threads execute the code in the order shown in Figure 2.3(a), `Thread2` accesses the array using the pointer that was declared but not initialized yet by `Thread1`, causing a segmentation fault and crashing the program.

[1] We disable the address space layout randomization.



(a) An access to an uninitialized variable causes a segmentation fault.



(b) An update of a shared variable out of transactions causes memory inconsistency.

Figure 2.3: Examples of concurrency bugs: Two threads execute `foo1()` and `foo2()` concurrently, and access shared memory. The circled numbers represent execution order of the code in the functions. The thread interleaving in (a) triggers a "use of uninitialized pointer" bug, which causes a segmentation fault and crashes the program. The thread interleaving in (b) causes a "data race" bug where the final value of the shared variable is incorrect because of the lost update `n = 2;`.

According to the order of threads execution in Figure 2.3(b) `Thread1` initializes the pointer, updates shared variable `n` to 0, and `Thread2` reads that value in the transaction, increments it and commits it to 1, without knowing that the value was changed to 2 by `Thread1` in the meanwhile (see weak isolation in Section 2.1.2). The transaction overwrites the memory update `n = 2;` (which is a data race, or a *intermediate lost update* [89]) and the final value of `n` is 1, which is incorrect.

With the other orders of threads execution, the program finishes its execution successfully and correctly.

To reduce the number of thread interleavings and to help programmers to reproduce, find and resolve concurrency bugs, researchers have proposed deterministic multithreading. A program runs deterministically if for the given input parameters, it always produces the same output in the same way [57], i.e. application threads always interleave in a single order. We define input parameters as the arguments used for running an application, and any input from the network or a user. We define output as what the program writes to the file descriptors (e.g. standard output, user terminal, sockets), and the state of shared memory at the end of the execution.

2.2.2 Benefit of Deterministic Execution

Deterministic execution provides *repeatability*, meaning that application threads interleave in a single order and the application finishes with only one possible output in every execution with the same input parameters. If the program from Figure 2.3 runs deterministically, the threads will always update the shared variable `n` in the same order, e.g. `Thread1`: `n` $\leftarrow$ `0`, `Thread2`: `n` $\leftarrow$ `1`, `Thread1`: `n` $\leftarrow$ `2` and the program will always finish execution successfully. Although the "use of uninitialized pointer" bug and the data race remain hidden in the other orders, running this example deterministically is always bug-free and with the correct output.

Repeatability reduces the number of possible thread interleavings. It helps a programmer to understand the behaviour of applications [14], and it is important for testing, debugging and providing fault tolerance.

- **Testing.** Programmers perform testing by comparing the program output with the *correct* output. In multithreaded applications, it is usually not trivial to know the correct output in the first place, and it is even harder to say if the variety in outputs

(even when the application finishes its execution successfully) is caused by a bogus execution, or a correct nondeterministic execution. The example in Figure 2.3 might give two different correct outputs: i) `Thread1: n ← 0`, `Thread2: n ← 1`, `Thread1: n ← 2`, and ii) `Thread1: n ← 0`, `Thread1: n ← 2`, `Thread2: n ← 3`. With repeatability, the time that a programmer spends in testing can be significantly reduced since there is only one possible output as long as the application is executed deterministically.

- **Debugging.** To debug an application, it is crucial that the application behaves the same way in every execution, and that it always finishes successfully, or it always exits with the same error. With deterministic execution, the order of threads execution remains the same, with or without a debugger, or even if the programmer adds function calls such as `printf`s to gather additional information about application execution. If the threads in Figure 2.3 run deterministically and in the order marked in (a), the deterministic execution will always guarantee the same order and will help the programmer in finding the bug of accessing the uninitialized variable.

- **Fault tolerance** To be able to tolerate faults, the systems based on replication [75, 85] execute the same code multiple times and if one of the replicas gets affected by a fault, the fault stays isolated and does not have influence on other replicas. The comparison of the replicas' outputs shows if any of the replicas was faulty. However, replicas have to be executed deterministically, so that any mismatch in the outputs is only due to a faulty execution.

2.2.3 Various Levels of Determinism

Systems for deterministic multithreading can be classified by the level of determinism they provide (application, standard library, or OS) and how they cope with data races (weak or strong determinism).

When a system guarantees deterministic execution of application code, we call this application-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.

Weak determinism provides deterministic execution only of critical sections and transactions. If an application is race-free, weak determinism is sufficient to have deterministic execution of the entire application. Strong determinism provides deterministic execution of the complete application code. In the presence of a data race, only strong determinism guarantees deterministic execution of an application. Although it induces higher overhead, we prefer strong over weak determinism since data races are common concurrency bugs [87] and their number increases with the increase of parallelism in software and hardware [49].

3 Experimental Setup

In this chapter, we introduce the setup used to perform experiments and evaluate the TM-aware standard library and the deterministic systems. We give a brief overview of processor specification, benchmarks, compilers, employed TM implementations, a simulator for TM implemented in hardware, and tests used to verify the correctness of implementations of the library and the deterministic systems.

3.1 Processor Specification

We ran benchmarks on the systems with the following processors.

- **Intel Xeon E5405** contains four cores serviced by private 32KB L1 Icache and 32KB Dcache, a private 64KB L2 cache, a shared 8MB L3 cache, 4GiB RAM, and with clock speed of 2.00GHz per core.
- **Intel Xeon E3-1220** contains four cores serviced by private 32KB L1 Icache and 32KB Dcache, a private 256KB L2 cache, a shared 8MB L3 cache, 16GiB RAM, and with clock speed of 3.10GHz per core.

3.2 TM Implementations

In Table 3.1 we show the TM implementations we use in this thesis and their characteristics.

TM implementations	Version control	Conflict detection	Strong isolation	Concurrent execution
Serial-TM	none	none	no	no
TinySTM-eager (STM)	eager	eager	no	yes
TinySTM-lazy (STM)	lazy	lazy	no	yes
ASF (HTM)	lazy	eager	yes	yes
RTM (HTM)	lazy	eager	yes	yes
HyLSA (HyTM)	lazy, eager	eager	yes	yes

Table 3.1: Characteristics of TM implementations.

The implementations are in software, hardware and as a combination of both.

- **Serial-TM** in an implementation that guarantees that a transaction is serialized and protected by a global lock when it cannot be executed in parallel with other transactions. Since it runs as the only transaction in an application, transactional memory accesses are not instrumented, and a serialized transaction accesses shared memory directly.

- **TinySTM** [31] is a lightweight TM software implementation, which applies a time-based algorithm derived from Lazy Snapshot Algorithm (LSA) [80]. It performs conflict detection at the word granularity and uses locks for memory updates. Except for the mutual exclusion, locks also hold the version number, which is used in version control of read and write accesses. **TinySTM-eager** refers to TinySTM with eager version control and eager conflict detection, and **TinySTM-lazy** to TinySTM with lazy version control and lazy conflict detection.

- **Advanced Synchronization Facility (ASF)** [17] is a TM implementation in hardware proposed by AMD. It ensures that operations on multiple memory objects are performed atomically inside speculative regions, and this can be used to implement more sophisticated synchronization mechanisms. ASF uses SPECULATE to start a speculative region and COMMIT to finish it. To access memory atomically

inside a transaction, MOV instructions have to be explicitly prefixed with LOCK. Memory operations without the prefix do not get undone when a speculative region is aborted.

ASF tracks data at the cache-line granularity. Conflicts are detected between transactional memory accesses, but also between transactional and non-transactional memory accesses; therefore, ASF guarantees strong isolation. Aborts happen due to conflicting memory accesses, capacity overflows, exceptions, interrupts, and unsupported instructions (e.g. system calls).

- **Restricted Transactional Memory (RTM)** [79] is one of two interfaces for developers to use transactions in Intel's Transactional Synchronization eXtensions (TSX). A transaction starts with an XBEGIN instruction and commits changes to memory in an XEND instruction. RTM tracks data at the cache-line granularity.

  We emulated RTM, which uses the facility similar to ASF. The main difference is that RTM does not require the LOCK prefix for MOV instructions and does not allow non-transactional accesses inside of transactions. Both implementations mimic the best-effort characteristics of hardware and provide a software fall back in Serial-TM for transactions that cannot finish their execution due to hardware limitations.

- **HyLSA** [81] is a library that allows hardware and software transactions to coexist (run and commit) in parallel. Transactions run in hardware first, synchronized by ASF, and in case they get aborted due to one of the limitations of HTM, they are re-executed using the STM part of the library. The STM part uses LSA to synchronize concurrent software transactions, but also to get notified by updates of concurrent hardware transactions. Version control is lazy in ASF and eager in LSA.

The benchmarks' executions, which employed HTM and HyTM, were conducted using PTLsim [105], a nearly cycle accurate CPU simulator, enhanced with ASF extension and with the configuration from Pohlack et al. [74].

3.3 Benchmarks

In this dissertation we use different types of benchmarks with the characteristics shown in Table 3.2.

Benchmarks	Special operations	Nested transactions	Ad hoc sync	Conflict rate
STAMP	none	none	none	low, medium, high
Microbenchmarks	I/O	none	none	low
TioBench	I/O	none	yes	high
RBTree	memory management	none	none	low, medium, high
Fluidanimate	none	yes	yes	low

Table 3.2: Characteristics of benchmarks.

- **Microbenchmarks** are the benchmarks we implemented to evaluate I/O standard library functions. Threads perform operations: `fgetc`, `fgets`, `fread`, `fputc`, `fputs`, and `fwrite` (although the last two have the same implementation in diet libc) inside transactions, and calculations on thread-local variables outside transactions using the output of the file operations. I/O operations are performed on a single file of 2 GiB using a shared file descriptor and its associated internal I/O structure. Microbenchmarks spend most of the time outside transactions. Transactions are short but with high contention since whenever two transactions concurrently access a file (even to read a character), a conflict occurs due to updates of the internal I/O structure performed in both transactions.

- **STAMP** [65] is a benchmark suite commonly used in the TM community. STAMP consist of eight benchmarks (Bayes, Genome, Intruder, Kmeans, Labyrinth, SSCA2, Vacation, and Yada) with different domain, algorithms, sizes of transactions and the contention level.

  - Bayes applies an algorithm for learning the structure of Bayesian networks. A network is represented as a directed acyclic graph with variables as nodes and dependences between nodes as edges. Transactions are used to synchronize a calculation and addition of a new dependency, which is time-consuming. As

a consequence, Bayes spends most of the execution time in long transactions with large read and write sets and high contention.

- Genome takes DNA segments and reconstructs the original source genome according to the segments. Threads access the segments in transactions, which are of moderate length and with the moderate size of read and write sets. Genome spends most of the execution time in transactions with low contention.
- Intruder uses a signature-based network intrusion detection algorithm to scan network packets looking for matches to intrusion signatures. Threads access a FIFO queue in transactions in the capture phase of the execution, and a self-balancing tree in transactions in the reassembly phase of the execution. Intruder has short transactions with moderate level of contention.
- Kmeans applies the K-means algorithm to group objects into proper subsets. The application uses transactions to protect the update of a subset. When threads are updating different subsets, Kmeans has short transactions with low contention.
- Labyrinth finds the shortest path between two points in a three-dimensional uniform grid that represents a maze. Threads take the start and the end points and try to connect them with the maze grid points, and they perform the path calculations in transactions. A conflict occurs when threads pick an overlapping path. Labyrinth spends almost all of the execution time in transactions, which are very long, with large read and write sets and with high contention.
- SSCA2 creates an efficient graph data structure. Multiple threads add nodes to the graph, which is performed in transactions. Since this operation is not time-consuming, SSCA2 does not spend a lot of execution time in transactions. The transactions are short, with the small sizes of read and write sets and with low contention.
- Vacation simulates a travel reservation system implemented as a set of tree data structures that keep track of customers and their reservations. Multiple client threads interact with the in-memory database to make reservations, cancellations, and updates, and every client session is performed in a transac-

tion. As a consequence, Vacation spends a lot of time in transactions and its transactions are of medium length with moderate read and write set sizes.

  - Yada implements an algorithm for mesh refinement. It uses a graph data structure to store all mesh triangles. Threads access a work queue in transactions to remove a triangle, perform retriangulation and add new triangles to the work queue. Yada spends almost all the execution time in transactions.

- **TioBench** [95] is a benchmark for measuring the performance of I/O operations. Worker threads invoke system calls `lseek`, `read`, and `write` to: (i) write data to a file starting from the beginning (*seq write*), (ii) write data to a random position of a file (*rnd write*), (iii) read data from a file starting from the beginning (*seq read*), and (iv) read data from a random position of a file (*rnd read*). Random positions in the file are repeatable in every execution since we use the same seed to generate the random values.

  We modified the original TioBench implementation in order to use it for evaluation of TM-dietlibc and our deterministic systems. First, to invoke I/O functions from TM-dietlibc, we replaced system calls with their corresponding standard library functions: `fseek`, `fread`, and `fwrite`. Second, to have concurrent execution of the I/O functions, worker threads access a shared file (using a shared file descriptor) instead of a per-thread file. Third, to evaluate our deterministic system that does not support ad hoc synchronization (DeTrans-lib), we replaced a variable and a loop used for ad hoc synchronization with a synchronization provided by a `pthread` barrier.

  We ran the benchmark with the file size of 8 MB, large transactions, and high contention.

- **Red-black tree** is a commonly used benchmark of inserting, searching and removing elements from a shared balanced tree structure. The original implementation [32] is lock-based; however, it allows reads to be performed in parallel and requires writes to be serialized. The benchmark allocates memory by invoking standard library function `malloc` (when adding an element) and releases memory by invoking standard library function `free` (when removing an element). We modified the benchmark to use transactions instead of locks and to call the memory manage-

ment functions from transactional code. We run the benchmark with the default proportion of insert and removal operations and a data set of 2^{16} elements.

- **Fluidanimate** is a benchmark from the PARSEC [12] benchmark suite. It uses an extension of the Smoothed Particle Hydrodynamics method to simulate the physical interactions between fluid particles in a bounded space. The collisions of particles are handled by adding forces in order to change the direction of movement of the involved particles. The force and mass are then used to compute the acceleration and the new velocity.

  The original implementation is lock-based, but we use the TM-based version with the conditional variables suitable for transactions [29]. The benchmark also has ad hoc synchronization and nested transactions. It has linear scalability and transactions with low contention that access a few shared variables.

3.4 Compilers

The benchmarks detailed in the previous sections are compiled using the optimization level 3 (-O3) and with one the following compilers with TM support that rely on the Draft C++ TM Specification [5].

- **The GNU Compiler Collection (GCC)**[1] supports TM since its version 4.7. It recognizes transactional boundaries (`__transaction_atomic {}`) and instruments transactional accesses to shared memory. Apart from the code path with instrumented memory accesses, the compiler also generates a path with no instrumentation, which is used by Serial-TM.

- **Dresden TM Compiler (DTMC)** [17] is based on GCC and LLVM[2] and it also provides several code paths. The path with non-instrumented memory accesses is used by Serial-TM. Code paths with instrumented memory accesses can invoke different ABI calls. This allows HyLSA to choose one code path to execute hardware transactions, and the other path to execute software transactions. The code path is selected at the beginning of a transaction.

[1] https://gcc.gnu.org/
[2] http://www.llvm.org/

3.5 Verification of Correctness

To verify the correctness of the implementation of TM-dietlibc, apart from the microbenchmarks, TioBench, and Red-black tree, we implemented additional benchmarks, linked them against either glibc[1] or TM-dietlibc, and executed them with various input parameters (e.g. with various sizes of transactions). Execution of a benchmark invoking TM-dietlibc gave the same output as execution of the same benchmark with the same input parameters invoking glibc, which verified the correctness of our modifications and TM-dietlibc in general.

To verify the correctness of our deterministic systems (DeTrans, DeTrans-lib, and DeTrans-adhoc), we use Racey [45, 103], the stress test that was used for verification of previously proposed systems for deterministic multithreading [9, 10, 19, 23, 72].

The test calculates a signature from the values in a shared array that threads update concurrently. The value of the signature depends on the order of threads accessing the shared array. One access includes: reading an element of the array, calculating a new value and an index of the new element that gets updated with this new value, and updating the element. If the test is executed nondeterministically, threads update the array in arbitrary order, and the test might give a different signature in every execution. The probability that nondeterministic execution of the tests gives the same signature twice is very low. If the test is executed deterministically, the signature remains the same in every execution.

In the original implementation, accesses to the shared array are not protected by any concurrency control mechanism. We modified the test so that threads access the shared array from transactions, which is needed for verification of DeTrans. But we also kept non-transactional accesses to the array to verify that DeTrans guarantees strong determinism.

To verify DeTrans-lib, we modified the test to invoke standard library calls inside transactions. While accessing the shared array, threads also access a shared file to read it and update it with new calculated values. The signature is then calculated according to the both elements in the array and the data in the file.

Finally, to verify DeTrans-adhoc, we added several ad hoc synchronization variables and loops, and they were all identified by DeTrans-adhoc, and the test always finished

[1] http://gnu.org/software/libc/

its execution successfully.

With all the modifications described above, we kept the signature highly dependable on the threads interleaving. We executed the test with each deterministic system 1000 times, and it gave the same signature for each thread configuration. Therefore, DeTrans, DeTrans-lib, and DeTrans-adhoc passed verification.

4

DeTrans: Deterministic and Parallel Execution of Transactions

4.1 Introduction

Transactional Memory (TM) simplifies the development of multithreaded applications and avoids many concurrency bugs that might appear when using the traditional locking mechanism (deadlocks, live locks, multiple releases of locks). However, concurrency bugs are hard to avoid in general. The difficulty of finding and fixing a bug lies in the underlying (nondeterministic) nature of multithreaded applications, where threads run in parallel and interleave nondeterministically. Depending on the thread interleaving, an application's output can vary, or a bug can occur in one execution, but be hidden in another.

Previous systems for deterministic multithreading [9, 11, 23, 24, 54, 56] focus on lock-based applications and provide deterministic order of lock acquisitions. However, these systems are not appropriate for transactional applications. Since they do not recognize transactions, they might violate the main properties of TM (atomicity, consistency, and

isolation of transactions), and as a consequence, they might execute transactional applications incorrectly.

In this chapter we present *DeTrans*, a runtime library that ensures deterministic execution of a multithreaded transactional application even in the presence of data races (see strong determinism in Section 2.2.3). DeTrans achieves this by executing non-transactional part of the application serially in round-robin order, and transactions in parallel. It relies on an STM library to ensure correct parallel execution of transactional code with low additional overhead. It is lightweight because it does not use memory protection hardware or facilities of the underlying operating system to execute multithreaded applications deterministically. DeTrans is STM agnostic and works with eager and lazy implementations of STM libraries. Our modifications remain within a shared runtime library and do not require modifications of the operating system, system libraries, or benchmarks.

The rest of this chapter is organized as follows. In Section 4.2, we give the background of one of the prior systems for deterministic multithreading (Dthreads [54]). In Section 4.3, we use Dthreads to illustrate the behaviour of a transactional application running deterministically, and explain our motivation to implement a deterministic system for transactional applications from scratch. In Section 4.4 we explain the design and implementation of *DeTrans*. DeTrans is based on a double-barrier technique to separate execution of transactional and non-transactional code. Non-transactional code is executed serially in round-robin order, and transactions in parallel, with commits in the same round-robin order.

In Section 4.5 we verify the correctness of the DeTrans implementation by using the *Racey* stress test [45, 103], and we evaluate DeTrans with the STAMP benchmark suite [65]. According to our results, DeTrans is 3.99x, 3.39x, 2.44x faster on average than Dthreads (when running lock-based STAMP) for 2, 4, and 8 threads, respectively.

Additionally, we discuss different orders of threads execution based on the number of transactional memory accesses in Section 4.6, and we conclude this chapter in Section 4.7.

The ideas discussed in this chapter were published at SBAC-PAD 2014[1].

[1]Vesna Smiljković, Srđan Stipić, Christof Fetzer, Osman S. Ünsal, Adrián Cristal, and Mateo Valero, DeTrans: Deterministic and Parallel Execution of Transactions, In Proceedings of the 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2014)

4.2 Background

Deterministic multithreading ensures repeatable execution, which greatly helps programmers to test and debug multithreaded programs, and to understand their behaviour in general [14]. However, prior systems for deterministic multithreading [9, 11, 23, 24, 54, 56] are limited in their applicability since they ensure correct and deterministic execution only of the programs that use locks as the synchronization mechanism.

In this section, we give the background of one of the prior systems for deterministic multithreading – Dthreads, which is a runtime system widely used in the research community.

Dthreads is a replacement for a pthread library, and relies on process isolation and virtual memory protection to isolate concurrent memory updates, and to perform them deterministically.

In Dthreads, threads are replaced with processes, which access shared memory pages only for reading shared variables (located either in global memory or on the heap) until the first update. When a process tries to update a shared variable, copy-on-write memory protection provides a private copy of the shared memory page that the process intents to modify. From the first update, all other writes and reads are performed on the private copy of the shared memory page.

Execution of programs running deterministically with Dthreads is divided into parallel and serial phases. While processes access their private pages in parallel, updating the shared pages is done serially. At a synchronization point (a `pthread` function call), Dthreads updates shared memory with the modifications from the private copies serially in round-robin order. The shared pages are updated only by modified bytes, which can be a cause of runtime overhead in case of numerous updates.

Since Dthreads maintains shared and private memory pages, it increases a memory footprint of a running application. The memory footprint depends on the number of modified pages and the number of processes. Furthermore, applications with intensive synchronization among threads might have a high runtime overhead due to frequent creation of private pages and modification of shared pages in Dthreads.

The implementation of Dthreads is based on deterministic token passing (only the thread that has the token can update the shared memory), and the double-barrier technique where two barriers separate parallel and serial phases of program execution.

4.3 Motivation

In this section, we show an example of transactional code and its behaviour when running deterministically with Dthreads.

