TM-dietlibc: A TM-aware Real-world System Library

Vesna Smiljković*, Martin Nowack†, Nebojša Miletić*°, Tim Harris‡⋆, Osman Ünsal*, Adrián Cristal*, Mateo Valero*
*Barcelona Supercomputing Center, Spain
{vesna.smiljkovic, osman.unsal, adrian.cristal, mateo.valero}.bsc.es
miletic n@gmail.com
†Technische Universität Dresden, Germany
martin@se.inf.tu-dresden.de
‡Oracle Labs, Cambridge, UK
timothy.l.harris@oracle.com

***Abstract*—The simplicity of concurrent programming with Transactional Memory (TM) and its recent implementation in mainstream processors greatly motivates researchers and industry to investigate this field and propose new implementations and optimizations. However, there is still no standard C system library which a wide range of TM developers can adopt. TM application developers have been forced to avoid library calls inside of transactions or to execute them irrevocably (i.e. in serial order). In this paper, we present the first TM-aware system library, a complex software implementation integrated with TM principles and suited for software (STM), hardware (HTM) and hybrid TM (HyTM).**

The library we propose is derived from a modified lock-based implementation and can be used with the existing standard C API. In our work, we describe design challenges and code optimizations that would be specific to any TM-based system library or application. We argue about system call execution within transactions, highlighting the possibility of unexpected results from threads. For this reason we propose: (1) a mechanism for detecting conflicts over kernel data in user space, and (2) a new barrier to allow hybrid TM to be used effectively with system libraries.

Our evaluation includes different TM implementations and the focus is on memory management and file operations since they are widely used in applications and require additional mechanisms for concurrent execution. We show the benefit we gain with our libc modifications providing parallel execution as much as possible. The library we propose shows high scalability when linked with STM and HTM. For file operations it shows on average a 1.1, 2.6 and 3.7x performance speedup for 8 cores using HyTM, STM and HTM, respectively (over a lock-based single-threaded execution). For a red-black tree it shows on average 3.14x performance speedup for 8 cores using STM (over a multi-read single-threaded execution).

***Keywords*-transactional memory; system library; system calls; I/O; memory allocation**

I. Introduction

The growing number of on-chip cores and hardware threads increases the need for parallel programming models that are easy to use by average programmers and that can assure proper multithreaded synchronization. Transactional Memory (TM) [1], [2], either as a software (STM), a hardware (HTM) or a hybrid (HyTM) implementation, has been proposed as a solution to this issue. After dozens of TM implementations, several TM applications [3], [4], [5], [6] and research on its fundamentals, a system library (a layer between an operating system and an application) compatible with transactional memory is missing.

Current TM implementations handle library code either by marking it ineligible for execution in a transaction (and reporting an error if the library is invoked), or by using irrevocable execution (so that only one transaction may call into the library at any time). The first approach limits the ability for complex programs to be adapted to use transactions. The second approach limits the scalability of transactional programs. In this paper, we examine how to adapt a complex existing library so that it can be used within transactions without making all operations irrevocable.

System libraries abstract and simplify the access to services of the operating system and encapsulate a shared state: e.g. memory allocation lists and file structures. If a library is used in a large program, then its internal state might be accessed within transactions in some threads, and concurrently accessed un-intendedly outside transactions in other threads using other protection schemas (e.g. 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 behavior [7]. In contrast, Hardware TM implementations like AMD's Advanced Synchronization Facility (ASF) [8] or Intel's Restricted TM (RTM) in the upcoming processor Haswell [9] provide strong isolation guarantees assuring consistent views even for unprotected access.

In this paper, we present a TM-aware implementation of a standard C system library, based on *diet libc* [10].

°This work was conducted when the author was with Barcelona Supercomputing Center.

⋆This work was conducted when the author was with Microsoft Research Cambridge.

Diet libc is an open-source standard C library designed to have the smallest possible code footprint. We modify the original lock-based implementation to make 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 require some degree of change, in the form of a particular API [11], [12] or specialized transactional calls [13].

This paper makes the following contributions:

- We present the first TM-aware system library with the standard libc API. It is based on an originally lock-based library and the API remains unchanged in order to allow software developers to maintain their programming habits and to ease the transactification of existing software.
- We explain general design choices related to (1) interaction between lock-based and transactional library code, (2) system library adaptation to a TM system, and (3) system library execution. We explain which design choices are suitable for system libraries, and we detail our experience with diet libc and TM integration, transactional code optimizations, insufficient support of TM tools, etc.
- We introduce a TM conflict detection mechanism extension, a new technique to solve the problem of detecting conflicts that cannot be detected automatically by TM because they modify kernel space. Furthermore, we explain how to support transactions with system calls in HyTM and avoid running them sequentially.

We evaluate our TM-aware diet libc implementation using a set of micro-benchmarks for file operations and the red-black tree benchmark for memory management functions. We compare our TM-aware implementation using different TM implementations with the traditional approach of running library calls irrevocably or with locks. We show that TM-aware implementation is scalable with a significant speedup for running on 8 cores. In addition, we demonstrate the first comparison with emulated Intel's RTM implementation. TM-dietlibc is an open-source system library, available at [14].

In the following sections, we present: different design choices for integrating a system library to TM usage (Section II); modifications we made to the library itself and experience we gained which can be applied to other system libraries (Section III); the quantification effort (Section IV); limitations of a *diet* system library and existing TM support (Section V); the experimental results (Section VI); the comparison of our work to previous studies (Section VII); and our conclusions (Section VIII).

II. Various system library designs

Developing a TM-based system library or changing an existing lock-based one involves choosing applicable programming models and designs. In this section, we describe different choices related to (1) how to *mix* locks and transactions, (2) how to *adapt* a library to a TM system, and (3) how to *execute* library code.

Our design choices for TM-dietlibc are based on two external dependencies (rather than on the original lock-based libc): the TM implementation and the TM compiler.

A. Mixing locks and transactions

Applications can invoke system 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 atomicity provided by a TM implementation. With strong atomicity, transactional accesses of a shared variable are totally synchronized with unprotected accesses by other threads. However, most of STM implementations do not guarantee strong isolation.

Partially shared data. Dynamic separation [15], [16] is a programming model where a programmer indicates shared variables that could be accessed within or outside of 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 [17] and the system library has to contain duplicated data structures and duplicated code wherever a memory access can be transactional and non-transactional.

TM-dietlibc design choice: In the entire system library we completely separate shared data protected by locks and protected by TM. The reasons for not choosing mixing transactional and lock-protected data and code are the following: (1) 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, (2) disabling concurrent execution: if one transaction acquires a lock, all other transactions would have to wait until the first one finishes, and (3) causing non-trivial pathological behaviour [7], e.g. a deadlock. The approach we choose is defensive, distinguishes transactional and non-transactional accesses and does not require strong atomicity from TM.

B. 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 system library adaptation to one specific or various TM systems influences its complexity and portability.

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

Library independent of TM implementations. Employing different TM implementations becomes straightforward with Intel's ABI [18], and recent work on a framework makes it easy to integrate an STM library backend to Intel/gcc ABI compliant STM compilers [19]. However, a system library that does not depend on a specific TM implementation has to rely only 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 systems [8], [20], [21], 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 in the irrevocable mode.

C. Library execution mode

To exploit optimistic concurrency that TM provides, the ideal case is when system-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. In this subsection, we describe different possibilities to run system library functions.

Complete transactional execution. Libc code can be executed transactionally if it contains only local variables and memory accesses at the user level.

Transactional execution employing TM techniques. More sophisticated TM systems might be able to handle non-trivial cases. e.g. TM support for locking inside transactions.

Transactional execution in a TM-adapted system 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 system library call to serial irrevocable execution ensures safe execution without any libc changes. However, executing transactions sequentially does not benefit from the parallel execution of programs.

Non-transactional execution within TM integration. A system library can be executed non-transactionally, but with a certain adaptation for TM integration. For instance, the Intel [18] and DTMC [8] 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: (1) complete transactional execution is possible when there are no system calls during the execution, (2) transactional execution employing TM techniques is preferable when some system calls occur, and we employ abort and commit handlers provided by TM, (3) transactional execution in a TM-adapted system library is for the cases when we apply the conflict detection extension implemented in TM-dietlibc, and (4) sequential execution for the system calls when we have to transit to irrevocable execution.