Figure 4.1 shows the code where two threads concurrently increment a shared counter inside a transaction. If we run the code nondeterministically, the final value of the counter is 2 (independently of the commit order of transactions). If we run it with Dthreads, at `pthread_create` threads make a private copy of the counter (the counter is 0), and then they increment only the private copy. Both transaction update their private copies to the value 1. At `pthread_join`, Dthreads updates the shared variable, first with the update from `Thread1`, and then from `Thread2`. The final value is 1, which is incorrect. This simple example shows that Dthreads is not TM-aware and might execute transactional applications incorrectly.

```
int counter;    // a shared variable

main                                  Thread1                    Thread2

main()                                increment()                increment()
{                                     {                          {
  counter = 0;                          __transaction_atomic {     __transaction_atomic {
  pthread_create(Thread1, increment());   counter++;                 counter++;
  pthread_create(Thread2, increment());   }                        }
  pthread_join(Thread1);              }                          }
  pthread_join(Thread2);
}
```

Figure 4.1: An example of two threads executing transactions and updating a shared counter.

In general, Dthreads is a widely-used deterministic system among researchers, and it is efficient when critical sections are short and threads perform a lot of work in parallel. However, even with the support for transactions, it would not be a good fit for the STAMP benchmarks because of high overhead (see the previous section for general limitations and causes of runtime overhead).

To show the runtime overhead in Dthreads, we ran lock-based STAMP benchmarks nondeterministically and deterministically with Dthreads[1], and we present the results in Figure 4.2. The maximum overhead of deterministic execution is for SSCA2. The benchmark runs 56.2x slower with Dthreads for 8 running threads in comparison to single-

[1]The evaluation environment and the input parameters are the same as in Section 4.5.

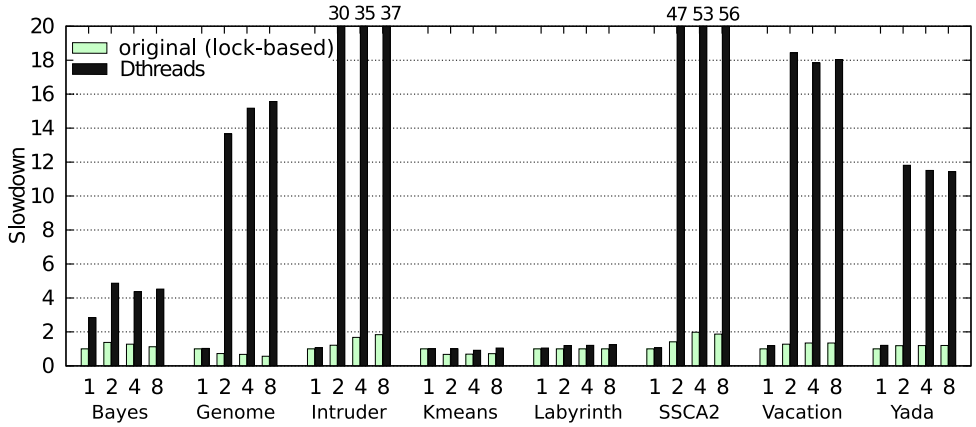


Figure 4.2: The STAMP benchmarks running with Dthreads.

threaded nondeterministic execution of the benchmarks. In addition, all the benchmarks have high memory footprint because of maintaining metadata and copies of memory pages (one copy per thread). As a consequence, we were able to run STAMP benchmarks only with a small input set.

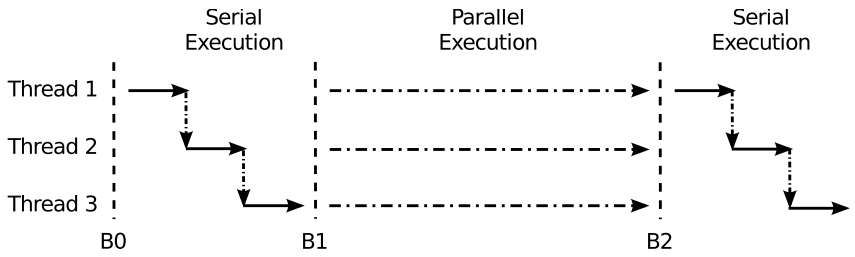
Because of the high overhead in execution time and memory, we choose to implement a deterministic system for transactional applications from scratch, rather than extending an existing one by adding support for transactions.

4.4 Implementation

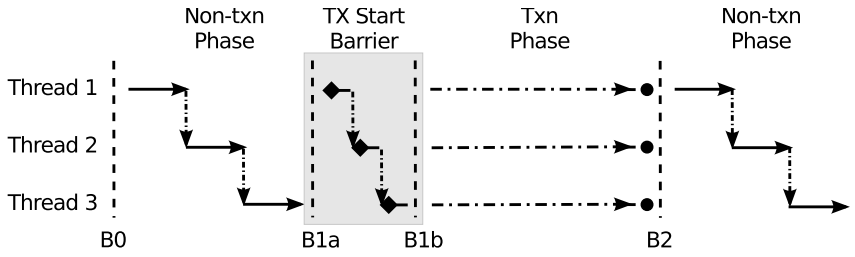
In this chapter we propose DeTrans, which implements a double-barrier technique (similarly to Dthreads, see Figure 4.3(a)) to ensure deterministic execution of transactional applications. DeTrans implements the total store ordering consistency model by using two barriers to separate transactional and non-transactional code of an application, and to execute non-transactional code serially in round-robin order and transactional code in parallel.

In order to avoid modifications of the STM library and applications, DeTrans is implemented as a library with wrappers for the functions from the standard library (`pthread*`) and from the STM library (`_ITM_*`). We preload and execute the wrappers (instead of the original functions) at runtime.

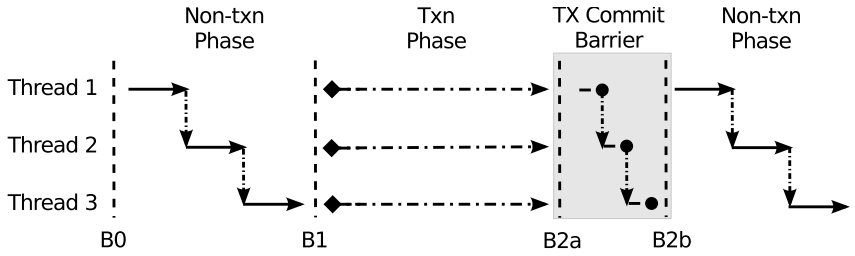
When a thread is created, DeTrans places the thread id in the queue of threads ready for execution (`ready_queue`). This queue defines the order of threads execution in the



(a) The double-barrier technique



(b) Deterministic execution with eager STM



(c) Deterministic execution with lazy STM

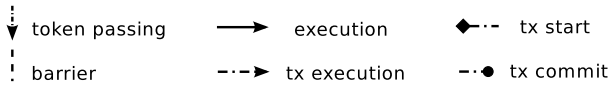


Figure 4.3: Deterministic execution with DeTrans: (a) DeTrans implements the double-barrier technique and executes non-transactional code of a program in round-robin order (Non-txn Phase) and transactional code in parallel (Txn Phase). (b) DeTrans-eager starts transactions in deterministic order (ensured by TX Start Barrier) and eager STM commits them in the same order. (c) DeTrans-lazy commits transactions in deterministic order (ensured by TX Commit Barrier).

non-transactional code of an application. When a thread finishes its execution, it is removed from the queue, or if it comes to a transactions, the thread id is moved from `ready_queue` to `txn_queue`, which then defines the order of threads committing their transactions. When a transaction commits, the thread id is moved back from `txn_queue` ro `ready_queue`.

4.4.1 Serial Deterministic Execution of Non-transactional Code

DeTrans executes non-transactional code in an application deterministically in round-robin order (Figure 4.3(a) – *Non-txn Phase*)[1]. DeTrans implements token passing where each thread is blocked until it acquires a global execution token and then it runs until it reaches a synchronization point (e.g. a transaction) or the end of execution. Only one thread from `ready_queue` is executed at a time. Non-txn Phase ends when all running threads reach the B1 barrier, i.e. when `ready_queue` is empty.

The B1 barrier is important for strong determinism – it guarantees that execution of non-transactional code is finished before execution of transactional code starts. This way, we do not allow mixing executions of non-transactional and transactional code, and we avoid nondeterministic concurrency bugs, such as the data race from Figure 2.3(b).

If a thread does not execute any transaction in a program, it runs sequentially until it exits and passes the execution token to the next available thread. Other threads blocked on the barrier (if any) do not wait for the exited thread.

4.4.2 Parallel Deterministic Execution of Transactional Code

To provide more parallelism, DeTrans executes transactions in parallel (Figure 4.3(a) – *Txn Phase*) and commits them in round-robin order. Txn Phase of a program starts when all threads start transactions (when `ready_queue` is empty and `txn_queue` is not empty), and ends when they finish the transactions and reach the B2 barrier (when `ready_queue` is not empty and `ready_queue` is empty). Similarly to the B1 barrier, the B2 barrier is important for strong determinism – it guarantees that execution of transactional code is

[1] Another approach would be that threads execute non-transactional code in parallel. To ensure strong determinism, the non-transactional code would have to be instrumented to track memory updates and to perform them in deterministic order. We prefer having serial execution with no instrumentation overhead.

finished before execution of nontransactional code starts.

DeTrans supports both eager[1] and lazy[2] implementations of STM. Eager STM performs conflict detection and memory updates during transaction execution, and lazy STM performs them at commit time. DeTrans relies on the commit policy called *commit-in-order* in TinySTM [31], which guarantees FIFO commit order (the first transaction that starts commits first). This commit policy is not sufficient to provide deterministic execution since in nondeterministic execution of a transactional application, transactions start in arbitrary order and as a consequence, commit in the same (nondeterministic) order.

Lazy STM does not provide commit-in-order and transactions commit in arbitrary order. In the following sections, we explain how to provide deterministic execution for eager and lazy STM.

4.4.3 Deterministic Eager STM Policy

DeTrans executing with eager STM (*DeTrans-eager*) introduces a TX Start Barrier (Figure 4.3(b)) that guarantees deterministic execution. After the execution of non-transactional part of a program (Non-txn Phase), all threads get blocked on the TX Start Barrier. The TX Start Barrier corresponds to the B1 barrier from Figure 4.3(a), and it consists of the B1a barrier, in-order transaction start, and the B1b barrier. After B1a, DeTrans sequentially starts transactions, and the threads get blocked on the B1b barrier. Finally, DeTrans lets all the threads execute transactions in parallel until they commit and reach the B2 barrier. After the Txn Phase, DeTrans starts executing the next Non-txn Phase of the program.

The Txn Phase in DeTrans-eager guarantees deterministic program execution because TX Start Barrier starts transactions in order and eager STM commits transactions in order (the commit-in-order policy).

In case a STM library does not have the commit-in-order feature, the conflict resolution of the STM library should be modified to ensure that a transaction that will commit earlier wins a conflicting situation (similarly to the conflict resolution proposed by Brito et al. [15]).

[1] eager versioning and eager conflict detection
[2] lazy versioning and lazy conflict detection

4.4.4 Deterministic Lazy STM Policy

DeTrans executing with lazy STM (*DeTrans-lazy*) introduces a TX Commit Barrier (Figure 4.3(c)) that guarantees deterministic execution. After the execution of non-transactional part of a program (Non-txn Phase), all threads get blocked on the B1 barrier. DeTrans allows threads to execute transactions in parallel until they reach the TX Commit Barrier. The TX Commit Barrier corresponds to the B2 barrier from Figure 4.3(a), and it consists of the B2a barrier, in-order transaction commit, and the B2b barrier. After the barrier B2a, DeTrans sequentially commits all transactions. When all transactions commit, the threads get blocked on the B2b barrier, which is the start of the next Non-txn Phase of the program.

Txn Phase of DeTrans-lazy guarantees deterministic program execution because the TX Commit Barrier commits transactions in order.

DeTrans (DeTrans-eager or DeTrans-lazy) guarantees the execution order of transactions that corresponds to the execution order as if the transactions were executed sequentially one after another. Furthermore, the execution order of transactions is independent of aborts. DeTrans ensures that the transaction that holds the deterministic token commits. Any other transaction that has a conflict with the token holder has to be aborted. The transaction then gets re-executed in the same Txn Phase and it is not skipped in the round-robin token passing.

DeTrans separates non-transactional and transactional code and executes them deterministically, and as a result, it provides deterministic execution even in the presence of data races, i.e. strong determinism.

4.5 Evaluation

We evaluate DeTrans with the benchmarks from the STAMP benchmark suite [65], using 2 Intel Xeon E5405 processors (8 cores in total). We compiled the benchmarks with GCC version 4.7 and linked them against TinySTM [31], version 1.0.5. We also verified the correctness of the DeTrans implementation by using the Racey stress test [45, 103]. More details about the experimental setup are given in Chapter 3.

4.5.1 Methodology

We compare DeTrans with the state-of-the-art deterministic system – Dthreads. Since TinySTM does not use `pthread` synchronization primitives (synchronization points for Dthreads), to run transactional applications with Dthreads we replaced transactions with critical sections protected by `pthread_mutex_lock` and `pthread_mutex_unlock` calls.

To measure performance we ran the STAMP benchmarks 10 times using the input parameters from Table 4.1, and calculated the arithmetic mean execution time. Note that the input parameters are different than the default parameters proposed by the STAMP authors, because of Dthreads running out of memory.

We used the Perf profiling tool [1] to analyse overheads of deterministic execution.

4.5.2 Results

Figure 4.4 presents the performance of the STAMP benchmarks running: (i) nondeterministically (original execution running with TinySTM), (ii) deterministically with Dthreads, and (iii) deterministically with DeTrans. The figure shows the slowdown of deterministic executions for 1, 2, 4, and 8 threads in comparison to single-threaded original execution of a benchmark. Evaluation of DeTrans based on eager STM (*TinySTM-eager*) is shown in Figure 4.4(a) (*DeTrans-eager*), and evaluation of DeTrans based on lazy STM (*TinySTM-lazy*) is shown in Figure 4.4(b) (*DeTrans-lazy*). We compare DeTrans-eager and DeTrans-lazy to the original execution of benchmarks running with TinySTM-eager and TinySTM-lazy, respectively.

Figure 4.4 shows that DeTrans-eager and DeTrans-lazy perform similarly (the maximum performance difference is 13.57% for Yada running with 8 threads). On average, DeTrans is 1.43x slower than Dthreads for 1 thread, and is 3.99x, 3.39x, 2.44x faster than Dthreads for 2, 4, and 8 threads, respectively. DeTrans performs better than Dthreads in all benchmarks except Kmeans. Kmeans has infrequent transactions - it spends most of the execution time outside of transactions, and that is the part of the program that Dthreads executes in parallel and DeTrans serially.

Figure 4.5 shows the execution breakdown for the STAMP benchmarks. The dominant overheads are: in the kernel when using locks and running nondeterministically (Figure 4.5(a)), in the Dthread implementation when running deterministically with Dthreads (Figure 4.5(b)), in the STM implementation when using transactions and running non-

benchmark	input parameters
Bayes	-v32 -r2048 -n10 -p40 -i2 -e8 -s1
Genome	-g16384 -s64 -n262144
Intruder	-a10 -l64 -n32768 -s1
Kmeans	-m40 -n40 -t0.00001 -i inputs/random-n65536-d32-c16.txt
Labyrinth	-i inputs/random-x512-y512-z7-n512.txt
SSCA2	-s15 -i1.0 -u1.0 -l3 -p3
Vacation	-n2 -q90 -u98 -r131072 -t262144
Yada	-a15 -i inputs/ttimeu10000.2

Table 4.1: STAMP benchmark input parameters.

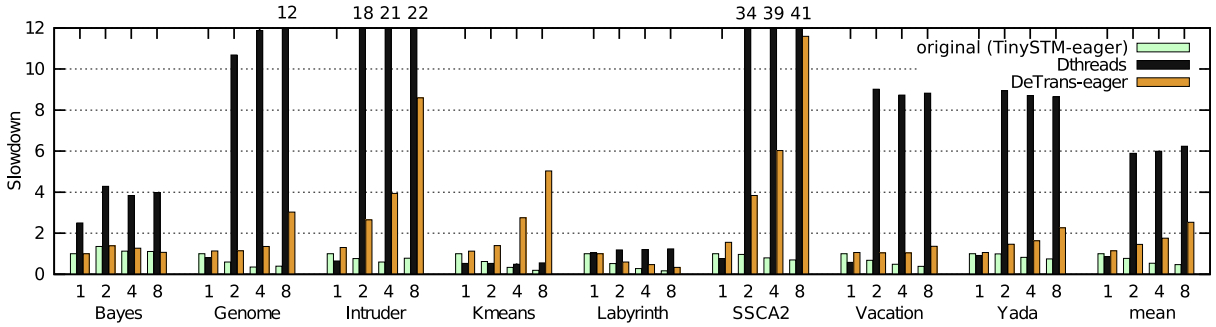
deterministically (Figure 4.5(c)(e)), and in the DeTrans implementation when running deterministically with DeTrans (Figure 4.5(d)(f)).

In Table 4.2 we show the more detailed execution breakdown of Vacation, which is a benchmarks with the medium size of transactions, running nondeterministically (with Lock, TinySTM-eager, and TinySTM-lazy) and deterministically (with Dthreads, DeTrans-eager, and DeTrans-lazy).

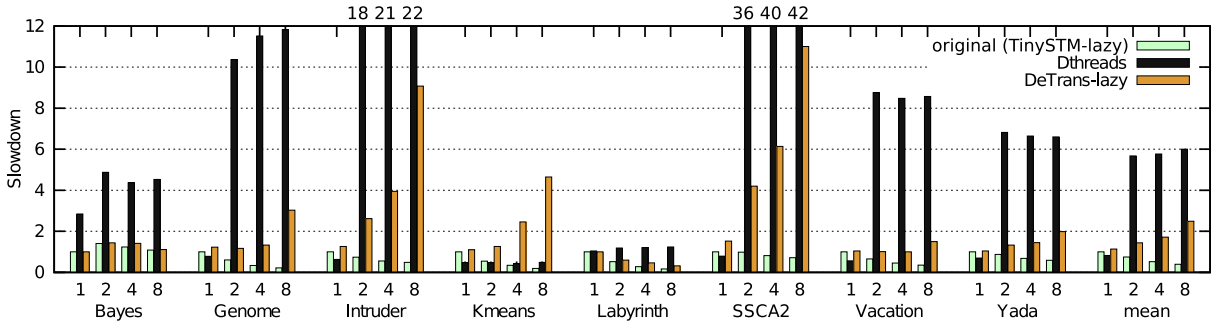
Execution with 1 thread: Dthreads, TinySTM-eager, TinySTM-lazy, DeTrans-eager and DeTrans-lazy introduce 1.26x, 2.05x, 2.09x, 2.14x and 2.19x slowdown respectively, compared to the Lock implementation while running with 1 thread. The slowdown in Dthreads is due the increased time spent in kernel execution (*kernel* - 12.9%). The slowdown in TinySTM-eager and TinySTM-lazy is due to overheads of the STM library (*libstm* - 57.2% for eager and 59.5% for lazy). DeTrans-eager and DeTrans-lazy have an additional slowdown (on top of the STM library) due to barrier and implementation overheads (*barr+imp* - 3.2% for eager and 2.9% for lazy).

Execution with 2 threads: Dthreads running with 2 threads are 19.6x slower than Dthreads running with 1 thread. This huge slowdown is caused by the use of memory protection to provide deterministic execution and by the implementation overheads[1]. When running with 1 thread, the kernel (*kernel*) and bookkeeping (*barr+imp*) times

[1]Dthreads creates a new process for each running thread, and uses the facilities of the underlying OS to protect accesses to shared memory.



(a) Deterministic execution provided by DeTrans-eager and Dthreads compared with original execution.



(b) Deterministic execution provided by DeTrans-lazy and Dthreads compared with original execution.

Figure 4.4: The STAMP benchmarks running nondeterministically and deterministically with DeTrans and Dthreads.

are 0.25s and 0.07s, respectively, and when running with 2 threads 19.27s and 17.06s, respectively. Dthreads' implementation consumes 94.6% (*barr+imp* + *kernel* time) of the execution time while running with 2 threads. DeTrans-eager and DeTrans-lazy are 1.43x and 1.54x slower than TinySTM-eager and TinySTM-lazy running with 2 threads. This slowdown is much lower than the slowdown of Dthreads. Even with all the overheads introduced when running with 2 threads, DeTrans-eager and DeTrans-lazy are 11.74x and 11.78x faster respectively, than Dthreads.

Execution with more threads: Vacation's execution time running with Dthreads stays relatively constant for 2 and more threads (38.4s, 38.28s, 36.23s for 2, 4, and 8 threads). On the other hand, Vacation's execution time running with DeTrans increases mostly when the number of threads is increased (3.27s, 3.27s, 4.26s for eager, and 3.26s, 3.23s, 4.86s for lazy with 2, 4, and 8 threads, respectively). Even though Vacation's execution time running with DeTrans increases with the increased number of threads, DeTrans

provides a speedup of 11.74x, 11.71x, and 8.5x (eager) and 11.78x, 11.85x, and 7.45x (lazy) compared to Dthreads for 2, 4, and 8 threads, respectively.

4.6 Various Orders of Threads Execution – Discussion

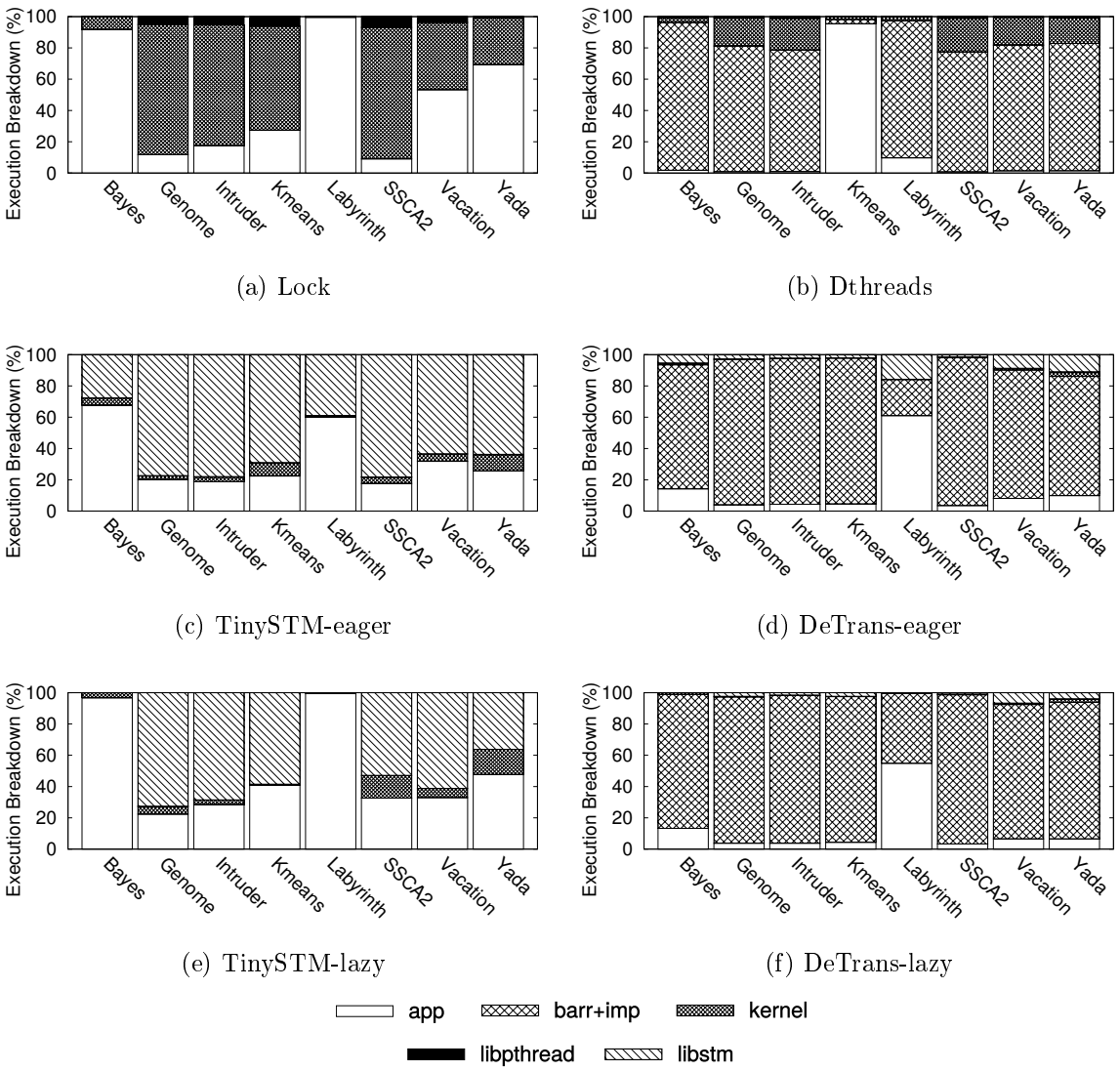
In the previous section, we showed that the overhead in deterministic execution provided by DeTrans is partially a result of the time that threads spend waiting on barriers for the deterministic execution token. The overhead would be reduced if an application had: (i) very short non-transactional parts, which shortens the Non-txn Phase, and (ii) non-conflicting transactions that finish their execution at the same time, which shortens the Txn Phase.

Since the STAMP benchmarks do not fulfil these requirements, we analyse orders of threads execution different from the traditional round-robin to reduce the time transactions spend on waiting to commit their changes in the Txn Phase.

We use three characteristics to describe transactions and define the order of transactions committing their changes: (i) transactional memory accesses – only writes (*ws*), only reads (*rs*), or both writes and reads (*wsrs*), (ii) the extrema – the minimal number (*min*) or the maximal number (*max*) of transactional memory accesses, and (iii) the range – transactional memory accesses of the transaction committed in the previous round (*prev*) or the transactions committed in all previous rounds of the same thread (*all*)[1]. Combining the characteristics in (i), (ii), and (iii) gives us 12 different orders. Note that we statically choose one of the orders to apply it in the benchmark's execution, rather than choosing different orders dynamically.

Our first assumption is that some benchmarks might benefit from the order that guarantees that a thread with the shortest transaction in the previous round also has a short transaction in the current round, and that it should commit first and release the deterministic token fast (the order *wsrs-min-prev*). Our second assumption is that some benchmarks might benefit from the order that guarantees that a thread with the longest transaction in the previous round has also a long transaction in the current round, and

[1] We cannot deterministically speculate about the transactional memory accesses in the current round, so we do it for the previous round or rounds.



(a) Lock

(b) Dthreads

(c) TinySTM-eager

(d) DeTrans-eager

(e) TinySTM-lazy

(f) DeTrans-lazy

Figure 4.5: Breakdown of benchmarks' execution for 8 running threads.

Imp. type	1 thread		2 threads		4 threads		8 threads	
Lock	sec	%	sec	%	sec	%	sec	%
app	1.45	93.8	1.29	66.3	1.15	53.7	1.13	53.3
barr+imp	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.0
kernel	0.09	5.8	0.60	30.9	0.91	42.5	0.91	43.0
libpthread	0.01	0.5	0.05	2.8	0.08	3.8	0.08	3.7
libstm	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.0
total	1.55	100.0	1.94	100.0	2.14	100.0	2.12	100.0
Dthreads	sec	%	sec	%	sec	%	sec	%
app	1.53	78.0	1.71	4.5	0.95	2.5	0.51	1.4
barr+imp	0.07	3.4	17.06	44.4	26.17	68.4	29.14	80.4
kernel	0.25	12.9	19.27	50.2	10.91	28.5	6.44	17.8
libpthread	0.11	5.7	0.35	0.9	0.25	0.7	0.15	0.4
libstm	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.0
total	1.96	100.0	38.40	100.0	38.28	100.0	36.23	100.0
TinySTM-eager	sec	%	sec	%	sec	%	sec	%
app	1.26	39.6	0.84	36.9	0.56	33.9	0.41	31.8
barr+imp	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.0
kernel	0.10	3.2	0.07	3.2	0.06	3.4	0.06	4.7
libpthread	0.00	0.0	0.00	0.0	0.00	0.1	0.00	0.0
libstm	1.82	57.2	1.37	59.9	1.03	62.7	0.82	63.5
total	3.18	100.0	2.28	100.0	1.64	100.0	1.29	100.0
DeTrans-eager	sec	%	sec	%	sec	%	sec	%
app	1.26	37.9	1.05	32.2	0.62	18.9	0.34	8.1
barr+imp	0.10	3.2	0.95	29.2	1.86	57.0	3.50	82.1
kernel	0.10	3.2	0.07	2.2	0.04	1.2	0.04	1.0
libpthread	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.0
libstm	1.85	55.8	1.19	36.4	0.75	22.9	0.37	8.8
total	3.32	100.0	3.27	100.0	3.27	100.0	4.26	100.0
STM-lazy	sec	%	sec	%	sec	%	sec	%
app	1.23	38.0	0.76	35.9	0.51	34.8	0.38	32.9
barr+imp	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.0
kernel	0.08	2.6	0.09	4.0	0.06	4.0	0.07	5.9
libpthread	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.0
libstm	1.93	59.5	1.28	60.1	0.90	61.2	0.70	61.2
total	3.24	100.0	2.12	100.0	1.47	100.0	1.14	100.0
DeTrans-lazy	sec	%	sec	%	sec	%	sec	%
app	1.31	38.8	1.04	31.8	0.57	17.6	0.32	6.5
barr+imp	0.10	2.9	1.00	30.6	1.95	60.5	4.18	85.9
kernel	0.08	2.5	0.06	1.9	0.07	2.1	0.04	0.8
libpthread	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.0
libstm	1.89	55.9	1.16	35.7	0.64	19.9	0.33	6.7
total	3.39	100.0	3.26	100.0	3.23	100.0	4.86	100.0

Table 4.2: **Vacation** execution breakdown for 1, 2, 4, and 8 threads for each benchmark implementation type (Lock, Dthreads, TinySTM-eager, DeTrans-eager, TinySTM-lazy, DeTrans-lazy). The breakdown shows the time spent in: application (*app*), barrier and implementation (*barr+imp*), linux kernel (*kernel*), pthread library (*libpthread*), and STM library (*libstm*). The application time is benchmark's total execution time excluding barr+imp, kernel, libpthread, and libstm times. The barr+imp time is the time spent in barrier waiting, synchronization, and implementation dependent bookkeeping.

that this transaction should commit first before it gets aborted by another transaction and waste a significant amount of work (the order *wsrs-max-prev*).

However, the STAMP benchmarks do not show performance improvement when applying the proposed orders. In general, even when there is some speedup (e.g. in Kmeans 19% for 8 threads with the order *wsrs-max-all*), there is also slowdown for the same order, just with a different number of threads (Kmeans 17% slowdown for 2 threads with the order *wsrs-max-all*). Since we cannot benefit from the proposed orders, we do not show any detailed evaluation.

4.7 Summary

In this chapter we presented DeTrans, a runtime library for deterministic execution of transactional applications. We explained how DeTrans ensures deterministic execution of transactional applications even in the presence of data races. DeTrans provides deterministic parallel execution of transactions using a STM library with low additional overhead. We implemented DeTrans to work with both eager and lazy implementations of the STM library.

We evaluated DeTrans with the STAMP benchmark suite, and we compared performance costs of DeTrans and Dthreads, the state-of-the-art deterministic system. Our results show that DeTrans is 3.99x, 3.39x, 2.44x faster on average than Dthreads for 2, 4, and 8 threads, respectively.

5

Increasing Concurrency in a Standard Library Invoked in Transactions

5.1 Introduction

In the previous chapter we proposed DeTrans, a system for deterministic execution of transactional applications, which could help a programmer in testing, debugging and providing fault tolerance. However, DeTrans has limited applicability and does not provide support for transactional applications that invoke external libraries, e.g. a C standard library (*libc*)[1].

Standard libraries abstract and simplify the access to operating system (OS) services and encapsulate shared data structures, e.g. memory allocation lists and file structures. In general, Transactional Memory (TM) is not able to track accesses and detect conflicts on shared data structures in a libc or an OS. Therefore, in order to guarantee atomicity

[1] Our deterministic system is suitable for C/C++ applications; therefore, the standard library these applications might invoke is the standard library for the C programming language.

and isolation, a TM library has to serialize a transaction that invokes libc functions or system calls. Before serialization, the transaction waits for all other transactions to finish their execution (either to commit their changes, or to discard them), and then it is re-executed as the only running transaction in the application.

Serialization might cause two issues in transactional application execution.

First, serialization hurts scalability and performance. To reduce the number of serialized transactions and to provide more concurrency in transactional applications researchers need a TM-aware standard library that would allow transactions to call standard library functions and to execute them concurrently. In this case, only transactions that invoke system calls would have to be serialized.

Second, serialization might induce deadlocks since it enforces the order of threads execution that might be different from the order enforced by a deterministic system. Therefore, deterministic execution with serialized transactions might be deadlock-prone. To prevent this, a deterministic system should support serialization enforced by TM.

In this chapter we address only the first issue – how to increase concurrency in transactional applications that invoke standard library functions inside transactions – and the second issue will be addressed later in this thesis.

Apart from serialization, another concern related to the invocation of a standard library in transactional applications is a possible interacation bettween transactions and other concurrency control mechanisms (e.g. locks). If a library is used in a large program, then its internal state might be accessed within transactions in some threads, and outside transactions in other threads using locks. This is especially the case for TM implementations with weak isolation guarantees like most of STM implementations. If an access to shared data structures is intended to be protected by locks, operations have to be implemented in a way that the interaction of locks and transactions cannot cause any unwanted or undefined behaviour [96]. In contrast, Hardware TM implementations like AMD's Advanced Synchronization Facility (ASF) [17] or Intel's Restricted TM (RTM) in the mainstream Haswell processor [79] provide strong isolation guarantees assuring consistent views even for unprotected memory accesses.

In this chapter, we present a TM-aware implementation of a C standard library, based on *diet libc* [98]. Diet libc is an open-source standard library designed to have the smallest possible code footprint. We modify the original lock-based implementation to use transactions as synchronization primitives without introducing new functions, system

calls, or instructions. Instead, we perform various modifications inside the library which are invisible to regular users. Other proposals of executing libc and system calls in transactions require some degree of change, in the form of a particular API [22, 97] or specialized transactional calls [76].

The rest of this chapter is organized as follows. In Section 5.2 we discuss issues that appear when a transactional application invokes libc functions inside transactions, and we suggest modifications of the application that are necessary for its successful compilation and execution. Without a TM-aware standard library, transactions that invoke libc functions have to be serialized, which limits concurrency in transactional applications.

In Section 5.3 we consider general design choices related to: (i) interaction between lock-based and transactional library code, (ii) standard library adaptation to a TM implementation, and (iii) standard library execution. Moreover, we explain which design choices are suitable for standard libraries.

In Section 5.4, we detail our experience about integration of diet libc and TM, which can be applied to other standard libraries and software in general. We introduce an extension to the existing TM conflict detection mechanism. It is a new technique to detect conflicts that cannot be detected automatically by TM because they modify kernel space. We also explain how to support transactions with system calls in HyTM [81] and avoid running them sequentially.

We quantify our effort for modifying a lock-based to a transactional standard library in Section 5.5. In Section 5.6, we explain the limitations of a *diet* standard library and existing TM tools, and we propose optimizations for transactional code.

In Section 5.7, we evaluate TM-dietlibc by using a set of benchmarks with file operations and memory management functions, and employing software, hardware and hybrid TM implementations. We show that for short transactions TM-dietlibc is scalable with a significant speedup for running on 8 cores. In addition, we demonstrate the first comparison of Intel's RTM implementation[1] with other TM implementations.

We conclude this chapter in Section 5.8.

[1] At time of this work, Haswell processors were still not available.

The ideas from this chapter were published at TRANSACT 2010[1] and IPDPS 2013[2].

5.2 Standard Library Function Calls in Transactions

In order to access shared data structures (e.g. a file buffer) concurrently in a libc and to request services of an OS (e.g. to write data to a file), a transactional application invokes libc functions inside transactions.

Listing 5.1 shows function `write_and_count` that should be executed concurrently by multiple threads. Each thread should perform two actions atomically: write the `len` number of characters to a shared file, and update the counter of the written characters.

```
1 long count_chars;  // global counter of chars written in the file
2
3 size_t fwrite(void* ptr, size_t size, size_t n, FILE *fp);
4
5 size_t write_and_count(void* ptr, size_t size, size_t n, FILE *fp)
6 {
7   size_t len;
8   __transaction_atomic {      // start a transaction
9     len = fwrite(ptr, size, n, *fp);      // call a libc function
10    count_chars += len;       // update the global counter
11  }                           // commit a transaction
12  return len;
13 }
```

Listing 5.1: Transactional code with a call of the `fwrite` libc function.

However, compiling this code with a compiler with TM support (GCC[3] 4.9) fails with an error

```
"unsafe function call ‘fwrite’ within atomic transaction"
```

[1] Nebojša Miletić, Vesna Smiljković, Cristian Perfumo, Tim Harris, Adrián Cristal, Ibrahim Hur, Osman S. Ünsal and Mateo Valero, Transactification of a Real-world System Library, In the 5th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 2010)

[2] Vesna Smiljković, Martin Nowack, Nebojša Miletić, Tim Harris, Osman S. Ünsal, Adrián Cristal, and Mateo Valero, TM-dietlibc: A TM-aware Real-world System Library, In Proceedings of 27th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2013)

[3] https://gcc.gnu.org/

when the compiler tries to instrument the code inside the transaction.

For the compiler, `fwrite` is a function of an external library, and without support for TM in this library, the function is not available for instrumentation. Therefore, we have to modify the program like in Listing 5.2. We declare `fwrite` as `transaction_pure` (line 3), and call the `serialize()` function (line 10) to execute the transaction serially. The `transaction_pure` attribute indicates GCC that the function does not have side effects and that it is safe to be called inside a transaction without instrumentation. Note that this does not stand for `fwrite`, since it accesses the shared file structure in the libc, and the shared file on the disk. However, since we guarantee that this function is always executed serially, it is safe to declare it as pure.

```
1  long count_chars;  // global counter of written chars in the file
2
3  __attribute__((transaction_pure))  // declare as pure
4  size_t fwrite(void* ptr, size_t size, size_t n, FILE *fp);
5
6  size_t write_and_count(void* ptr, size_t size, size_t n, FILE *fp)
7  {
8    size_t len;
9    __transaction_atomic {      // start a transaction
10     serialize();              // serialize the transaction
11     len = fwrite(ptr, size, n, *fp);     // call a libc function
12     count_chars += len;       // update the global counter
13   }                           // commit the transaction
14   return len;
15 }
```

Listing 5.2: Transactional code with a call of the `fwrite` libc function and the modifications needed for successful compilation and correct execution.

In `serialize()`, a transaction waits for all other running transactions to finish their execution, and then it gets restarted and executed as the only running transaction in the application. Frequent serialization might significantly decrease application performance, and a standard library compatible with TM could compensate this by reducing the number of serialized transactions and increasing concurrency in transactional applications.

5.3 Various Standard Library Designs

Developing a TM-aware standard library or changing an existing lock-based one involves choosing applicable programming models and designs. In this section, we describe different choices related to (i) how to *mix* locks and transactions, (ii) how to *adapt* a standard library to a TM library, and (iii) how to *execute* standard library code.

5.3.1 Mixing Locks and Transactions

Applications can invoke standard library functions from both transactional and non-transactional parts, and having lock-based code is still inevitable in some cases. For example, the functions that are part of TM initialization at application startup have to remain thread-safe, but should not contain transactions. Therefore, we must handle any interaction of locks and transactional boundaries as well as any interaction of lock-protected and TM-protected accesses to the same shared data.

Completely shared data. This programming model requires strong isolation provided by a TM implementation, which is not the case for most of the implementations in software. Only with strong isolation, transactional memory accesses are totally synchronized with unprotected memory accesses.

Partially shared data. Dynamic separation [2, 3] is a programming model where a programmer indicates shared variables that could be accessed inside or outside a transaction, and TM provides the necessary synchronization between transactional and non-transactional accesses.

Completely separated shared data. Some TM implementations require transactional and non-transactional data to be separated. The programming model is called static separation [4] and the standard library has to contain duplicated data structures and duplicated code wherever a memory access can be transactional and non-transactional.

TM-dietlibc design choice: We completely separate shared data protected by locks and protected by TM in TM-dietlibc. The reasons for avoiding mixing transactional and lock-protected data and code are the following: (i) breaking the TM isolation rule: a transaction should not be able to see the intermediate state of another transaction, e.g. when it directly accesses memory of a lock held by the other transaction, (ii) disabling

concurrent execution: if one transaction acquires a lock, all other transactions would have to wait until the first one finishes, and (iii) causing non-trivial pathological behaviour [96], e.g. a deadlock. The approach we choose is defensive and safe, and it does not require strong isolation from a TM implementation.

5.3.2 The Library Adaptation Level

Various TM implementations present a wealth of different features, algorithms and solutions in order to exploit better usability and performance. The level of standard library adaptation to one specific or various TM implementations influences its complexity and portability.

Library adapted to a specific TM implementation. A developer of a TM-based standard library could make design decisions depending on the chosen TM implementation. As each TM implementation has its policies regarding conflict detection, validation, isolation, nesting, etc., adopting a library to one TM implementation increases the performance and decreases the range of TM problems that a developer could face modifying or developing a standard library.

Library independent of TM implementations. Employing different TM implementations becomes straightforward with: Intel's ABI [18], the compilers that conform to this ABI convention, and STM libraries that are compatible with these compilers. However, a standard library that does not depend on a specific TM implementation has to rely on common TM features which weakens TM optimization and exploitation opportunities.

TM-dietlibc design choice: We adapt TM-dietlibc to be compatible with different TM implementations [17, 31, 81], all compatible to Intel's ABI [18], with flat nesting and eager conflict detection. Flat nesting allows deferring actions from the outer or any inner transaction until the commit phase of the outer one. Eager conflict detection provides the discovery of a conflict at the moment of a conflicting data access, and immediate re-execution of the aborted transaction. Relying on these TM features, we enable system calls to be executed if no conflicts with other transactions occur. For TM with lazy conflict detection, transactions with system calls are executed serially.

5.3.3 The Library Execution Mode

To exploit optimistic concurrency that TM provides, the ideal case is when standard library functions access only local data, or user-space shared data, and can be completely synchronized by TM. However, more frequent cases are when functions can be only executed as transactional with the developer's usage of additional TM mechanisms, or they have to remain as non-transactional. Therefore, we describe different possibilities to run standard library functions.

Complete transactional execution. Libc code can be executed transactionally if it contains only local variables and user-level shared variables.

Transactional execution employing TM techniques. More sophisticated TM implementations are able to handle non-trivial cases, e.g. to support locking inside transactions.

Transactional execution in a TM-adapted standard library. Library code is transformed to transactional code, but has to be modified to allow the exploitation of TM. For instance, a lack of support for locks within transactions requires removing locks and using another synchronization mechanism whenever needed.

Sequential execution. Transition of a transaction that contains a standard library call to serial execution ensures safe execution without any libc changes. However, executing transactions sequentially hurts parallelism and scalability of programs.