III. The transactification of diet libc - implementation experience

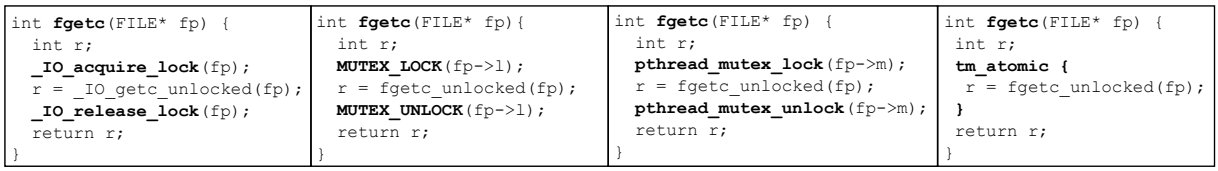
In this section, we present details on how we modified the lock-based diet libc in order to integrate it with TM. Some of the modifications are simple; however, the majority required significant effort to: (1) identify groups of locks which are used to protect access to a shared resource, (2) replace them with transaction boundaries, (3) use TM techniques in appropriate way for system library functions, (4) implement and apply a TM conflict detection extension, and (5) support hybrid TM.

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

A. Identifying groups of locks

A system 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.

[1]http://gnu.org/software/libc/
[2]http://eglibc.org/
[3]http://uclibc.org/



```
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->l);
  r = fgetc_unlocked(fp);
  MUTEX_UNLOCK(fp->l);
  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;
 tm_atomic {
  r = fgetc_unlocked(fp);
 }
 return r;
}
```

(d) TM-dietlibc

Figure 1: Lock-based implementations of `fgetc` for different system libraries (a), (b), (c) and the transactional counterpart (d). Lock operations in diet libc are replaced with block marked with boundaries `tm_atomic{}`.

B. 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 function (shown in Figure 1 for different system libraries (a), (b), (c)). This way, they can be easily replaced with the boundaries of an atomic block (Figure 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 system 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: (1) when `lock`/`unlock` pairs do not satisfy a TM requirement of having critical section boundaries in one function scope, and (2) 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 transaction's atomicity and isolation when a transaction contains only local variables or memory accesses at the user level. These cases appear in system libraries infrequently.

C. Applying TM techniques on the library functions with system calls

Various functions from a system library make modifications in kernel state which a TM system cannot track or they cause side effects which a TM system 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. TM compilers such as DTMC [8] and Intel [18] wrap original lock-based functions, adding additional structures and commit/abort handlers [22]. Since our goal is to ensure concurrent execution of memory management operations, we do not rely on the compiler's wrappers. Instead, we provide: (1) speculative execution of these functions, (2) an abort handler, used for the functions that allocate memory, and (3) a commit handler, used for the functions that release memory. The usage of abort and commit handlers is shown in Figure 2(a) and (b). In our implementation, additional structures are not necessary, and the handlers are registered only when a system call occurs. In all other cases, the TM system 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 read and write sets of transactions, 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 2(a) and (b)).

Although TM-dietlibc provides two different implementations for every memory management function, benchmarks contain only the calls of the original functions no matter if the calls are inside or outside of transactions. When a TM compiler instruments a benchmark, it finds the calls from within transactions and replaces them with their transactional counterpart. Therefore, the API used by the programmer and benchmarks remain unchanged.

File operations provide communication (1) between a user and a program and (2) between 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 as irrevocable [23]. In that case, the transaction waits the other running transactions to finish their execution, and then it continues as the only transaction running. When it commits, new

```
void* malloc(size){
 void* r;
 lock(&mutex_alloc);
 if(libc_chunks.empty())
  r = mmap(size);

 else
  r=libc_chunks.pop();
 unlock(&mutex_alloc);
 return r;
}

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

```
void* tx_malloc(size){
 void* r;
 tm_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 and TM-based malloc

```
void free(ptr){
 lock(&mutex_alloc);
 if (libc_chunks.full())
  munmap(ptr);
 else
  libc_chunks.push(ptr);
 unlock(&mutex_alloc);
}

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

```
void tx_free(ptr){
 tm_atomic {
  if (tx_libc_chunks.full())
   onCommit(munmap,ptr);
  else
   tx_libc_chunks.push(ptr);
 }
}