Non-transactional execution within TM integration. A standard library can be executed non-transactionally, but with a certain adaptation for integration with TM. For instance, the Intel [18] and DTMC [17] compilers provide deferral and compensation actions for memory management functions and allow non-transactional execution inside a transaction.

TM-dietlibc design choice: We use a combination of four design choices: (i) complete transactional execution is possible when there are no system calls during the execution, (ii) transactional execution employing TM techniques is preferable when some system calls occur, and we employ abort and commit handlers provided by TM, (iii) transactional execution in a TM-adapted standard library is for the cases when we apply the conflict detection extension implemented in TM-dietlibc, and (iv) sequential execution for the system calls where we cannot employ an abort or a commit handler.

TM-dietlibc provides transactional execution of its functions, relying on the TM im-

plementation to handle only trivial data sharing. For nontrivial cases, we (i) employ abort handlers for compensation and commit handlers for deferral; (ii) apply conflict detection extensions, and (iii) infrequently transit to serial execution. The library execution mode is detailed in the next section.

5.4 The Transactification of Diet Libc - Implementation Experience

In this section, we present details on how we modified the lock-based diet libc in order to integrate it with TM. Some of the modifications are simple; however, the majority required significant effort to: (i) identify groups of locks which are used to protect access to shared data structures, (ii) replace them with transaction boundaries, (iii) use TM techniques in the appropriate way for standard library functions, (iv) implement and apply a TM conflict detection extension, and (v) support hybrid TM.

The experience we gained during the transactification[1] can be used for other standard libraries, irrespective of them being "diet" or not, e.g. glibc[2], EGLIBC[3] and uClibc[4], or for writing a TM-aware standard library from scratch. We assume that standard library developers would face many challenges we faced during the transactification of diet libc.

5.4.1 Identifying Groups of Locks

A standard library contains various synchronization primitives to control accesses to its critical sections. The first challenge is identifying different groups of locks and choosing groups that are in our area of interest. Since we do not want interaction between locks and transactions, all locks and locking operations from a chosen group should be replaced with appropriate TM support.

```
int fgetc(FILE* fp)
{
  int r;

  _IO_acquire_lock(fp);
  r = _IO_getc_unlocked(fp);
  _IO_release_lock(fp);

  return r;
}
```

(a) glibc/EGLIBC

```
int fgetc(FILE* fp)
{
  int r;

  MUTEX_LOCK(fp->_lock);
  r = fgetc_unlocked(fp);
  MUTEX_UNLOCK(fp->_lock);

  return r;
}
```

(b) uClibc

```
int fgetc(FILE* fp)
{
  int r;

  pthread_mutex_lock(fp->m);
  r = fgetc_unlocked(fp);
  pthread_mutex_unlock(fp->m);

  return r;
}
```

(c) diet libc

```
int fgetc(FILE* fp)
{
  int r;

  __transaction_atomic {
    r = fgetc_unlocked(fp);
  }

  return r;
}
```

(d) TM-dietlibc

Figure 5.1: Lock-based implementations of `fgetc` for different standard libraries (a), (b), (c) and the transactional counterpart (d). Locking operations in diet libc are replaced with transactional boundaries `__transaction_atomic{}`.

5.4.2 Defining Critical Section Boundaries

Defining critical section boundaries is trivial when it is easy to recognize locking operations and localize them in one function. In other cases, the operations might be missing, hidden behind macros, or operations for acquiring and releasing the same lock might be located in multiple files.

Simple lock-operation pairing. Defining critical section boundaries is straightforward when the functions `lock` and `unlock` are paired up and located inside a single

[1] also called transactionalization in Ruan et al. [84]
[2] http://gnu.org/software/libc/
[3] http://eglibc.org/
[4] http://uclibc.org/

function (shown in Figure 5.1 for different standard libraries (a), (b), (c)). This way, they can be easily replaced with the boundaries of a transaction (Figure 5.1(d)).

Locking operations missing. Some of the library functions in the original lock-based implementation are left to be unsafe on purpose, i.e. declared to be a weak alias for a non thread-safe version. Since the thread-safe implementation of these functions exists in other standard libraries, we wrap them with transactional boundaries to make them thread-safe.

Locking operations of lexically unstructured critical sections. The examples we encounter in diet libc are: (i) when `lock`/`unlock` pairs do not satisfy a TM requirement of having critical section boundaries in one function scope, and (ii) when the code flow can lead from one `lock` to multiple `unlock`s, meaning that it is not possible to establish one-to-one relationships between them. Lexically unstructured critical sections require manual program-flow analysis from the starting point of the critical section until all possible ending points, and gathering the code distributed in different functions into a single transaction.

However, transactional boundaries are sufficient for TM to provide atomicity and isolation when a transaction contains only local variables or shared variables at the user level, which is not a common case for a standard library.

5.4.3 Applying TM Techniques on the Library Functions with System Calls

Various functions from a standard library make modifications in kernel space that TM cannot track, or they cause side effects that TM cannot revert. To illustrate these cases and our design choices in practice, we use memory management and file operations.

Memory management functions operate over arrays of pre-allocated memory chunks, and they invoke a system call `mmap` only when no free chunks remain in the chunk array. Similarly, a system call `munmap` is called only when the size of the memory ready to be released is greater than the acceptable size of chunks. Compilers with TM support (DTMC [17] and Intel [18]) wrap original lock-based functions, add additional structures and use commit and abort handlers.

Since our goal is to ensure concurrent execution of memory management operations, we do not rely on the compiler's wrappers. Instead, we provide: (i) speculative execution

of these functions, (ii) an abort handler, used for the functions that allocate memory, and (iii) a commit handler, used for the functions that release memory. The usage of abort and commit handlers is shown in Figure 5.2(a) and (b). In our implementation, the handlers are registered only when a system call occurs. In all other cases, the TM implementation is sufficient to deal with accesses to shared variables in user space.

In addition, as the memory management is a vital part of the TM implementation itself, e.g. for managing buffers for transactional reads and writes, it is necessary to keep the original lock-based functions and to separate chunk arrays of transactional from non-transactional usage (`libc_chunks` and `tx_libc_chunks` in Figure 5.2).

Although TM-dietlibc provides two different implementations for every memory management function (`malloc` and `tx_malloc`, `free` and `tx_free`), applications call only the original functions no matter if the calls are inside or outside transactions. When a TM compiler instruments an application, it finds the calls from inside transactions and replaces them with their transactional counterparts. Therefore, the API remains unchanged, and applications do not require any modifications.

File operations provide communication between (i) a user and a program and (ii) a program and an operating system. Many of them have visible and nonreversible side effects; therefore, if they are invoked within a transaction, the transaction has to be executed serially. In this case, the transaction waits for the other running transactions to finish their execution, and then it continues as the only running transaction in the application. When the serialized transaction commits, other transactions are allowed to start and they are executed again in parallel.

I/O functions operate over a shared libc structure called `FILE`. This structure contains a storage buffer for parts of a file, pointers to the next and the last character in the buffer, etc. Only in cases when the buffer is empty, full, or changes need to be applied to disk, the library calls `read`, `write` and `lseek` to fill the buffer, empty the buffer, and update the file position, respectively. Since I/O operations invoke file changes in the kernel space occasionally, we allow *late serialization*, i.e. a transaction executes concurrently until a system call occurs and only then the TM library changes the execution mode of the transaction – from parallel to serial (illustrated with an update to disk in Figure 5.3).

```
void* malloc(size_t size)
{
  void* r;

  pthread_mutex_lock(&mutex_alloc);

  if(libc_chunks.empty()) {
    r = mmap(size);

  }
  else {
    r = libc_chunks.pop();
  }

  pthread_mutex_unlock(&mutex_alloc);

  return r;
}

// syscall:
void *mmap(size_t size);
```

```
void* tx_malloc(size_t size)
{
  void* r;

  __transaction_atomic {

    if(tx_libc_chunks.empty()) {
      r = mmap(size);
      onAbort(munmmap, r);
    }
    else {
      r = tx_libc_chunks.pop();
    }

  }

  return r;
}

// syscalls:
void *mmap(size_t size);
int munmap(void* addr);
```

(a) Lock-based (left) and TM-based (right) implementations of `malloc`

```
void free(void* ptr)
{
  pthread_mutex_lock(&mutex_alloc);

  if (libc_chunks.full()) {
    munmap(ptr);
  }
  else {
    libc_chunks.push(ptr);
  }

  pthread_mutex_unlock(&mutex_alloc);

}

// syscall:
int munmap(void* addr);
```

```
void tx_free(void* ptr)
{
  __transaction_atomic {

    if (tx_libc_chunks.full()) {
      onCommit(munmap, ptr);
    }
    else {
      tx_libc_chunks.push(ptr);
    }

  }

}

// syscall:
int munmap(void* addr);
```

(b) Lock-based (left) and TM-based (right) implementations of `free`

Figure 5.2: Examples of lock-based (left) and TM-based (right) libc functions for memory allocation: (a) `malloc` with the abort handler and the distinct structure for transactional access to provide static separation, (b) `free` with the commit handler and the same structure as in `malloc`.

```
size_t fwrite(void* ptr, size_t size,
              size_t n, FILE* fp)
{
  size_t len = size * n;

  pthread_mutex_lock(fp->m);

  if (!fp->buf.full()) {
    memcpy(fp->buf, ptr, size);
  }
  else {

    len = write(fp->fd, ptr, size);
  }

  pthread_mutex_unlock(fp->m);

  return len;
}

// syscall:
size_t write(int fd, void* buf,
             size_t size);
```

```
size_t fwrite(void* ptr, size_t size,
              size_t n, FILE* fp)
{
  size_t len = size * n;

  __transaction_atomic {

    if (!fp->buf.full()) {
      memcpy(fp->buf, ptr, size);
    }
    else {
      serialize();
      len = write(fp->fd, ptr, size);
    }

  }

  return len;
}

// syscall:
size_t write(int fd, void* buf,
             size_t size);
```

Figure 5.3: Examples of lock-based (left) and TM-based (right) function `fwrite`. TM-based `fwrite` employs the late serialization, i.e. executing serially only if a system call is invoked.

5.4.4 The Conflict-Detection Extension

The mechanisms explained so far are sufficient for integrating any code with TM. However, executing serial transactions impacts parallelism, and threads spend most of the time waiting for a serialized transaction to finish execution and commit changes. One of the examples when TM should run a transaction serially is when the transaction causes side effects in kernel space. Kernel space is out of the scope of TM; therefore, TM cannot observe changes the transaction makes and cannot detect conflicts with other transactions.

In order to reduce the number of serial transactions running in an application, we propose an extension for the TM conflict detection mechanism. The extension ensures that: (i) transactions run concurrently, (ii) TM keeps track of updates of data structures in kernel space, and (iii) TM detects conflicting accesses to these structures.

Our proposal creates and maintains copies of the relevant shared data from kernel space that are accessed inside transactions. These copies reside in user space and they are under complete control of a programmer and the TM library. The standard library programmer is responsible for making a copy of the data and for keeping the copy updated

according to the always up-to-date kernel data. Based on the copy, the TM library can detect conflicts and invoke abort handlers to revert the state of kernel space in case the transaction aborts.

We illustrate our approach using the `fseek` function. Figure 5.4(a) shows the original (lock-based) implementation of the function, and Figure 5.4(b) the TM-based implementation with the extension for the conflict detection. Since the `lseek` system call changes the file pointer in kernel space, a standard library programmer has to make a copy of the file pointer in user space and to keep the copy updated. In our example the copied variable is `OS_fpos`, and it is a part of the `FILE` structure in TM-dietlibc. With the shared variable stored in user space, a TM library is able to detect conflicting memory accesses.

In order to detect a conflict before invoking the `lseek` system call, the transaction tries to acquire a writing lock for the update of `OS_fpos`[1]. If another thread holds the lock, TM detects the conflict before the system call, aborts the transaction and rolls back returning the old values of the shared variables. On the other hand, if an abort happens after `lseek`, TM calls the undo function which invokes a call of `lseek` with the earlier stored file position value. Whenever the abort occurs, the states of user and kernel space are rolled back to how they were before the transaction started its execution.

5.4.5 The System-Call Barrier in HyTM

The different approaches detailed in the previous sections allow concurrent multithreaded executions of transactions with system calls inside of software transactions. However, TM using hardware support does not allow system calls inside a running hardware transaction; therefore, it aborts the transaction and re-execute it serially.

To increase the possibility of the parallel execution of transactions with system calls, we propose a *safe-syscall* execution mode. Before each system call in a transaction, we put a *go_safe_syscall* barrier (Figure 5.4(c)), which notifies HTM about an upcoming system call. HTM aborts and re-executes the transaction using software fall back solutions, thus it increases the number of software and hardware transactions that run in parallel.

[1] In eager implementations of STMs a transaction acquires a lock to update a shared variable, and all the locks acquired during execution of the transactions are released at commit or abort.

```
int fseek(FILE* fp, long offset, int whence)
{
  int r; param* p;

  pthread_mutex_lock(fp->m);

  r = lseek(fp->fd, offset, whence);

  pthread_mutex_unlock(fp->m);

  return r;
}

// syscall:
int lseek(int fd, long offset, int whence);
```

(a) Lock-based fseek

```
int fseek(FILE* fp, long offset, int whence)
{
  int r; param* p;

  __transaction_atomic {

    p = {fp->fd, fp->OS_fpos};



    //cause a possible conflict:
    acquire_wr_lock(fp->OS_fpos);

    r = lseek(fp->fd, offset, whence);

    fp->OS_fpos = r; //update the pointer

    onAbort(undo_lseek, p);

  }

  return r;
}

void undo_lseek(void* p) {
  lseek(p->fd, p->pos, SEEK_SET);
}

// syscall:
int lseek(int fd, long offset, int whence);
```

(b) Conflict detection for STM.

```
int fseek(FILE* fp, long offset, int whence)
{
   int r; param* p;

   __transaction_atomic {

     p={fp->fd, fp->OS_fpos};

     go_safe_syscall; //abort a hw txn:

     //cause a possible conflict:
     acquire_wr_lock(fp->OS_fpos);

     r=lseek(fp->fd, offset, whence);

     fp->OS_fpos = r; //update the pointer

     onAbort(undo_lseek, p);

   }

   return r;
}

void undo_lseek(void* p) {
  lseek(p->fd, p->pos, SEEK_SET);
}

// syscall:
int lseek(int fd, long offset, int whence);
```

(c) Conflict detection for HyTM.

Figure 5.4: Examples of the `fseek` implementation (a) lock-based, (b) TM-based for running with STM, and (c) TM-based for running with HyTM. The variable `OS_fpos`, reflecting the current position in a file stored in kernel space, is used for detecting conflicts in user space. In addition for HyTM (c), a `go_safe_syscall` barrier aborts a hardware transaction and re-execute it concurrently in software.

5.5 Quantifying Software Development Effort

The significance of the effort needed in modifying diet libc is in its integration in a complex TM implementation. Each TM principle implemented in the standard library or used as an existing TM tool was the result of a time-consuming investigation, rather than complex code writing. We present quantified effort in terms of code modifications, transactions complexity and consumed time.

The number of lines of code (LoC) of C and Assembly in diet libc implementations is: 64k LoC for the original diet libc (version 0.33) and 71k LoC for TM-dietlibc. Therefore, 7k lines of code were added for the transactional version of dietlibc. As an example of modifications, the file operations in the original diet libc contain 20 critical sections. In comparison, 25 transactions were inserted into TM-dietlibc. The difference arises from the fact that we modified original unsafe functions to use transactions, as well (described in Section 5.4.2).

The type and the length of transactions depend on different code flows. For example, if an `fputc` operation is invoked, the character might fit into the internal buffer leading to the selection of a very short transactional code path with less than 10 lines of code of interest (LoCI)[1]. Otherwise, `write` and `seek` operations occur leading to longer and more complex paths, with almost 100 LoCI.

We developed TM-dietlibc during a three-year period. At the beginning, appropriate TM tools and TM benchmarks were not mature and needed to include support for building a TM-aware standard library.

5.6 Limitations of a "Diet" Standard Library and TM Tools

Choosing a standard library with a small software footprint can make applying changes and new designs easier and time-saving. A developer deals with fewer and less complex structures and functions. On the other hand, the "diet" library presents additional obstacles, e.g. missing function implementations, missing thread local storage support, a small initial stack size, etc. Although the problems might sound trivial, without being

[1] The code of interest is the instrumented and executed code.

```
char* fgets(char* s, int N, FILE* fp)
{
  pthread_mutex_lock(&fp->m);

  for (int l=0; l<size; l++) {

    //read shared variables N times:
    int c = fp->buf[fp->bm];

    //update shared variable N times:
    fp->bm++;

    s[l] = c;

    if (c=='\n' || c==EOF) {
      s[l]=0;
      break;
    }
  }

  phread_mutex_unlock(&fp->m);

  return s;
}
```

(a) Original lock-based `fgets`

```
char* fgets(char* s, int N, FILE* fp)
{
  __transaction_atomic {

    //read shared variables once:
    int bm = fp->bm;
    for (int l=0; l<size; l++) {
      int to_read;

      //get position of newline or EOF:
      if (!(to_read = findChar(&fp->buf[bm],'\n')) &&
          !(to_read = findChar(&fp->buf[bm],EOF)) {
        to_read = N; // no newline, no EOF
      }

      //copy as a string:
      memcpy(&s[l], &fp->buf[bm],to_read);

      //update position in buffer
      bm += to_read;
      // iterate faster
      l += to_read;
    }
    //update once:
    fp->bm = bm;
  }
  return s;
}
```

(b) Optimized TM-based `fgets`

Figure 5.5: An optimization of `fgets` using local variables to avoid additional, unneeded transactional accesses that might cause conflicts among threads.

aware of them, the behaviour of some benchmarks was unpredictable and unclear.

The implementations of some libc functions are not well suited for optimistic TM concurrency, and cause an influential contention on shared resources. For instance, function `fgets` reads a string from a file. First, it prefetches and fills a shared buffer with a part of the file, and then reads characters - one by one - from the buffer, and increments a shared buffer pointer for each character. As a result, reading more characters makes the transaction longer, the number of transactional memory accesses larger, and conflicting situations more likely. Our optimization consists of reading as many characters as possible and using local instead of shared variables for intermediate values (Figure 5.5). This way, we reduce the number of instrumented memory accesses and lower the running overhead.

Finally, the lack of debugging and profiling tools for various TM libraries at the

time when we were transactifying the standard library made testing, debugging and performance tuning substantially difficult and time-consuming. Even general-purpose debuggers like gdb[1] were not able to debug an application if it invoked diet libc.

Furthermore, TM libraries and TM compilers were developed for the application usage, rather than for the usage of standard libraries, and many modifications were needed to ensure a TM-aware standard library to be successfully built. For successful compilation and execution of benchmarks invoking TM-dietlibc, the modifications were applied in the standard library, the compiler and the TM library.

5.7 Evaluation

We evaluate TM-dietlibc with three types of benchmarks: (i) microbenchmarks with I/O operations, (ii) TioBench [95] with I/O operations, and (iii) Red-black tree [32] with memory management functions, all running on 2 Intel Xeon E5405 processors with 8 cores in total. To verify the the correctness of the TM-dietlibc implementation, we compared executions of tests linked against TM-dietlibc with executions of tests with the same input parameters, but linked against glibc[2].

Details about the experimental setup are given in Chapter 3.

5.7.1 Methodology

The microbenchmarks we implemented for evaluation of TM-dietlibc invoke file operations: `fgetc`, `fgets`, `fread`, `fputc`, `fputs`, and `fwrite` inside transactions, and calculations on thread-local variables outside transactions. Multiple threads access a single file using a shared file descriptor and its associated `FILE` structure.

TioBench performs `fopen`, `fclose`, `fseek`, `fread`, and `fwrite` to write data to and read data from a shared file sequentially or randomly (*seq write*, *rnd write*, *seq read*, *rnd read*).

Red-black tree performs read, write and removal operations over the elements of a balanced tree structure. It invokes memory management functions `malloc` and `free` inside transactions when creating and removing nodes, respectively.

[1] http://www.gnu.org/software/gdb/
[2] http://gnu.org/software/libc/

For evaluation with the microbenchmark and Red-black tree, we compiled TM-dietlibc and the benchmarks with DTMC [17], and for evaluation with TioBench, we compiled TM-dietlibc and the benchmark with GCC[1], version 4.9.

We run the microbenchmarks linked against STM, HTM, and HyTM implementations, and TioBench and Red-black tree exclusively with the STM implementation. The STM implementation is TinySTM [31], the HTM is the Advanced Synchronization Facility (ASF) extension [17] from AMD, and the HyTM library is HyLSA [81]. The executions involving HTM and HyTM were conducted using a nearly cycle-accurate CPU simulator PTLsim [105] with the ASF extension.

In addition, we emulated the Intel Haswell processor [79] to employ Restricted Transactional Memory (RTM) and to compare it with other TM implementations.

The simulations were performed on the same machine as the other experiments for evaluation of TM-dietlibc.

Without TM support in a standard library, a programmer would have to modify an application to switch to serial execution before calling library functions inside transactions (see Section 5.2 for details). To compare our implementation with this approach, we compiled two versions of our TM-aware library: (i) transactional employing various TM implementations (TinySTM, HyLSA, ASF, and RTM), and (ii) transactional with switching to serial execution at the beginning of a transaction (Serial-TM), which is the only option without proper TM support in a standard library,.

5.7.2 Results

In Figure 5.6, we show that for the microbenchmarks with short transactions (reading or writing a single character per transaction), transactional versions perform better than the serial version (Serial-TM) on average. In comparison to Serial-TM, TM-dietlibc provides on average 2.9x (*TinySTM*), 4.1x (*ASF*), and 4.2x (*RTM*) performance speedup for 8 running threads. However, HyLSA does not perform as well as STM and HTMs. Due to many aborts, transactions have to be restarted in software-transactional mode. This slows down the execution on average 1.04x in comparison to Serial-TM execution for 8 running threads.