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

(b) lock-based and TM-based free

```
size_t fwrite(ptr,size,fp){
 size_t r;
 lock(fp->m);
  if (!fp->buf.full())
   r = size;
   memcpy(fp->buf,ptr,size);
  else

   r = write(fp->fd,ptr,size);
 unlock(fp->m);
 return r;
}
// syscall:
size_t write(fd,buf,size);
```

```
size_t fwrite(ptr,size,fp){
 size_t r;
 tm_atomic {
  if (!fp->buf.full())
   r = size;
   memcpy(fp->buf,ptr,size);
  else
   go_irrevocable;
   r = write(fp->fd,ptr,size);
 }
 return r;
}
// syscall:
size_t write(fd,buf,size);
```

(c) lock-based and TM-based fwrite

Figure 2: Examples of lock-based and TM-based libc functions: (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`, and (c) `fwrite` with the late irrevocability, i.e. going irrevocably only if a system call needs to be invoked.

transactions are allowed to start and execute again in parallel.

I/O operations 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 disc, the library calls `read`, `write` and `lseek` to fill the buffer, empty the buffer, and update the file position, respectively. Since I/O operations occasionally invoke file changes in the kernel space, we allow *late irrevocability*, i.e. a transaction executes concurrently until the system call occurs and only then the TM system changes the execution mode of the transaction (illustrated with an update to disk in Figure 2(c)).

D. Conflict detection extension

The mechanisms explained so far are sufficient for integrating any code with TM. However, executing irrevocable transactions impacts concurrency, and threads spend most of the time waiting for an irrevocable transaction to finish execution and commit changes. One of the examples when TM should run a transaction irrevocably 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 irrevocable transactions running in a system, we propose an extension for the TM conflict detection mechanism. The extension ensures that: (1) transactions run simultaneously, (2) TM keeps track of shared kernel resources and (3) TM detects possible conflicts over shared kernel resources.

Our proposal produces and leverages copies of the relevant data from kernel space that are changed during transaction execution. These copies reside in user space; consequently, they are under complete control of the programmer and the TM system. The system 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 system can detect conflicts and invoke the transaction's undo function which will use the updated copy to revert the state of kernel space.

We illustrate our idea using the `lseek` system call as an example, as shown in Figure 3(a). Since `lseek` changes the file pointer from kernel space, a system library programmer has to make a copy of the file pointer in user space and to keep the copy updated. The variable is called `OS_fpos`, and it is kept in the system library, as a part of the `FILE` structure. As a consequence, a TM system is able to track reads and writes over the shared variable.

In order to provide conflict detection without invoking the call `lseek`, the transaction tries to acquire a writing lock[4] for the shared variable before the system call. 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[5]. 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. In any case of an abort, the state of both user space and kernel space is rolled back to how it was before the transaction started its execution.

E. System-call barrier in HyTM

The different approaches detailed in the previous sections allow concurrent multi-threaded executions of transactions with system calls inside of software transactions. However, TM using hardware support, e.g. ASF-TM [8], does not allow system calls inside a running hardware transaction; the transaction is aborted immediately and re-executed as an irrevocable transaction.

[4]All locks acquired during transaction execution are released at commit/abort.

[5]Early conflict detection is typical for eager TM systems.

```
int fseek(fp,offset,whence) {
 int r; param* p;
 tm_atomic {
  p={fp->fd, fp->OS_fpos};
  //causing a possible conflict if
  //another thread holds the lock:
  acquire_wr_lock(fp->OS_fpos);
  r=lseek(fp->fd, offset, whence);
  //update the pointer:
  fp->OS_fpos = r;
  onAbort(undo_lseek, p);
 }
 return r;
}
void undo_lseek(void* p) {
  lseek(p->fd, p->pos, SEEK_SET);
}
// syscall:
int lseek(fd, offset,whence);
```

(a) conflict detection extension for STM

```
int fseek(fp,offset,whence) {
 int r; param* p;
 tm_atomic {
  p={fp->fd, fp->OS_fpos};
  //the barrier aborts a hw txn:
  go_safe_syscall;
  acquire_wr_lock(fp->OS_fpos);
  r=lseek(fp->fd, offset, whence);

  fp->OS_fpos = r;
  onAbort(undo_lseek, p);
 }
 return r;
}
void undo_lseek(void* p) {
  lseek(p->fd, p->pos, SEEK_SET);
}
// syscall:
int lseek(fd, offset,whence);
```

(b) conflict detection extension for HyTM with a safe-syscall barrier

Figure 3: The conflict detection extension: the variable `OS_fpos`, reflecting the current position indicator of the kernel, is used for tracking the file position and detecting conflicts. A `go_safe_syscall` barrier in (b) aborts a hardware transaction and re-execute it speculatively in software.