TioBench running I/O operations with larger transactions (reading or writing 8 char-

[1] https://gcc.gnu.org/

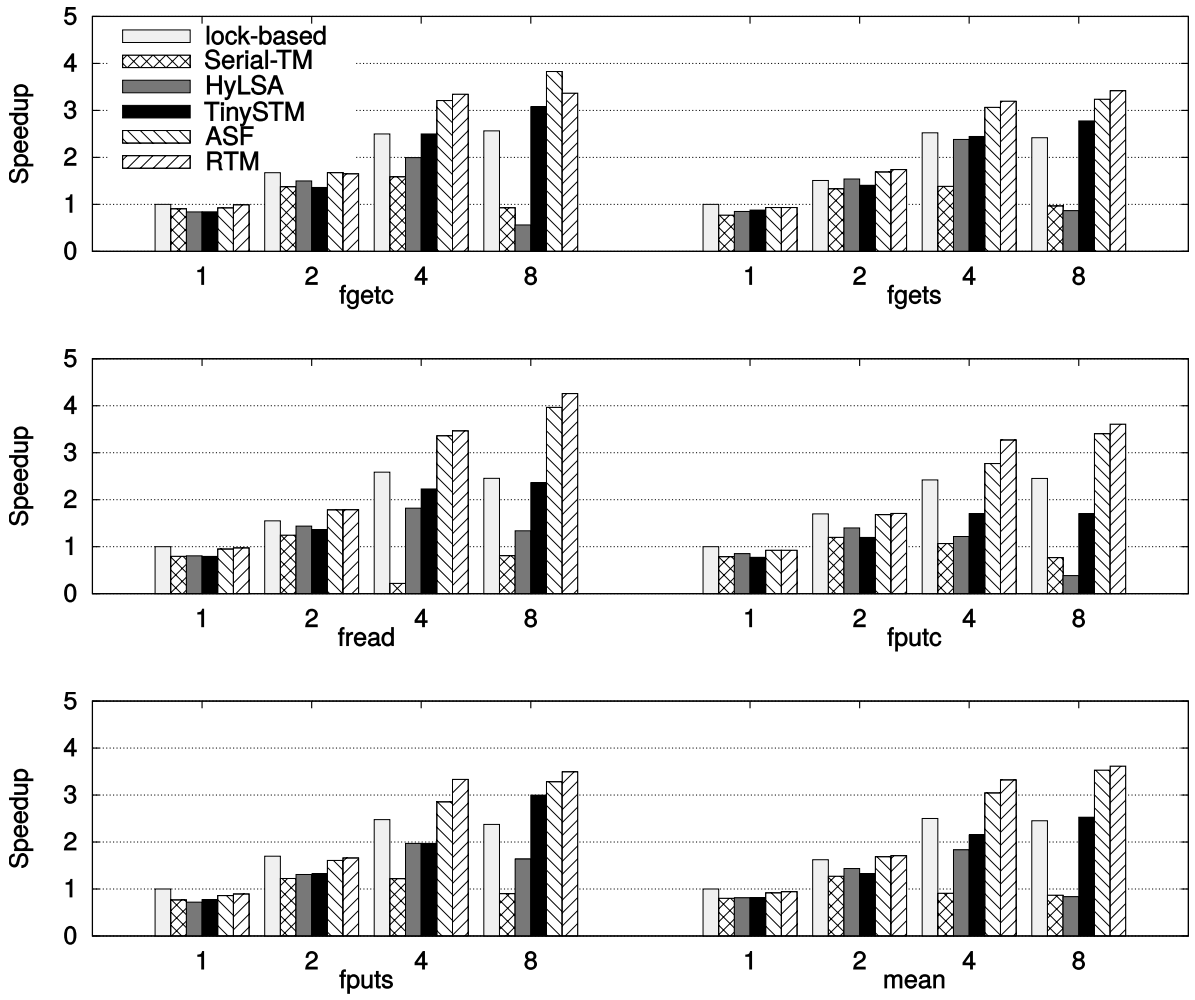


Figure 5.6: Evaluation of file operations executed by 1, 2, 4 and 8 threads, with various TM implementations and normalized to a lock-based single-threaded execution. We show speedup (higher is better).

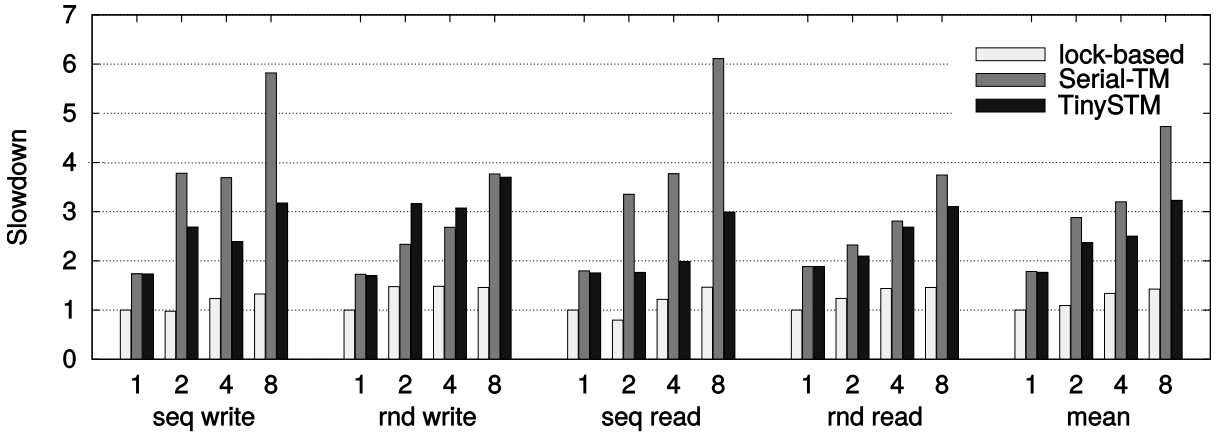


Figure 5.7: Evaluation of TioBench by 1, 2, 4 and 8 threads, with locks, Serial-TM, and TinySTM, all normalized to the lock-based single-threaded execution. We show slowdown (lower is better).

acters per transactions) do not scale, although running concurrently with 8 threads and with TinySTM is 1.5x faster than running serially (Serial-TM) (Figure 5.7)[1].

Increasing the number of characters in the microbenchmarks and TioBench decreases performance due to frequent conflicts among threads, so that the benchmarks neither scale nor are faster than serial transactional execution.

To evaluate memory management functions, we use Red-black tree. We varied the ratio of reads to other operations (25%, 50% and 75%), and ran the benchmark with the original implementation (*original*), and with TM-dietlibc running transactions serially (*Serial-TM*) and in parallel (*TinySTM*) (Figure 5.8)[2].

With fewer reads, the benchmark performs more writes and removals, which requires memory allocation and deallocation. Therefore, general performance is lower in comparison to more read-dominated executions. However, the differences in performance between the transactional and serial versions are larger for the benchmarks with many memory operations. On average, performing memory operations concurrently with TinySTM and 8 running threads is 1.7x faster than serial transactional execution.

For the evaluation of TM-dietlibc we are not able to use any other transactional

[1]The evaluation was extended by this benchmark at the time of writing this dissertation, and since we had difficulties to rebuild the evaluation environment, we omit the evaluation with HyLSA, ASF, and RTM.

[2]The original benchmark is 32-bit, which does not permit us to use the simulator and employ HyLSA, ASF, or RTM.

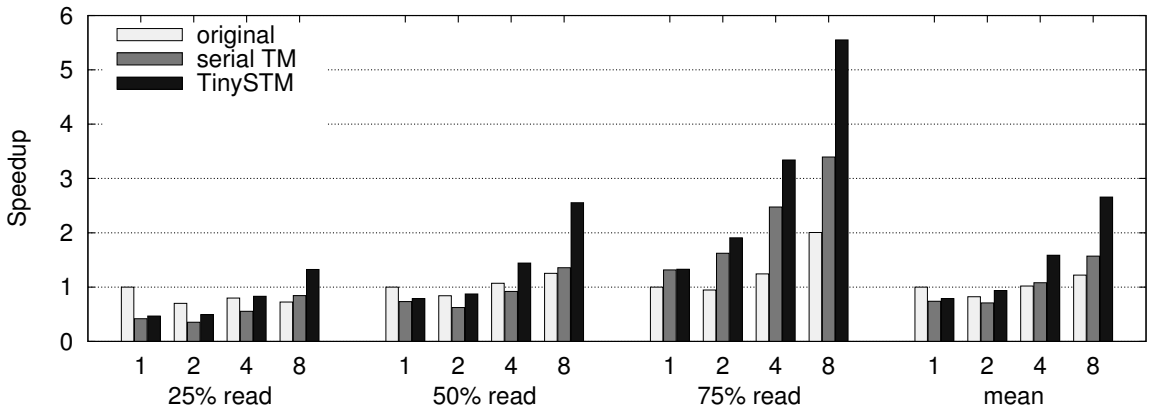


Figure 5.8: Evaluation of Red-black tree by 1, 2, 4 and 8 threads, with locks, Serial-TM, and TinySTM, all normalized to the original single-threaded execution. We show speedup (higher is better).

benchmarks since they are either without standard library calls (STAMP [65] and Eigenbench [46]) or they postpone these calls until non-transactional code (Atomic Quake [106] and Memcached [84]).

5.8 Summary

In this chapter, we presented the first real-world TM-aware standard C library implementation with the unmodified API. We described various design choices for integrating a standard library, or any other software, with TM and the design choices most suitable for diet libc. We proposed static separation of locks and transactions as a safe way to handle the interaction of these two different synchronization concepts. We discussed handling system calls inside transactions and revealed a pitfall in detecting kernel space conflicts that would require many transactions to be serialized. To solve this, we proposed a technique that enables detection of such conflicts in the scope of the standard library, rather than involving complex kernel modifications.

We provided the first comparison of emulated RTM with other TM implementations. Our results for memory management and file operations showed that, for the benchmarks with short transactions and various TM implementations, TM-dietlibc performs better than TM-based serialized execution, which is the only option for invoking a standard library without support for TM.

The effort we put into diet libc to integrate it with a TM system (a compiler with TM support and various TM implementations) and the experience we gained while transactifying this library showed that even a "diet" standard library is complex software. As a consequence, applications are even more difficult to develop, test and debug when they invoke standard library functions.

6

Deterministic Execution in a Standard Library

6.1 Introduction

In the previous chapter we introduced TM-dietlibc, the first TM-aware standard library that applies multiple techniques to allow transactions with standard library functions to be executed concurrently. TM-dietlibc helps TM to track accesses to shared data structures in the standard library and the kernel, but some changes cannot be postponed or reverted; therefore, transactions that invoke system calls like `write` still have to be serialized so that TM can guarantee atomicity and isolation of these transactions.

Before a transaction is serialized, it waits for all other transactions to finish their execution (either to commit their changes, or to discard them), and then it is re-executed as the only running transaction in the application. However, if running deterministically, while a transaction is waiting for other transactions to finish their execution, the other transactions might be waiting for their turn to be able to commit in deterministic order.

When the deterministic system and TM try to enforce different orders of transactions committing their changes, or different orders of threads execution in general, they are prone to deadlocks.

In this chapter we present *DeTrans-lib*, a TM-based standard C library that ensures deterministic multithreading at application and standard-library level. It is based on a runtime (DeTrans) that provides deterministic execution even in the presence of data races, and a TM-aware libc (TM-dietlibc) that allows transactions to execute libc functions concurrently, and serializes them only if a libc function invokes a system call with no-reversible side effects. In addition, DeTrans-lib ensures that transactions invoke system calls in deterministic order, which prevents deadlocks caused by serialization of transactions in deterministic execution.

The rest of this chapter is organized as follows. In Section 6.2. we show an example of a transaction in TM-dietlibc that has to be serialized in case it invokes a system call, and we explain why a deterministic system has to provide support for serialization due to system calls.

In Section 6.3 we describe the DeTrans-lib design. We port the DeTrans deterministic system in TM-dietlibc to ensure deterministic multithreading at application and standard-library level. DeTrans-lib ensures that threads invoke system calls in deterministic order, and it avoids deadlocks caused by busy-waiting in serialization (enforced by a TM library due to system calls) and busy-waiting in deterministic execution (enforced by DeTrans).

DeTrans-lib is evaluated in Section 6.4. We verify the correctness of the DeTrans-lib implementation by using stress test Racey [45, 103]. For the evaluation, we use benchmarks that invoke libc I/O calls: microbenchmarks and modified TioBench [95]. For the microbenchmarks and TioBench invoking DeTrans-lib, the maximum average slowdown in comparison to original (nondeterministic) single-threaded execution is 4.53x and 2.29x, respectively, for 8 running threads.

We conclude this chapter in Section 6.5.

This chapter present the ideas published at NPC 2015 and IJPP 2015[1].

[1]Vesna Smiljković, Osman S. Ünsal, Adrián Cristal, and Mateo Valero, Determinism at Standard-Library Level in TM-Based Applications, In Proceedings of the 12th Annual IFIP International Conference on Network and Parallel Computing (NPC 2015), and International Journal of Parallel Programming (IJPP 2015), special edition for NPC.

6.2 Standard-Library Calls in Deterministic Execution

In order to access shared data structures (e.g. a file buffer) in a libc and to request services of an OS (e.g. to write data to a file), a transactional application might invoke libc functions inside transactions.

Listing 6.1 (similar to Listing 5.2) shows function `write_and_count`, where a thread performs two actions atomically. It writes the `len` number of characters to a shared file and updates the counter of the written characters.

If this code invokes a standard library without TM support, `fwrite` is declared as `transaction_pure` (line 4), and the transaction has to be serialized (line 13). On the other hand, TM support in TM-dietlibc (see Chapter 5) allows threads to concurrently execute transactions with standard library function calls and serializes only the transactions with non-reversible side effects, e.g. when system call `write()` is invoked (see Section 5.4.3 for details).

Serialization, implemented in a STM library (Listing 6.2), requires that the transaction that has to be serialized waits for all other running transactions to finish their execution (line 5), so that it gets restarted and executed as the only running transaction in the application. If an application is running deterministically, the serialization might enforce the order of threads execution that is different from the one enforced by a system for deterministic multithreading, causing a deadlock.

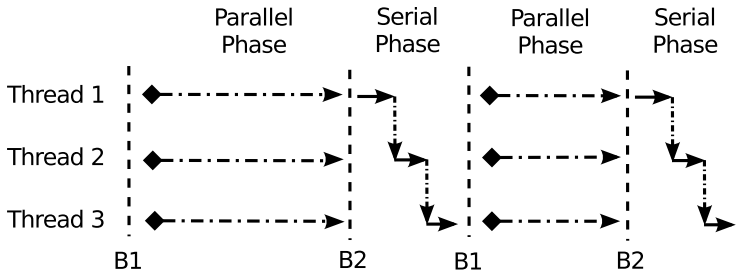
As explained in the previous chapter, DeTrans is a runtime system that ensures deterministic multithreading in transactional applications. It implements the *double-barrier technique* and *deterministic-token passing* (Figure 6.1(a)). In Figure 6.1(b) we show how application threads cannot make any progress if a thread (`Thread1`) waits in the serialization due to a libc function call while holding the deterministic token, and the other threads (`Thread2` and `Thread3`) wait for `Thread1` to commit its transaction and pass the token.

As a consequence, an application running deterministically might be prone to a deadlock caused by busy-waiting in transaction serialization (Listing 6.2 line 5), and a deterministic system has to be aware of synchronizations that are necessary to invoke libc functions inside transactions in applications.

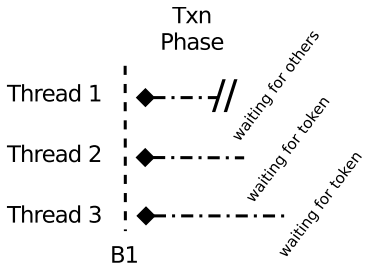
```
1  long count_chars;  // global counter of written chars in the file
2
3  #ifndef TM_SUPPORT                    // if libc doesn't support TM
4    __attribute__((transaction_pure))  // declare as pure
5    size_t fwrite(void* ptr, size_t size, size_t n, FILE *fp);
6  #endif
7
8  size_t write_and_count(void* ptr, size_t size, size_t n, FILE *fp)
9  {
10   size_t len;
11   __transaction_atomic {      // start a transaction
12 #ifndef TM_SUPPORT            // if libc doesn't support TM
13     serialize();              // serialize the transaction
14 #endif
15     len = fwrite(ptr, size, n, *fp);     // call a libc function
16     count_chars += len;       // update the global counter
17   }                           // commit the transaction
18   return len;
19 }
```

Listing 6.1: An example of transactional code that has to be serialized if the code invokes a standard library without support for TM (when macro `TM_SUPPORT` is undefined) or can be executed concurrently if the code invokes TM-dietlibc (when macro `TM_SUPPORT` is defined)[a].

[2]Note that TM-dietlibc does not require modifications of transactional applications, and macro `TM_SUPPORT` is added in this example only for comparison of code that invokes a standard libraries with and without TM support.



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



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

Figure 6.1: (a) DeTrans implements the double-barrier technique and passes the deterministic token in round-robin order to ensure deterministic execution. (b) Serialization in a STM library is prone to deadlock when running deterministically.

```
1 void serialize()
2 {
3   if(this->tx->isSerialized)  // check if the transaction
                                // is already serialized
4     return;
5   while (others.areExecuting()){} // wait for others to finish
6   this->tx->isSerialized = 1;
7   restart();                      // restart the transaction
8   return;
9 }
```

Listing 6.2: Serialization in a STM library.

6.3 Design of DeTrans-lib

In this chapter, we propose DeTrans-lib, the first TM-aware libc that provides deterministic execution of transactional applications. For this, we needed to port deterministic system DeTrans, which guarantees deterministic execution at application level, in TM-dietlibc to provide determinism at libc level[1]. In addition, DeTrans-lib guarantees deterministic execution of libc functions and system calls, and avoids deadlocks caused by serialization in TM.

We ported DeTrans by copying and adjusting structures and functions from the DeTrans wrappers to TM-dietlibc. We modified `pthread_*` functions directly and added additional functions that are called before and after `_ITM_*` functions in the STM library. This way, the implementation remains in TM-dietibc and does not require modifications of benchmarks.

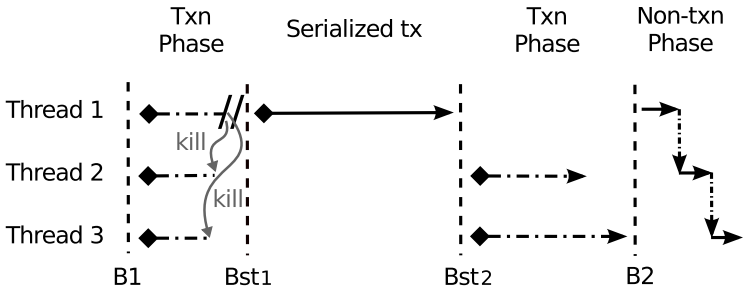
DeTrans-lib guarantees strong determinism by executing non-transactional code serially in round-robin order, and transactions in parallel committing them in also round-robin order. In addition, it wraps the function for transaction serialization implemented in a TM library (Listing 6.3), and ensures that only the transaction executed by the thread that holds the deterministic token (the *token owner*) can invoke a system call (line 3). When the thread gets the token, it kills other running transactions (line 4), so there are no other running transactions to wait for, which was the original implementation and a cause of deadlock.

```
1 int __wrap_serialized()
2 {
3   while (this != token.owner) {} // wait for the token
4   kill(others);                  // kill other transactions
5   return serialized();
6 }
```

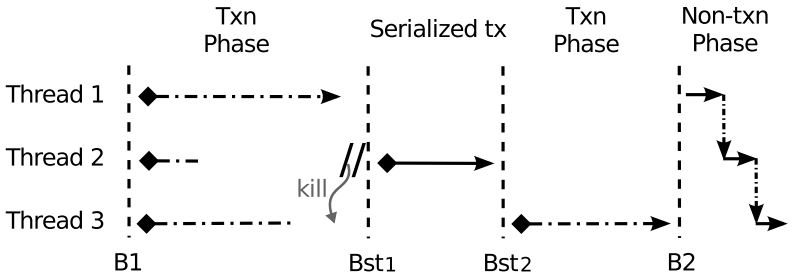
Listing 6.3: Serialization in DeTrans-lib

Figure 6.2 shows examples of threads invoking libc function calls within transactions while running deterministically with DeTrans-lib.

[1]Due to the limitations in compilation of TM-dietlibc, we were not able to simply preload DeTrans while invoking TM-dietlibc. Instead, we had to integrate DeTrans into TM-dietlibc.



(a) The token owner invokes a libc function call.



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

Figure 6.2: DeTrans-lib invoking system calls in deterministic order.

In Figure 6.2(a) the token-owner thread (`Thread1`) kills other transactions due to the libc function call, restarts its transaction as the only running transaction in the application, passes the deterministic token to the next thread and the other threads re-execute their transactions in parallel.

In Figure 6.2(b) one of the threads that is not the token owner (`Thread2`) waits for the deterministic token to be serialized due to the libc function call, kills the only remaining running transaction (executed by `Thread3`) and restarts its transaction. After the serial execution, the remaining transaction can be executed.

`Bst1` and `Bst2` are the barriers that separate a serialized transaction from the Txn 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 libc function. However, since transactions have to be serialized, the order of serialization and libc functions invocation is repeatable and deterministic in every execution of an application.

6.4 Evaluation

We evaluate DeTrans-lib with the benchmarks that call libc I/O functions, using 2 Intel Xeon E5405 processors with 4 cores (8 cores in total). We compiled the benchmarks with GCC[1] 4.9 and linked them against TinySTM [31] 1.0.5. We verified the correctness of the DeTrans-lib implementation by using the Racey [45, 103] stress test. Chapter 3 describes the experimental setup in detail.

6.4.1 Methodology

For evaluation, we use microbenchmarks and TioBench [95], 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. TioBench performs I/O to write data to and read data from a shared file sequentially or randomly (*seq write*, *rnd write*, *seq read*, *rnd read*).

For showing the benefit of TM-dietlibc when the benchmarks run deterministically, we implemented *DeTrans-serial*, the extended DeTrans implementation that supports serialization of transactions (like DeTrans-lib), and serializes all the transactions that invoke standard library functions (unlike DeTrans-lib). This way, we can show the performance improvement in DeTrans-lib that comes from concurrent execution of standard library functions in TM-dietlibc.

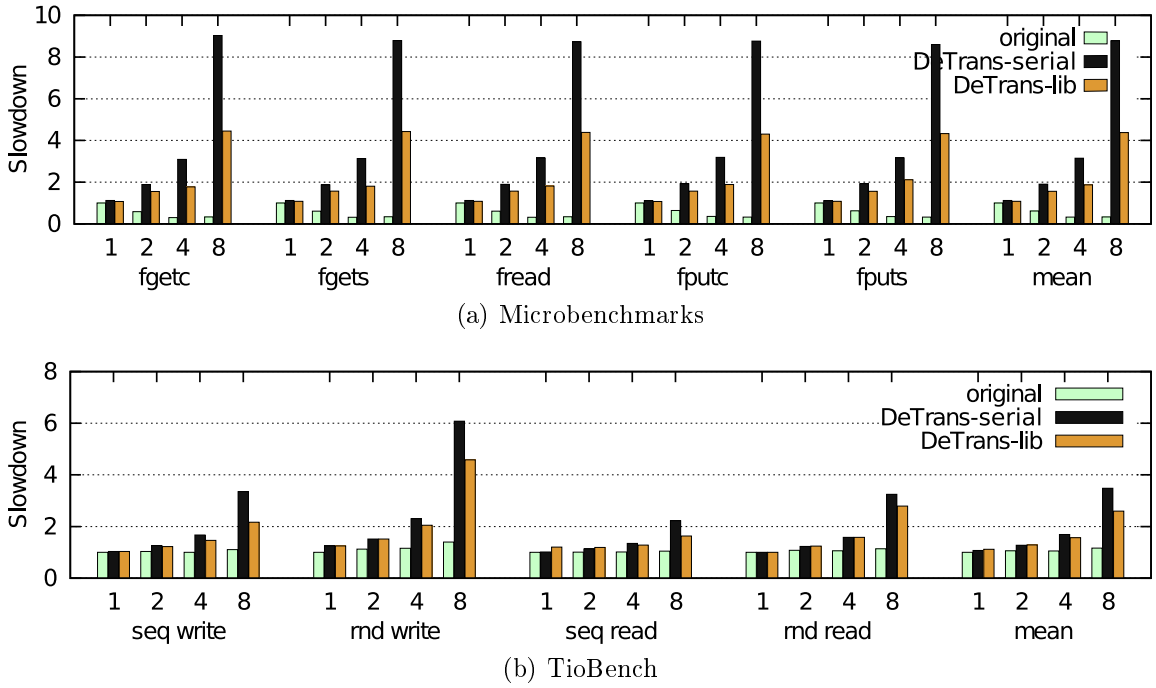
We ran the benchmarks multiple times with 1, 2, 4, and 8 threads, and calculated the geometric mean of the slowdown of deterministic executions (*DeTrans-serial* and *DeTrans-lib*) in comparison to the original (nondeterministic) single-threaded execution invoking TM-dietlibc (*original*).

6.4.2 Results

Figure 6.3 shows the performance of the microbenchmarks and TioBench.

DeTrans-serial and DeTrans-lib behave similarly and have the same source of overhead as DeTrans – waiting on barriers for the deterministic token.

[1] https://gcc.gnu.org/



(a) Microbenchmarks

(b) TioBench

Figure 6.3: The slowdown of deterministic execution of the benchmarks.

Since the microbenchmarks spend most of the execution time out of transactions, and that is the part that DeTrans-lib (as well as DeTrans-serial) executes serially (the non-transactional phase in the double-barrier technique), on average DeTrans-lib slows down the nondeterministic single-threaded execution by 1.12x, 1.62x, 1.94x, and 4.53x for 1, 2, 4, and 8 threads, respectively. However, DeTrans-lib reduces the number of serialized transactions by 99.95% for all the threads and speeds up DeTrans-serial by 1.21x, 1.68x and 2.01x for 2, 4, and 8 threads, respectively.

On the other hand, TioBench spends most of the execution time inside transactions, and that is the part that DeTrans-lib executes in parallel (the transactional phase in the double-barrier technique), DeTrans-lib has low overhead and on average slows down the nondeterministic single-threaded execution by 0.99x, 1.14x, 1.39x, and 2.29x for 1, 2, 4, and 8 threads, respectively. The number of serialized transactions is reduced in Detrans-lib by 41.97%, 51.38%, 56.09%, and 58.44% for 1, 2, 4, and 8 threads, respectively. The average speedup of DeTrans-lib in comparison to DeTrans-serial is 1.08x and 1.34x for 4 and 8 threads, respectively.

In summary of the evaluation, the overhead of the DeTrans-lib depends on the benchmark implementation. First, if threads perform time-intensive operations outside transactions, which is the code that the extensions execute serially, then the overhead of deterministic execution is high. Second, if a benchmark spends most of the execution time within transactions, which is the code that the extensions execute in parallel, then the performance is closer to the performance of the original execution.

6.5 Summary

In this chapter, we presented DeTrans-lib – the first libc that provides deterministic execution of transactional applications at application and standard-library level. Since libc functions invoke system calls occasionally and transactions that invoke system calls have to be serialized, DeTrans-lib reduces the number of serialized transactions and prevents deadlocks caused by busy-waiting in serialization (enforced by the TM library) and busy-waiting in deterministic execution (enforced by DeTrans), and ensures deterministic order of libc functions calls.

7

Support for Ad Hoc Synchronization in Deterministic Execution

7.1 Introduction

In the previous chapters we extended the applicability of deterministic multithreading by providing support for transactions, standard library functions and system calls. However, deterministic multithreading can be applied only for the applications that use explicit synchronization operations (lock/unlock, condition wait/broadcast/signal, barrier wait, and transaction start/commit). If an application, or an external library loaded by the application, accesses shared memory directly to synchronize threads, this is called *ad hoc synchronization*.

Programmers implement ad hoc synchronization as loops (*sync loops*) with busy waiting on shared variables (*sync variables*) and use them to ensure the order of threads execution and the order of their accesses to shared memory. Sync loops are hard to distinguish from computation loops, and researchers have proposed static and dynamic

techniques for their detection [48, 91, 102].

Ad hoc synchronization can be used in applications (e.g. in TioBench, where we avoided ad hoc synchronization by replacing it with a standard barrier, see Section 3.3) or in external libraries loaded by applications (e.g. in the STM library, where we handled ad hoc synchronization by providing support for serialization of transactions, see Section 6.2). In both cases applications might be deadlock prone when running deterministically; therefore, ad hoc synchronization detection should be integrated into the systems for deterministic multithreading.

In this chapter we propose *DeTrans-adhoc*, a standard library that ensures deterministic multithreading in transactional applications and provides support for ad hoc synchronization. DeTran-adhoc identifies sync loops and changes the order of threads execution to guarantee their progress.

The rest of this chapter is organized as follows. In Section 7.2 we illustrate the behaviour of an application with ad hoc synchronization running deterministically and discuss our motivation for implementing a deterministic system with support for ad hoc synchronization. In Section 7.3 we explain the design of DeTrans-adhoc. We use hardware performance counters (registers in the modern processors) to detect loops that contain few instructions and iterate many times, which are usually sync loops. When DeTrans-adhoc detects a sync loop, it changes the order of threads execution, so that another thread can update the sync variable needed to exit the sync loop.

In Section 7.4, we evaluate DeTrans-adhoc with TioBench [95] and Fluidanimate from the PARSEC [12] benchmark suite and show that DeTrans-adhoc detects sync loops successfully, and with low runtime overhead in case of occasional synchronization of threads. We conclude this chapter in Section 7.5.

7.2 Motivation

Figure 7.1 shows a common example of ad hoc synchronization used in applications. The shared variable `start` synchronizes two threads, so that the initialization of `counter` in `foo2()` happens before its incrementation in `foo1()`, and that the threads continue with their execution in parallel.

However, when this code is running deterministically, a deadlock might occur as a consequence of two characteristics of deterministic multithreading.

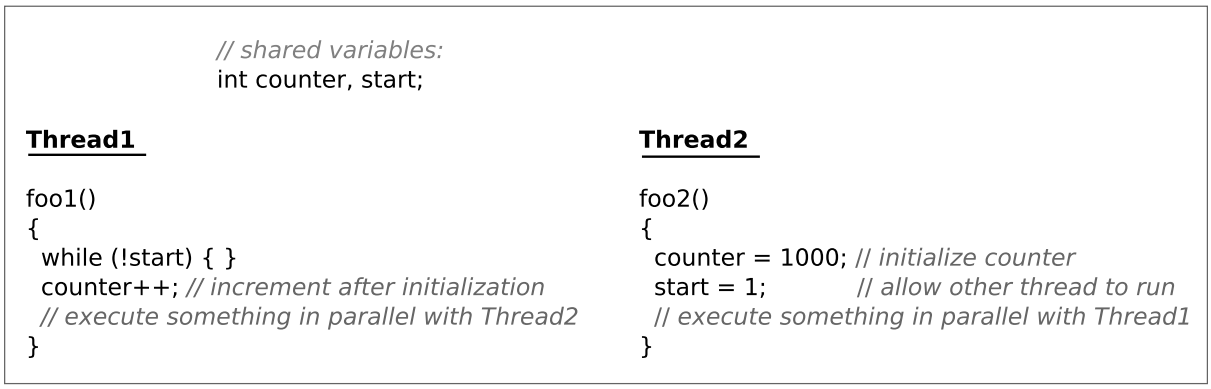


Figure 7.1: A common example of ad hoc synchronization.

First, some of the systems for deterministic multithreading (e.g. DeTrans, see Chapter 4) serialize the code that programmers write as parallel. In serialized execution, threads are executed in statically-defined order (e.g. round-robin) and one at a time. This means that `Thread2` can start the execution of `foo2()` only after `Thread1` finishes the execution of `foo1()`, which does not happen due to the busy-waiting loop in `foo1()`. As a result, the program never finishes its execution.

Second, some of the systems for deterministic multithreading (e.g. Dthreads [54]) postpone writes to shared memory by working on local copies first, and then updating shared memory at synchronization operations in round-robin order. If our example is executed deterministically and threads postpone updates to shared memory until they exit, then `Thread1` waits for `start` to be updated, `Thread2` updates only its local copy of `start` and waits for its turn to update shared memory, and the program also never finishes its execution.

To prevent this, we propose support for ad hoc synchronization in a system for deterministic multithreading.

7.3 Design

In this section we introduce *DeTrans-adhoc*, a standard library that provides deterministic execution of transactional applications and supports ad hoc synchronization in the code of applications and external libraries. DeTrans-adhoc identifies sync loops at runtime and changes the order of threads execution to guarantee their progress.

DeTrans-adhoc is extended DeTrans-lib (see Chapter 6 for details), and its implementation is completely in the standard library, as modified `pthread_*` functions and additional functions that are called in `ITM_*` functions in the STM library. It maintains two queues to ensure determinism. `ready_queue` keeps track of threads that are ready for execution (in the non-transactional part of the execution, where threads are executed serially), and `txn_queue` keeps track of threads that are executing their transactions in parallel. The global deterministic token is passed in the round-robin order in `ready_queue` to guarantee order of threads execution, and in `txn_queue` to guarantee the order of commits of transactions.

7.3.1 Detecting Sync Loops

Programmers implement loops to perform computations (*computational loops*) or to synchronize threads. A sync loop has the characteristics of a *tight loop*, since it contains only a few instructions that are executed in many iterations. Therefore, DeTrans-adhoc uses performance events to count the number of instructions, to detect sync loops and to distinguish them from computational loops.

DeTrans-adhoc creates two hardware performance events for each thread: *Instructions Retired* and *Branch Instructions Retired*. Two hardware performance counters – which are the registers in the modern processors – count these events. The former counts the number of instructions at retirement, and the latter counts the number of branches at retirement, which is needed because a loop consists of a branch instruction. The events are created when threads are created – in the initialization of the program for the main thread and in `pthread_create` for worker threads.

After every N[1] retired instructions, the counter overflows and triggers the signal handler in DeTrans-adhoc. The handler reads the values of the counters and calculates their ratio:

$$ratio = \frac{number\_of\_retired\_instructions}{number\_of\_retired\_branches}$$

If the ratio is less or equal to the previously defined threshold, it means that the thread executes a branch with only a few instructions, and this branch is identified as a

[1] N is a statically defined constant.

sync loop. Therefore, the thread passes the deterministic token to the next ready thread and waits to get the token back to continue its execution. When the token is returned, the thread resets the counters and returns to its execution of the loop where the counter of instructions was overflowed. If the condition to exit the loop is still not fulfilled (i.e. if the sync variable has not been set yet), the thread continues with the execution of the sync loop until the next overflow. The whole process (sync loop identification and token passing) might be repeated multiple times until another thread sets the sync variable, which causes the exit of the sync loop.

With support for ad hoc synchronization, the threads from the example in Figure 7.1 finish their execution successfully since `Thread1` stops its execution of the sync loop, passes the deterministic token to `Thread2`, then `Thread2` updates `start`, exits `foo2()`, and passes the token back to `Thread1`, which also finishes the execution and exits `foo1()`.

7.3.2 Choosing the Right Threshold

The threshold value is a statically-defined constant. If a programmer chooses a too small value, sync loops might not be detected. On the other hand, if a programmer chooses a too large value, some computational loops might be falsely detected as sync loops, which degrades the performance since every time a loop is detected as a sync loop, the thread stops its execution, passes the deterministic token and waits for the next round to get the token back and to continue the execution of the loop.

To help a programmer to choose the threshold, we provide an option to enable recording the ratios in the application loops at runtime while the ad hoc support is disabled. The output of the application is a list of the ratios, and usually the last one is the one from the sync loop, where program hangs when running deterministically without support for ad hoc synchronization.

In order to run an application deterministically, a programmer has to link the application against DeTrans-adhoc, which was previously build with the chosen threshold and the ad hoc synchronization support enabled.

This approach can be automated by integrating the dynamic analysis that calculates ratios into DeTrans-lib.

7.3.3 Making Timing Functions Deterministic

To allow one or more threads to perform some work in a given time interval, programmers use a function that returns a value that depends on the current time (`gettimeofday`) or a function that exits after the given time has passed (`sleep`). These functions are a source of nondeterminism, and they greatly affect debugging of a program since the amount of work that threads execute with and without a debugger might defer significantly, especially if the debugger stops the execution of the program while the program is measuring the time.

To avoid nondeterminism and modifications of programs, we implemented deterministic versions of functions `gettimeofday` and `sleep`.

The first time a program calls `gettimeofday` while running deterministically with DeTrans-adhoc, the return value is the number of the seconds that passed since the reference date (1970-01-01 00:00:00 +0000 UTC), which is 0. Every next call of the function returns the number of the seconds increased by a constant value. As a result, the number of calls of this function and its result are deterministic and remain the same in every execution of the program.

We implemented the deterministic version of `sleep` as a computational loop that iterates multiple time while performing several mathematical operations in every iteration. If the function is called from a sync loop, the computational loop increases the number of instructions at retirement and might affect the sync loop detection. To prevent this, the performance events are disabled while performing `sleep`. A simplified alternative is to have the `sleep` function with an empty body.

7.3.4 Detecting Deadlocks in Sync Loops

If a program has a deadlock where all running threads busy wait on a sync variable or multiple sync variables and cannot make any progress in their execution, the program running deterministically with DeTrans-adhoc cannot avoid this deadlock, and the program behaves in the same way as running nondeterministically – it hangs.

To identify this problem, DeTrans-adhoc maintains a queue in the signal handler – `overflow_queue`. When the counter overflows, the handler saves thread's id in `overflow_queue`, and removes it when the thread exits the handler. If all the threads from `ready_queue` are also in `overflow_queue`, and `txn_queue` is empty, it means that

all ready threads are in the handler, and none of them is making progress in the execution. In this case, DeTrans-adhoc prints the order of threads in `overflow_queue`, which helps a programmer to debug the program and to know in which order the sync variables should be provided to solve the deadlock.

7.4 Evaluation

We evaluate DeTrans-adhoc by running TioBench [95] and Fluidanimate from PARSEC [12] on an Intel Xeon E3-1220 processor with 4 cores[1]. We compiled the benchmarks with GCC 4.9 and linked them against TinySTM [31] 1.0.5. All the details about the experimental setup are in Chapter 3.

7.4.1 Methodology

The first benchmark we use in evaluation is TioBench, where two sync loops guarantee that worker threads are synchronized with the main thread before they start performing I/O operations (Figure 7.2). In the first loop, the main thread waits for worker threads (e.g. `Thread1`) to be initialized. In the second loop worker threads wait for the main thread to start executing the I/O operations.

After the synchronization, worker threads write or read data sequentially (*seq write*, *seq read*), or randomly (*rnd write*, *rnd read*) inside transactions.

Apart from replacing the system calls with standard library function calls (in order to evaluate the standard libraries), this benchmark does not require any modifications to run deterministically with DeTrans-adhoc. As shown in the previous section, the timing functions called from the benchmark (`gettimefoday` and `sleep`) are deterministic with DeTrans-adhoc.

When running deterministically, the main thread detects the first sync loop and passes the deterministic token to the worker thread. The worker thread initializes `child_status`, detects its sync loop and passes the token back. The main thread exits the sync loop and sets `start`, which is necessary for the worker thread to start performing I/O operations.

[1]Note that for the performance counters we need a modern architecture; therefore, the machine is different from the one used in the previous chapters.

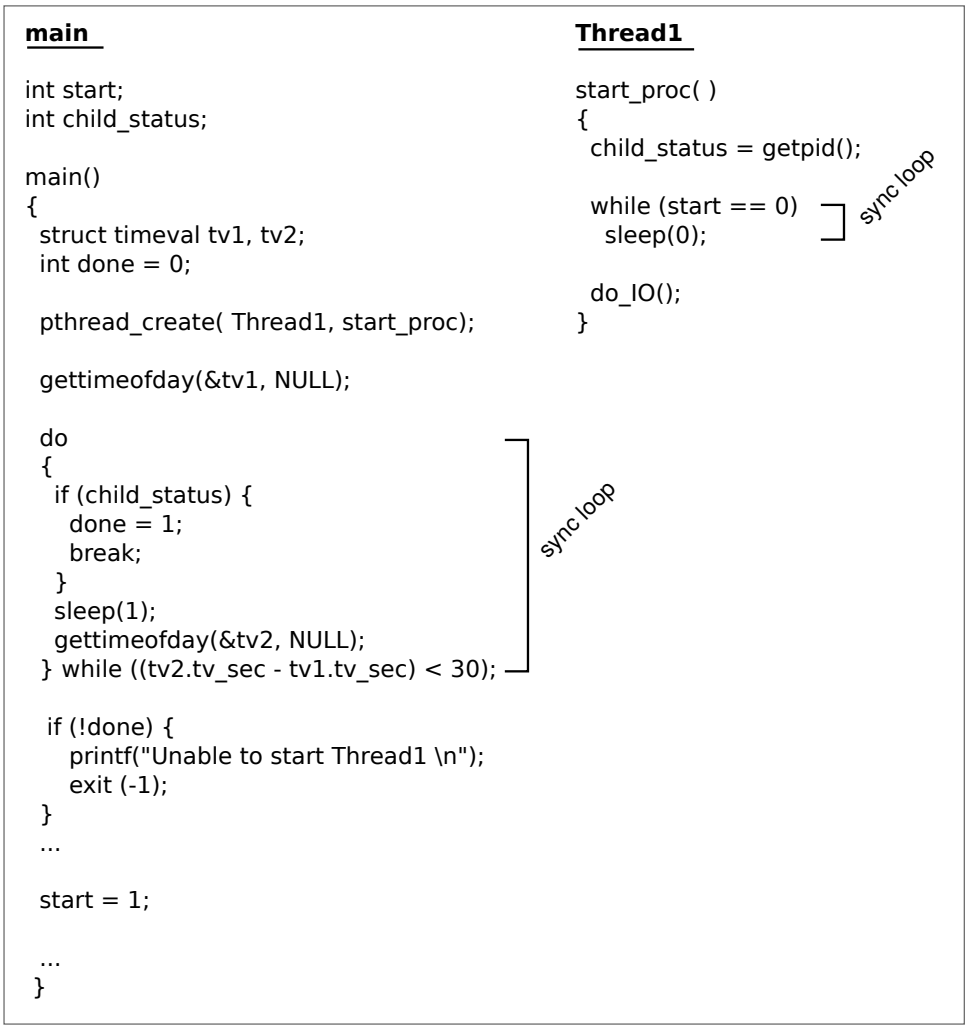


Figure 7.2: Ad hoc sync loops in TioBench.

The second benchmark is Fluidanimate, which simulates an incompressible fluid for interactive animation purposes. Phases of the simulations are separated by ten barriers, so that worker threads are always in the same phase. Furthermore, each barrier has two sub-barriers: one to ensure that worker threads have left the previous barrier, and another to keep the threads until the last one arrives.

The original implementation of the benchmark is lock-based, which was not possible to run with other systems for deterministic multithreading (Dthread [54] and Consequence [64]) due to ad hoc synchronization and the limitations of these systems.

We use the TM-based version of the benchmark where the barriers are implemented

by using condition variables suitable with TM [29]. Since transactions enclose accesses to all shared variables, this version of the benchmark does not have ad hoc synchronization.

To evaluate the DeTrans-adhoc implementation, we added one more version of the barrier by using compare-and-swap instructions to access the current number of threads at a barrier and by implementing a busy-waiting loop to wait until the last thread arrives. This way, only a deterministic system with the ad hoc synchronization support can execute this benchmark.

We executed TioBench and Fluidanimate with the threshold for the ratio that was statically defined and remained constant during execution of the benchmarks. The number of retired instructions when the performance counter overflows was 100000.

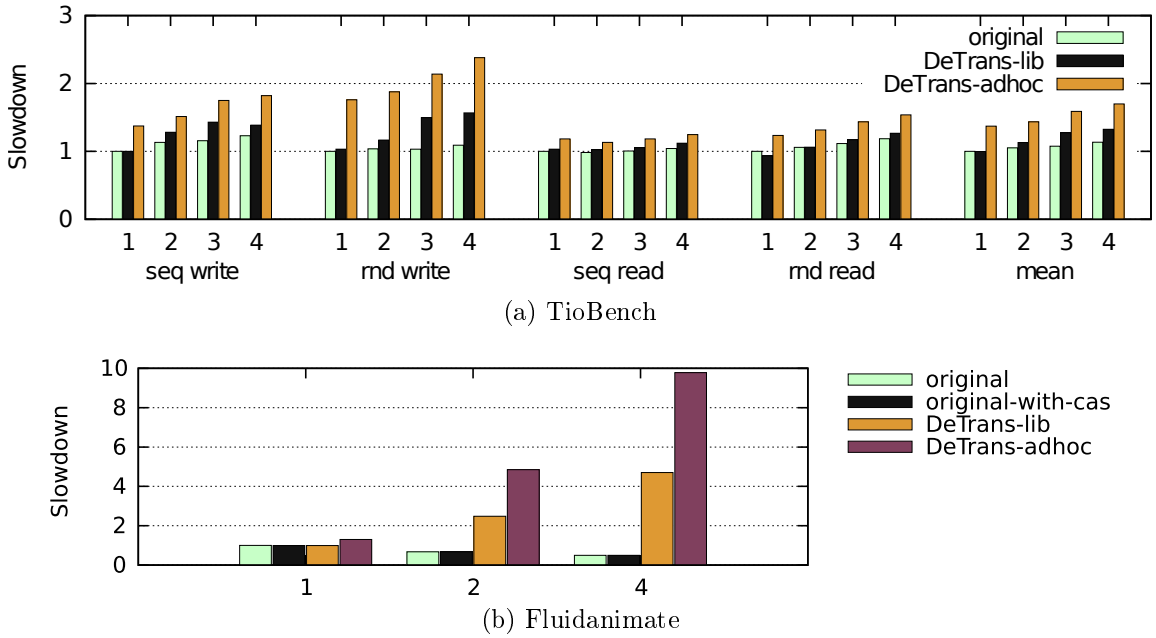
We ran TioBench multiple times with 1, 2, 3, and 4 threads, where each thread read or wrote 128 characters per transaction. We ran Fluidanimate with 1, 2, and 4 threads (due to the restriction that the number of threads has to be a power of 2), and with the maximum input set. We calculated the geometric mean of the slowdown of deterministic executions in comparison to the original (nondeterministic) single-threaded execution.

7.4.2 Results

In Figure 7.3(a), we compare the overhead in TioBench's execution running nondeterministically and invoking TM-dietlibc (*original*), deterministically with DeTrans-lib where we replaced the ad hoc synchronization with a `pthread_barrier`, and deterministically with DeTrans-adhoc. The average overhead of DeTrans-adhoc in comparison to DeTrans-lib is 37.29%, 27.01%, 24.52%, 28.18% for 1, 2, 3, and 4 threads, respectively.

In Figure 7.3(b) we show the slowdown of the benchmark running deterministically with DeTrans-lib (when the barrier implementation is the original TM-based implementation) and with DeTrans-adhoc (when the barrier implementation is with compare-and-swap) in comparison to the original (non-deterministic) single-threaded execution of the benchmark (*original*). *original-with-cas* presents the non-deterministic execution of the benchmark with the barrier implemented by compare-and-swap instructions and busy waiting loops.

First, *original* and *original-with-cas* perform similarly (the execution times varies in less than 1% for the same number of threads), which means that we did not improve nor worsen performance by changing the implementation of the barrier. Second, for four



(a) TioBench

(b) Fluidanimate

Figure 7.3: The slowdown of the benchmarks running deterministically with DeTrans-adhoc.

running threads DeTrans-lib slows down the single-threaded execution of the benchmark by 4.71x and DeTrans-adhoc by 9.78x.

The high overhead in DeTrans-adhoc is a result of the intensive synchronization among threads in Fluidanimate (unlike TioBench). Worker threads perform a workload in transactions between the barriers, and the number of transactions varies in different threads. Therefore, while one thread is waiting at a barrier, i.e. busy waiting in the sync loop, other threads might still perform transactions. This means that the thread that arrives first in the sync loop passes the token and gets the token back as long as other threads are executing transactions. E.g. the token is passed 636 times for 2 worker threads, which can be reduced by better distributions of work (i.e. transactions) among threads.

7.5 Summary

In this chapter we presented DeTrans-adhoc, the standard library that provides deterministic execution of transactional applications and supports ad hoc synchronization. It uses hardware performance counters to count the retired instructions and retired branches. De-

pending on their ratio, DeTrans-adhoc might dynamically, but deterministically, change the order of threads execution, which prevents an application from hanging in a busy-waiting loop used for ad hoc synchronization. The additional runtime overhead is low in case of a benchmark with poor synchronization among threads, and significant in the benchmarks with intensive synchronization and frequent changes of the order of threads execution.

8

Related work

In this section we describe work related to this dissertation regarding deterministic multithreading and other techniques for testing and debugging multithreaded applications, the techniques for ordering transactions, ad hoc synchronization detection, and integration of TM with standard library and system calls, as well as with any other software.

Deterministic multithreading

Deterministic multithreading has been a popular research area in the last several years. Researchers have proposed deterministic systems for programs with data races (*strong determinism*) [9, 11, 23, 24, 54, 56, 63], or more relaxed deterministic systems for data-race-free programs (*weak determinism*) [19, 72, 78]. Furthermore, according to [86], the order of threads execution can be defined statically (the serial and the round-robin order) [11, 19, 54, 63] or dynamically (the order depending on the logical clock) [24, 56, 72], or as a combination of both [9, 23].

Olszewski et al. [72] propose Kendo, a software implementation for weak determinism of lock-based applications. In Kendo, each thread maintains its logical clock that counts logical time. Events like lock acquisition and spinning increase logical time. A thread acquires a lock only if it is available in both logical and physical time, which is sufficient for weak determinism.

Devietti et al. [23] and Bergan et al. [9] propose several software-only, hardware-only and software-hardware implementations to ensure strong determinism. The basic implementation is deterministic serialization of parallel applications: execution of threads is divided into quanta and threads execute their quanta in round-robin order. Parallelism is improved by (i) using a shared memory ownership table (strong determinism); (ii) maintaining thread-local store buffers (strong determinism), or (iii) tracking happens-before dependencies of lock acquisitions (weak determinism). Although they execute quanta atomically in transactions and commit them in order, they do not use any state-of-the-art STM library, evaluation with transactional applications is not provided. Furthermore, the optimization the authors suggest – to allow transactions to see uncommitted updates of memory – is against the basic TM properties; therefore, it cannot be applied for transactional applications.

BulkCompactor (Duan et al. [28]) is a deterministic system implemented in hardware with the improved order of chunk (quantum) execution in comparison to the traditional round-robin order. The chunks that have to be re-executed due to conflicts are skipped, and they are re-executed in the next round of token passing. This way, BulkCompactor reduces the time spent on chunks waiting for an aborted chunk to be re-executed. In our implementation, re-execution of transactions in another round would increase the number of barriers and rounds in the execution, and would negatively affect the benefit of reduced waiting time.

Another modification of the round-robin order is implemented in Parrot (Cui et al. [19]), where the threads interleaving depends on developers' annotations of potential bottlenecks, which makes deterministic execution more efficient.

Grace (Berger et al. [11]) and Dthreads (Liu et al. [54]) 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. As a consequence of duplicating memory pages, applications running with Grace or Dthreads have high memory consumption.

Combining the ideas from Dthreads [54] and Kendo [72], Lu et al. [56] implemented RFDet, a deterministic system where: (i) the order of synchronization operations is deterministic and based on logical time that each thread measures by measuring the number of synchronization operations, (ii) each thread works on a local copy of memory, and (iii) at a lock acquisition, a thread gets modifications from other threads that happen before in logical time. Similarly to Dthreads, RFDet is efficient according to the execution time of benchmarks (because threads do not waste time on barriers), but inefficient according to memory consumption.

When Dthreads uses Conversion (Merrifield et al. [63]) as the underlying memory model, it can omit the barriers where threads wait to commit their changes. Therefore, it outperforms the original Dthreads implementation. Further optimizations are introduced in Consequence (Merrifield et al. [64]), a system for deterministic multithreading that uses private memory pages like in Conversion and guarantees deterministic order of threads execution by executing chunks according to the number of retired instructions in the chunks.

The systems discussed above are suitable for lock-based applications. However, they violate atomicity and isolation of transactions, and as a consequence, execute transactional applications incorrectly (see 4.3 for details).

The recent work by Ravichandran et al. (DeSTM [78]) ensures weak determinism in transactional applications. DeSTM relaxes the barriers in the double-barrier technique and allows some transactions to pass the barriers earlier than others, i.e. to start their execution earlier and to start with the commit earlier. Unlike DeSTM, we guarantee strong determinism in DeTrans, and also analyse and compare overheads in deterministic multithreading provided by DeTrans and one of the state-of-the-art systems for deterministic multithreading in lock-based applications.

Other techniques for testing and debugging

Apart from deterministic multithreading, Record and Replay (R&R) is another technique useful for testing and debugging. R&R systems [53, 67, 70, 94] save a log of memory interleavings in a program's execution and then re-execute the same program according to the saved log; therefore, they induce overhead in execution time and in memory.

A R&R approach is used by Gottschlich et al. [34] to help a programmer to debug

transactional applications running with HTM on Intel and IBM processors. The R&R system they propose records information about transactions (transaction starts, commits and aborts) to be able to reproduce the order of transactions, i.e., the order of shared memory accesses. The system is based on a hardware mechanism that divides execution into chunks depending on the number of retired instructions.

Zyulkyarov et al. [107] extended a C# debugger to provide facilities for debugging STM applications. This approach allows a programmer to control the interleaving of transactions at debug time, and it can be combined with DeTrans, where an application runs deterministically until the bug appears, and after that, the programmer could inspect and change the application state in order to fix the bug.

Harmanci et al. [38, 39] present a framework for testing, evaluating and comparing TM implementations. The tests from this framework enforce the order of memory accesses that might produce unwanted behaviour in the execution. While the authors focus on testing TM implementations, our goal with DeTrans is to provide easier testing, debugging, and fault tolerance of transactional applications.

Kestor et al. [50] detect data races in transactional applications. The data race detection tool instruments memory operations performed from transactional and non-transactional code. Although they found data races in a few benchmarks, the instrumentation overhead slows down the STAMP benchmarks from 5x to 85x for 8 threads runs. In comparison to this, DeTrans ensures that a program always behaves the same in the presence of a data race with less runtime overhead.

Deterministic and redundant multithreading for standard library and system calls

From the systems for deterministic multithreading 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 transactional applications due to differences in synchronization mechanisms, as mentioned above.

Dobel et al. [27] 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. The runtime overhead is reduced by using a standard library that was modified to be aware of redundant threads. However. RomainMT improves fault tolerance 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 result in different outputs.

Deterministic operating system Determinator [7] enforces determinism at application and OS level by creating private copies of shared objects that threads obtain at a fork and merging the copies back to the global object at a join. On the other hand, DeTrans-lib is implemented at user level, ensures application and libc-level determinism and guarantees deterministic order of invoking system calls.

Ordering Transactions

Shpeisman et al. [89] characterize data races that are caused by weak isolation in STMs and propose read and write memory barriers to enforce strong isolation in Java transactional applications. DeTrans could benefit from strong determinism provided by a STM implementation since it would allow transactional and non-transactional code to run in parallel while still guaranteeing strong determinism.

Ordering of transactions can be also used to parallelize events in event processing [15], to parallelize sequential applications [99], or to meat deadlines in reactive transactional applications [59]. However, these systems do not guarantee repeatability in execution of applications.

Detection of ad hoc and infinite loops

Tian et al. [92] propose detection of ad hoc synchronization loops (*sync loops*) and their corresponding writes to memory (*sync writes*) by either instrumenting code in software or using the cache coherence protocol in hardware. Furthermore, they provide an extension for conflict resolution in TM that is used for runtime monitoring of applications. With the extension, TM ensures that a transaction with the sync update commits its changes before the transaction with the sync loop. On the other hand, DeTrans-adhoc detects sync loops efficiently without code instrumentation of hardware modifications.

SyncFinder [102] is a tool that statically analyses multi-threaded C/C++ programs to

detect ad hoc synchronization. It identifies sync loops and sync writes and annotates them to help a programmer to replace ad hoc synchronization with structures and functions for synchronization (e.g. using a standard POSIX thread library). Unlike SyncFinder, DeTrans-adhoc provides successful execution of applications with ad hoc synchronization without programmers removing ad hoc synchronization.

The Bolt detector [51] is a tool that detects infinitive loops. Bolt gets attached to a running application to detect if the application is inside of a loop. It takes and records a snapshot of the register and memory state at the end of each loop iteration, then compares the snapshot with the previous snapshots, and detects an infinite loop in case the there is a match between the snapshots.

Jannesari et al. [48] and Tian et al. [91] propose techniques to identify ad hoc synchronization in a running application and to improve data-race detectors by excluding sync loops from the data-race detection.

Importantly for deterministic multithreading, ad hoc synhronization is a limitation of many systems for deterministic multithreading mentioned above [19, 54, 56]. However, Consequence [64] limits the number of retired instructions in a chunk and similarly to our approach, postpones execution of the chunks with sync loops. Consequence has very low overhead in general, but limiting the number of instructions slows down execution significantly.

Libc and system calls in transactions

TM-dietlibc is the first standard C system library with transactional semantics. Different proposals handle I/O and other system calls within transactions, but all of them require some changes of the software that use these features.

Volos et al. [97] provide system call execution within transactions by implementing wrappers for system calls and acquiring a lock before accessing a kernel resource, which hurts parallelism. Demsky et al. [22] provide a Java library with an API extended with several functions, which ensures that file changes remain local until commit time.

Porter et al. [76] implement transactions on the operating system level, which requires invocation of specific system calls in the user transaction.

On the other hand, in TM-dietlibc we keep the standard C API so an application developer does not have to modify application code when invoking a standard library.

Transactification of software

To be able to compile and execute code within transactions, in related work [8, 25, 97] the authors suggest that some critical actions should be deferred until commit time. However, for nested transactions it can be a pitfall causing certain side effects. In flat nesting, all nested transactions are combined into a single one; therefore, deferred actions will be executed after the outer-most transaction commits. The problem occurs when the result of the deferred operation is needed inside the outer one. Only actions with no influence on the rest of the program's execution can be postponed, e.g. memory deallocation.

Regarding the interaction of locks and transactions, there are proposals to decide dynamically whether a critical section should be transactional or lock-based by Usui et al. [93] or for a new type of lock (*transaction-safe*) by Volos et al. [96]. Similar to that, Rossbach et al. [82] introduce *cooperative transactional locks*. Gottschlich and Chung [33] describe how to statically encode the conflicts between locks and transaction. In contrast, we take the path of full static separation, which is the safest approach for any TM implementation.

In TM-dietlibc we presented the novel technique that detects kernel space conflicts at the user level and compensates their side effects. No other related work mentions the possibility of such conflicts to remain undetected, although some of them propose compensation and deferral actions [8, 62] or irrevocable execution [90, 101] for handling system calls and I/O inside transactions.

Ruan et al.[84] provide a transactified version of the Memcached benchmark to analyse and evaluate the features of the Draft C++ TM Specification [5]. To be able to invoke standard library functions in transactions, they propose several techniques: (i) reimplementing the functions to be available for compiler instrumentation, (ii) declaring them as pure and wrapping them so that conflict can still be detected in the wrapper, or (iii) postponing them until commit time when they are executed out of transactions. We implemented similar techniques in TM-dietlibc, so that the benchmarks invoking TM-dietlibc can remain unmodified.

Pankratius et al. [73], by analysing their students' work on a lock-based and a TM-based search engine development, concluded that TM needs better support for I/O operations since running transactions serially limits concurrency and scalability.

9 Conclusions

In this thesis we proposed techniques to extend the applicability of deterministic multithreading by supporting transactions, standard library calls, system calls, and ad hoc synchronization, which allows a programmer to apply deterministic multithreading on applications with various concurrency mechanisms and at both application and standard-library level. Furthermore, we extend the applicability of transactional applications by allowing threads to invoke standard library functions inside transactions and to execute them concurrently.

In Chapter 4 we presented DeTrans, a runtime library that ensures deterministic execution of multithreaded transactional applications. DeTrans provides strong determinism, meaning that even in the presence of a data race an application produces the same output if it runs with the same input parameters. DeTrans achieves this by executing non-transactional code serially in round-robin order, and transactions in parallel and committing them in the same round-robin order. It relies on TM implemented in software to preserve memory consistency and ensure correct parallel execution of transactional code. DeTrans can reduce the time a programmer spends in developing, testing

and debugging a transactional application since it guarantees a single order of threads execution whenever the application runs with the same input parameters.

To increase concurrency in transactional applications that invoke standard library functions inside transactions, in Chapter 5 we presented TM-dietlibc, the first TM-aware standard library. TM-dietlibc is based on an originally lock-based library, but modified to support transactions and to be suitable with various TM compilers and TM implementations. The experience we gained during TM-dietlibc and TM integration can help software developers to apply similar techniques when they write transactional programs from scratch or modify existing lock-based programs to use TM as a concurrency control mechanism. TM-dietlibc extends the applicability of transactional applications and allows transactions to invoke standard library functions without modifications of the application code since the API remained unchanged. In addition, TM-dietlibc serializes only the transactions that invoke system calls with non-reversible side effects.

By porting deterministic system DeTrans to TM-dietlibc, in Chapter 6 we provided DeTrans-lib, the first TM-aware standard library that ensures deterministic multithreading in transactional applications. DeTrans-lib avoids deadlocks caused by busy-waiting in serialization (enforced by TM due to system calls) and busy-waiting in deterministic execution (enforced by DeTrans). With DeTrans-lib, threads deterministically execute standard library calls and invoke system calls.

Finally, in Chapter 7, we presented DeTrans-adhoc, the extended DeTrans-lib implementation that supports ad hoc synchronization in transactional applications while running deterministically. DeTrans-adhoc relies on hardware performance counters to identify synchronization loops at runtime and to avoid deadlocks by dynamically and deterministically changing the order of threads execution. It detects ad hoc synchronization in the code of applications and external libraries.

9.1 Future Work

Even though the TM community has provided TM support in compilers and efficient TM implementations in hardware and software, the number of transactional applications and transactional standard libraries is still negligible. The experience we gained during the transactification of a standard library can help developers to write any transactional software or to transactify an existing one.

On the other hand, deterministic multithreading should be further investigated and optimized to be widely used for testing and debugging. Systems for deterministic multithreading should be with low additional overhead in execution time and memory consumption.

If we modify our deterministic system to employ TM implemented in hardware, it might reduce the runtime overhead since strong isolation provided by this hardware would allow more code to be executed in parallel and deterministically even in the presence of data races. There are a few use cases of the deterministic system with low overhead.

The first use case is exhaustive testing and benchmarking where the system executes many threads interleavings deterministically, which could be useful for finding bugs in applications. After the exhaustive testing is done, and found bugs are fixed, the applications could be executed nondeterministically.

The second use case is having determinism "by default"(unless required differently), where a programmer tests and debugs only one thread interleaving in an application, and the deterministic system guarantees that the application is (and always will be) executed with this interleaving.

Finally, the deterministic system can be integrated into other tools, like debuggers and data race detectors to help programmers in developing and debugging multithreaded applications.

10 Publications

The contents of this thesis led to the following publications:

Nebojša Miletić, **Vesna Smiljković**, Cristian Perfumo, Tim Harris, Adrián Cristal, Ibrahim Hur, Osman S. Ünsal and Mateo Valero, **Transactification of a Real-world System Library**, In the 5th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 2010).

Vesna Smiljković, Martin Nowack, Nebojša Miletić, Tim Harris, Osman S. Ünsal, Adrián Cristal, and Mateo Valero, **TM-dietlibc: A TM-aware Real-world System Library**, In Proceedings of the 27th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2013).

Vesna Smiljković, Srđan Stipić, Christof Fetzer, Osman S. Ünsal, Adrián Cristal, and Mateo Valero, **DeTrans: Deterministic and Parallel Execution of Transactions**, In Proceedings of the 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2014).

Vesna Smiljković, Osman S. Ünsal, Adrián Cristal, and Mateo Valero, **Determinism at Standard-Library Level in TM-Based Applications**, In Proceedings of the 12th Annual IFIP International Conference on Network and Parallel Computing (NPC 2015). Also appears in the International Journal of Parallel Programming (IJPP 2015) special edition for NPC.

The following publications are related but not included in this thesis:

Vesna Smiljković, Srđan Stipić, Osman S. Ünsal, Adrián Cristal, and Mateo Valero, **Transaction Coalescing - Lowering Transactional Overheads by Merging Transactions**, In the 6th Workshop On Programmability Issues for Heterogeneous Multicores (MULTIPROG 2013).

Srđan Stipić, **Vesna Smiljković**, Adrián Cristal, Osman S. Ünsal, and Mateo Valero, **Profile-Guided Transaction Coalescing - Lowering Transactional Overheads by Merging Transactions**, In Proceedings of the 9th International Conference on High-Performance and Embedded Architectures and Compilation (HiPEAC 2014). Also appears in the ACM Transactions on Architecture and Code Optimization (TACO 2014).

Srđan Stipić, Vasileios Karakostas, **Vesna Smiljković**, Vladimir Gajinov, Adrián Cristal, Osman S. Ünsal, and Mateo Valero, **Dynamic Transaction Coalesing**, In Proceedings of In Proceedings of the 11th Conference on ACM Computing Frontiers (CF 2014).

References

[1] Linux kernel profiling with perf. https://perf.wiki.kernel.org. Cited on page: 42

[2] Martín Abadi, Tim Harris, and Katherine F. Moore. A model of dynamic separation for transactional memory. In *Proceedings of the 19th international conference on Concurrency Theory*, CONCUR '08, pages 6–20. Springer-Verlag, 2008. ISBN 978-3-540-85360-2. Cited on page: 54

[3] Martín Abadi, Andrew Birrell, Tim Harris, Johnson Hsieh, and Michael Isard. Implementation and use of transactional memory with dynamic separation. In *CC '09: Proceedings of the 18th International Conference on Compiler Construction*, pages 63–77, 2009. ISBN 978-3-642-00721-7. Cited on page: 54

[4] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. *ACM Trans. Program. Lang. Syst.*, 33(1):2:1–2:50, January 2011. ISSN 0164-0925. DOI: 10.1145/1889997.1889999. Cited on page: 54

[5] Ali-Reza Adl-Tabatabai, Tatiana Shpeisman, and Justin Gottschlich. Draft specification of transactional language constructs for C++, February 2012. version 1.1. http://justingottschlich.com/tm-specification-for-c-v-1-1/. Cited on page: 10, 17, 29, 101

[6] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages

316–327, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2275-0. DOI: 10.1109/HPCA.2005.41. Cited on page: 2

[7] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association. Cited on page: 99

[8] Lee Baugh and Craig Zilles. An analysis of I/O and syscalls in critical sections and their implications for transactional memory. In *Proceedings of the ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '08, pages 54–62, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-1-4244-2232-6. DOI: 10.1109/ISPASS.2008.4510738. Cited on page: 15, 101

[9] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. 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, pages 53–64, New York, NY, USA, 2010. ACM. Cited on page: 30, 33, 35, 95, 96

[10] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in dos. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association. Cited on page: 30

[11] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe multithreaded programming for C/C++. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 81–96, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. DOI: 10.1145/1640089.1640096. Cited on page: 33, 35, 95, 96

[12] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011. Cited on page: 29, 84, 89

[13] Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), Feb 2006. ISSN 1556-6056. DOI: 10.1109/L-CA.2006.18. Cited on page: 11

[14] Robert L. Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 4–4, Berkeley, CA, USA, 2009. USENIX Association. Cited on page: 20, 35

[15] Andrey Brito, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. Speculative out-of-order event processing with software transaction memory. In *Proceedings of the Second International Conference on Distributed Event-based Systems*, DEBS '08, pages 265–275, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-090-6. DOI: 10.1145/1385989.1386023. Cited on page: 40, 99

[16] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. Robust architectural support for transactional memory in the Power architecture. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 225–236, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2079-5. DOI: 10.1145/2485922.2485942. Cited on page: 2, 16

[17] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Riviere. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 27–40. ACM, 2010. ISBN 978-1-60558-577-2. Cited on page: 17, 24, 29, 50, 55, 56, 59, 68

[18] Intel Corporation. Intel transactional memory compiler and runtime application binary interface. `http://software.intel.com/sites/default/files/m/5/a/2/a/f/8097-Intel_TM_ABI_1_0_1.pdf`, November 2008. Cited on page: 11, 55, 56, 59

[19] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the Twenty-Fourth ACM*

Symposium on Operating Systems Principles, SOSP '13, pages 388–405, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. DOI: 10.1145/2517349.2522735. Cited on page: 30, 95, 96, 100

[20] Luke Dalessandro and Michael L Scott. Strong isolation is a weak idea. In *TRANSACT '09: 4th Workshop on Transactional Computing*, February 2009. Cited on page: 11

[21] Luke Dalessandro, François Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory. *SIGPLAN Not.*, 46(3):39–52, March 2011. ISSN 0362-1340. DOI: 10.1145/1961296.1950373. Cited on page: 2

[22] Brian Demsky and Navid Farri Tehrany. Integrating file operations into transactional memory. *Journal of Parallel and Distributed Computing*, 71:1293–1304, October 2011. DOI: 0.1016/j.jpdc.2011.04.005. Cited on page: 51, 100

[23] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: deterministic shared memory multiprocessing. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 85–96, New York, NY, USA, 2009. ACM. Cited on page: 3, 18, 30, 33, 35, 95, 96

[24] Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. RCDC: a relaxed consistency deterministic computer. *SIGPLAN Not.*, 47(4):67–78, 2011. Cited on page: 33, 35, 95

[25] Ricardo Dias, João M. S. Lourenço, and Gonçalo Cunha. Developing libraries using software transactional memory. *Computer Science and Information Systems*, 5(2): 103–117, 2008. Cited on page: 101

[26] Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 3–14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2809-8. DOI: 10.1145/2628071.2628080. Cited on page: 17

[27] Björn Döbel and Hermann Härtig. Can we put concurrency back into redundant multithreading? In *Proceedings of the 14th International Conference on Embedded Software*, EMSOFT '14, pages 19:1–19:10, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3052-7. DOI: 10.1145/2656045.2656050. Cited on page: 98

[28] Yuelu Duan, Xing Zhou, Wonsun Ahn, and J. Torrellas. BulkCompactor: Optimized deterministic execution via conflict-aware commit of atomic blocks. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12, February 2012. DOI: 10.1109/HPCA.2012.6169040. Cited on page: 96

[29] Polina Dudnik and Michael M. Swift. Condition variables and transactional memory: Problem or opportunity? In *TRANSACT '09: 4th Workshop on Transactional Computing*, February 2009. Cited on page: 29, 91

[30] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 237–246, 2008. Cited on page: 17

[31] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *Parallel and Distributed Systems, IEEE Transactions on*, 21(12):1793–1807, 2010. Cited on page: 2, 14, 17, 24, 40, 41, 55, 68, 80, 89

[32] Keir Fraser. *Practical lock freedom.* PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579. Cited on page: 28, 67

[33] Justin E. Gottschlich and JaeWoong Chung. Optimizing the concurrent execution of locks and transactions. In *Proceedings of the 24th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, September 2011. Cited on page: 101

[34] Justin E. Gottschlich, Rob Knauerhase, and Gilles Pokam. But how do we really debug transactional memory programs. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Parallelism, Berkeley, CA*, 2013. Cited on page: 97

[35] Per Hammarlund, Alberto J. Martinez, Atiq A. Bajwa, David L. Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, Randy B. Osborne, Ravi Rajwar, Ronak Singhal, Reynold D'Sa, Robert Chappell, Shiv Kaushik, Srinivas Chennupaty, Stephan Jourdan, Steve Gunther, Tom Piazza, and Ted Burton. Haswell: The fourth-generation intel core processor. *Micro, IEEE*, 34(2):6–20, March 2014. ISSN 0272-1732. DOI: 10.1109/MM.2014.10. Cited on page: 2, 16

[36] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. *SIGARCH Comput. Archit. News*, 32(2):102–, March 2004. ISSN 0163-5964. DOI: 10.1145/1028176.1006711. Cited on page: 2

[37] Ruud Haring, Martin Ohmacht, Thomas Fox, Michael Gschwind, David Satterfield, Krishnan Sugavanam, Paul Coteus, Philip Heidelberger, Matthias Blumrich, Robert Wisniewski, alan gara, George Chiu, Peter Boyle, Norman Chist, and Changhoan Kim. The IBM Blue Gene/Q compute chip. *IEEE Micro*, 32(2):48–60, March 2012. ISSN 0272-1732. DOI: 10.1109/MM.2011.108. Cited on page: 16

[38] Derin Harmanci, Pascal Felber, Vincent Gramoli, and Christof Fetzer. TMUNIT: Testing software transactional memories. In *In: The 4th ACM SIGPLAN Workshop on Transactional Computing*. Citeseer, 2009. Cited on page: 98

[39] Derin Harmanci, Vincent Gramoli, Pascal Felber, and Christof Fetzer. Extensible transactional memory testbed. *J. Parallel Distrib. Comput.*, 70(10):1053–1067, October 2010. ISSN 0743-7315. DOI: 10.1016/j.jpdc.2010.02.008. Cited on page: 98

[40] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 388–402, New York, NY, USA, 2003. ACM. ISBN 1-58113-712-5. DOI: 10.1145/949305.949340. Cited on page: 14

[41] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60. ACM, 2005. Cited on page: 2

[42] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2nd edition, June 2010. ISBN 978-1598291247. Cited on page: 2, 9, 14

[43] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM. ISBN 0-8186-3810-9. DOI: 10.1145/165123.165164. Cited on page: 2, 9, 16

[44] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM. ISBN 1-58113-708-7. DOI: 10.1145/872035.872048. Cited on page: 14

[45] Mark D. Hill and Min Xu. Racey: A stress test for deterministic execution. `http://www.cs.wisc.edu/~markhill/racey.html`. Cited on page: 30, 34, 41, 74, 80

[46] Sungpack Hong, Tayo Oguntebi, Jared Casper, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. Eigenbench: A simple exploration tool for orthogonal tm characteristics. In *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-9297-8. DOI: 10.1109/IISWC.2010.5648812. Cited on page: 71

[47] IBM. IBM XL C/C++ for AIX, V13.1 delivers IBM POWER8 exploitation and enhancements to improve performance and language standards conformance. `http://www-01.ibm.com/common/ssi/rep_ca/2/897/ENUS214-162/ENUS214-162.PDF`, April 2014. Cited on page: 17

[48] Ali Jannesari and Walter F. Tichy. Identifying ad-hoc synchronization for enhanced race detection. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10, April 2010. DOI: 10.1109/IPDPS.2010.5470343. Cited on page: 84, 100

[49] Baris Kasikci, Cristian Zamfir, and George Candea. Automated classification of data races under both strong and weak memory models. *ACM Trans. Program. Lang. Syst.*, 37(3):8:1–8:44, May 2015. ISSN 0164-0925. DOI: 10.1145/2734118. Cited on page: 22

[50] Gokcen Kestor, Osman S. Unsal, Adrian Cristal, and Serdar Tasiran. T-Rex: A dynamic race detection tool for c/c++ transactional memory applications. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 20:1–20:12, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2704-6. DOI: 10.1145/2592798.2592809. Cited on page: 98

[51] Michael Kling, Sasa Misailovic, Michael Carbin, and Martin Rinard. Bolt: On-demand infinite loop escape in unmodified binaries. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 431–450, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6. DOI: 10.1145/2384616.2384648. Cited on page: 100

[52] Tom Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 105–112, New York, NY, USA, 1986. ACM. ISBN 0-89791-200-4. DOI: 10.1145/319838.319854. Cited on page: 16

[53] Dongyoon Lee, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Chimera: Hybrid program analysis for determinism. In *In ACM Conference on Programming Language Design and Implementation*, pages 463–474, 2012. Cited on page: 97

[54] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. DTHREADS: Efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 327–336, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. DOI: 10.1145/2043556.2043587. Cited on page: 33, 34, 35, 85, 90, 95, 96, 97, 100

[55] David B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *SIGOPS Oper. Syst. Rev.*, 11(2):128–137, March 1977. ISSN 0163-5980. DOI: 10.1145/390018.808319. Cited on page: 16

[56] Kai Lu, Xu Zhou, Tom Bergan, and Xiaoping Wang. Efficient deterministic multithreading without global barriers. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 287–300, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2656-8. DOI: 10.1145/2555243.2555252. Cited on page: 33, 35, 95, 97, 100

[57] Li Lu and Michael L. Scott. Toward a formal semantic framework for deterministic parallel programming. In *Proceedings of the 25th International Conference on Distributed Computing*, DISC'11, pages 460–474, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24099-7. Cited on page: 20

[58] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-958-6. DOI: 10.1145/1346281.1346323. Cited on page: 18

[59] Walther Maldonado, Patrick Marlier, Pascal Felber, Julia Lawall, Giller Muller, and Etienne Rivière. Deadline-aware scheduling for software transactional memory. In *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 257–268, June 2011. DOI: 10.1109/DSN.2011.5958224. Cited on page: 99

[60] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of nonblocking software transactional memory. In *DEPT. OF COMPUTER SCIENCE, UNIV. OF ROCHESTER*, 2006. Cited on page: 14

[61] Alexander Matveev and Nir Shavit. Reduced hardware transactions: A new approach to hybrid transactional memory. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13,

pages 11–22, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1572-2. DOI: 10.1145/2486159.2486188. Cited on page: 17

[62] Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. *SIGARCH Computer Architecture News*, 34(2):53–65, 2006. ISSN 0163-5964. Cited on page: 101

[63] Timothy Merrifield and Jakob Eriksson. Conversion: Multi-version concurrency control for main memory segments. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 127–139, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1994-2. DOI: 10.1145/2465351.2465365. Cited on page: 95, 97

[64] Timothy Merrifield, Joseph Devietti, and Jakob Eriksson. High-performance determinism with total store order consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 31:1–31:13, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3238-5. DOI: 10.1145/2741948.2741960. Cited on page: 90, 97, 100

[65] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46. IEEE. Cited on page: 26, 34, 41, 71

[66] Mark Moir, Kevin Moore, and Dan Nussbaum. The adaptive transactional memory test platform: A tool for experimenting with transactional code for rock. In *TRANSACT '08: 3rd Workshop on Transactional Computing*, Februar 2008. Cited on page: 16

[67] Pablo Montesinos, Luis Ceze, and Josep Torrellas. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 289–300, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3174-8. DOI: 10.1109/ISCA.2008.36. Cited on page: 97

[68] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265, 2006. Cited on page: 14

[69] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: Model and architecture sketches. *Sci. Comput. Program.*, 63(2):186–201, December 2006. ISSN 0167-6423. DOI: 10.1016/j.scico.2006.05.010. Cited on page: 14

[70] Satish Narayanasamy, Gilles Pokam, and Brad Calder. BugNet: Continuously recording program execution for deterministic replay debugging. *SIGARCH Comput. Archit. News*, 33(2):284–295, May 2005. ISSN 0163-5964. DOI: 10.1145/1080695.1069994. Cited on page: 97

[71] Adrian Nistor, Darko Marinov, and Josep Torrellas. InstantCheck: Checking the determinism of parallel programs using on-the-fly incremental hashing. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 251–262. IEEE Computer Society, 2010. Cited on page: 3, 18

[72] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. *SIGPLAN Not.*, 44(3):97–108, March 2009. Cited on page: 30, 95, 96, 97

[73] Victor Pankratius and Ali-Reza Adl-Tabatabai. A study of transactional memory vs. locks in practice. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 43–52. ACM, 2011. Cited on page: 101

[74] Martin Pohlack and Stephan Diestelhorst. From lightweight hardware transactional memory to lightweight lock elision. In *TRANSACT '11: 6th Workshop on Transactional Computing*, June 2011. Cited on page: 25

[75] Stefan Poledna. Replica determinism in distributed real-time systems: A brief survey. *Real-Time Syst.*, 6(3):289–316, May 1994. ISSN 0922-6443. DOI: 10.1007/BF01088629. Cited on page: 21

[76] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 161–176. ACM, 2009. Cited on page: 51, 100

[77] Thomas Rauber and Gudula Rünger. *Parallel Programming - for Multicore and Cluster Systems.* Springer, 2010. ISBN 978-3-642-04817-3. Cited on page: 1, 2

[78] Kaushik Ravichandran, Ada Gavrilovska, and Santosh Pande. DeSTM: Harnessing determinism in STMs for application development. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 213–224, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2809-8. DOI: 10.1145/2628071.2628094. Cited on page: 95, 97

[79] James Reinders. Transactional synchronization in Haswell. http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/, Februar 2012. Cited on page: 16, 25, 50, 68

[80] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 284–298, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-44624-9, 978-3-540-44624-8. Cited on page: 24

[81] Torvald Riegel, Patrick Marlier, Martin Nowack, Pascal Felber, and Christof Fetzer. Optimizing hybrid transactional memory: the importance of nonspeculative operations. In *SPAA '11: Proceedings of the twenty-third annual symposium on Parallelism in algorithms and architectures*, pages 53–64. ACM, 2011. ISBN 978-1-4503-0743-7. DOI: 10.1145/1989493.1989501. Cited on page: 2, 25, 51, 55, 68

[82] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. TxLinux: using and managing hardware transactional memory in an operating system. In *SOSP '07: Proceedings of the Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, pages 87–102. ACM, 2007. Cited on page: 101

[83] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? In *ACM Sigplan Notices*, volume 45, pages 47–56. ACM, 2010. Cited on page: 2

[84] Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. 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*, pages 399–412. ACM, 2014. Cited on page: 58, 71, 101

[85] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990. ISSN 0360-0300. DOI: 10.1145/98163.98167. Cited on page: 21

[86] Cedomir Segulja and Tarek S. Abdelrahman. What is the cost of weak determinism? In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 99–112, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2809-8. DOI: 10.1145/2628071.2628099. Cited on page: 95

[87] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-793-6. DOI: 10.1145/1791194.1791203. Cited on page: 22

[88] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM. ISBN 0-89791-710-3. DOI: 10.1145/224964.224987. Cited on page: 2, 16

[89] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in STM. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 78–88, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. DOI: 10.1145/1250734.1250744. Cited on page: 20, 99

[90] Michael F. Spear, Maged Michael, and Michael L. Scott. Inevitability mechanisms for software transactional memory. In *TRANSACT '08: 3rd Workshop on Transactional Computing*, February 2008. Cited on page: 101

[91] Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. Dynamic recognition of synchronization operations for improved data race detection. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 143–154, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-050-0. DOI: 10.1145/1390630.1390649. Cited on page: 84, 100

[92] Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. Automated dynamic detection of busy wait synchronizations. *Softw. Pract. Exper.*, 39(11): 947–972, August 2009. ISSN 0038-0644. DOI: 10.1002/spe.v39:11. Cited on page: 99

[93] Takayuki Usui, Reimer Behrends, Jacob Evans, and Yannis Smaragdakis. Adaptive locks: combining transactions and locks for efficient concurrency. In *PACT '09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14. IEEE Computer Society. ISBN 978-0-7695-3771-9. DOI: http://dx.doi.org/10.1109/PACT.2009.20. Cited on page: 101

[94] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. DoublePlay: Parallelizing sequential logging and replay. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 15–26, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0266-1. DOI: 10.1145/1950365.1950370. Cited on page: 97

[95] Duc Vianney. Tiobench benchmark - ltc linux performance team. `http://linuxperf.sourceforge.net/tiobench/tiobench.php`. Cited on page: 28, 67, 74, 80, 84, 89

[96] Haris Volos, Neelam Goyal, and Michael Swift. Pathological interaction of locks with transactional memory. In *TRANSACT '08: 3rd Workshop on Transactional Computing*, February 2008. Cited on page: 50, 55, 101

[97] Haris Volos, Andres Jaan Tack, Neelam Goyal, Michael M. Swift, and Adam Welc. xCalls: Safe I/O in memory transactions. In *EuroSys '09: Proceedings of the 4th ACM European Conference on Computer Systems*, pages 247–260. ACM, 2009. ISBN 978-1-60558-482-9. Cited on page: 51, 100, 101

[98] Felix von Leitner. diet libc. http://www.fefe.de/dietlibc/talk.pdf, 2001. Cited on page: 50

[99] Christoph von Praun, Luis Ceze, and Calin Caşcaval. Implicit parallelism with ordered transactions. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 79–89, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-602-8. DOI: 10.1145/1229428.1229443. Cited on page: 99

[100] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of blue Gene/Q hardware support for transactional memories. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 127–136, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1182-3. DOI: 10.1145/2370816.2370836. Cited on page: 16

[101] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 285–296, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-973-9. DOI: http://doi.acm.org/10.1145/1378533.1378584. Cited on page: 101

[102] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association. Cited on page: 84, 99

[103] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 122–133. IEEE, 2003. Cited on page: 30, 34, 41, 74, 80

[104] Junfeng Yang, Heming Cui, Jingyue Wu, Yang Tang, and Gang Hu. Making parallel programs reliable with stable multithreading. *Commun. ACM*, 57(3):58–69, March 2014. ISSN 0001-0782. DOI: 10.1145/2500875. Cited on page: 3, 18

[105] Matt T. Yourst. PTLsim: a cycle accurate full system x86-64 microarchitectural simulator. In *Performance Analysis of Systems Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 23 –34, April 2007. DOI: 10.1109/ISPASS.2007.363733. Cited on page: 25, 68

[106] Ferad Zyulkyarov, Vladimir Gajinov, Osman S. Unsal, Adrián Cristal, Eduard Ayguadé, Tim Harris, and Mateo Valero. Atomic Quake: Using transactional memory in an interactive multiplayer game server. *SIGPLAN Not.*, 44(4):25–34, February 2009. ISSN 0362-1340. DOI: 10.1145/1594835.1504183. Cited on page: 71

[107] Ferad Zyulkyarov, Tim Harris, Osman S. Unsal, Adrián Cristal, and Mateo Valero. Debugging programs that use atomic blocks and transactional memory. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, pages 57–66. ACM, 2010. ISBN 978-1-60558-877-3. Cited on page: 98