To increase the possibility of the parallel execution of transactions with system calls, we propose a *safe-syscall* execution mode. Before each system call inside a transaction, we put a *safe-syscall barrier* (Figure 3(b)) which will notify HTM about an upcoming system call. HTM aborts and re-executes the transaction using software fall back solutions; this allows a high number of software and hardware transactions to run in parallel.

IV. Quantification effort

The significance of the effort needed in modifying diet libc is in its integration in a complex TM system. Each TM principle implemented in the system 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 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 transaction blocks were inserted into TM-dietlibc. The difference arises from the fact that we mapped original unsafe functions with transactions as well (described in III-B).

The type and length of the transactions itself 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), i.e. only instrumented and executed code. Otherwise, `write` and `seek` operations will 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 TM-aware system libraries. In Section V we describe in detail the major impact on the development.

V. Limitations of a "diet" system library and existing TM tools

Choosing a system 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 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, a function `fgets` that reads a string from a file, first 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, read/write sets 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.

The lack of debugging and profiling tools for various TM libraries makes testing, debugging and performance tuning substantially difficult and time-consuming. Moreover, TM libraries and TM compilers were developed for the application usage, rather than for the usage of system libraries, and many modifications were needed to ensure a TM-aware library to be successfully built.

VI. Evaluation

For the evaluation of TM-dietlibc, we use two different types of applications: (1) micro-benchmarks with file operations and (2) the red-black tree benchmark [24] with memory management functions. For compiling TM-dietlibc and the benchmarks, we use Dresden TM Compiler (DTMC) [8], based on GCC and LLVM. DTMC ensures that all code within transactions is either instrumented and executed in the parallel mode or executed in the irrevocable mode, which happens only when we explicitly change the execution mode to irrevocable. This provides correct execution of transactions in TM-dietlibc. We also executed additional micro-benchmarks with glibc and TM-dietlibc and compared

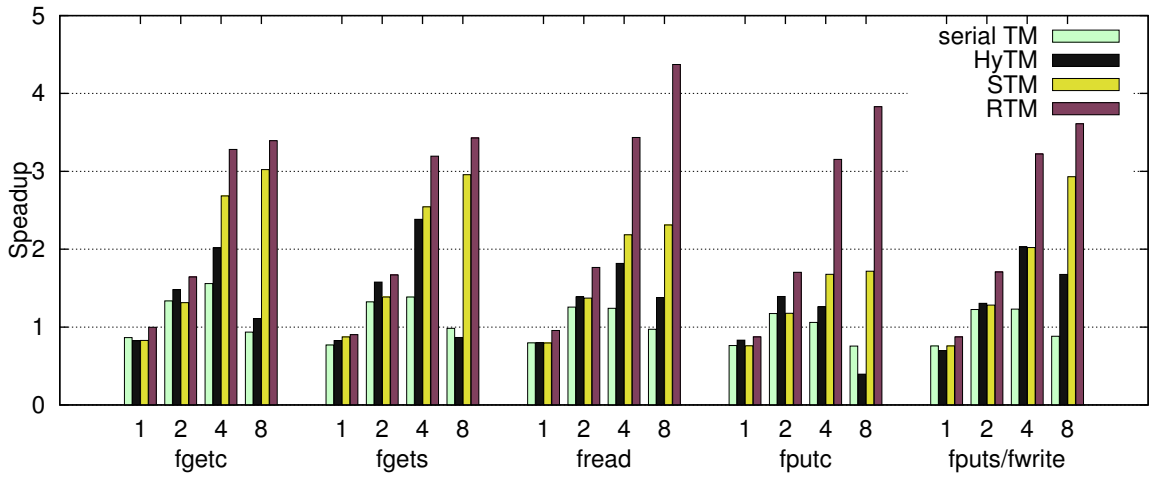


Figure 4: Evaluation of file operations executed by 1, 2, 4 and 8 threads, with various TM implementations and normalized to a lock-based single-threaded execution. Emulated ASF gives results very close to RTM, therefore they are not shown.

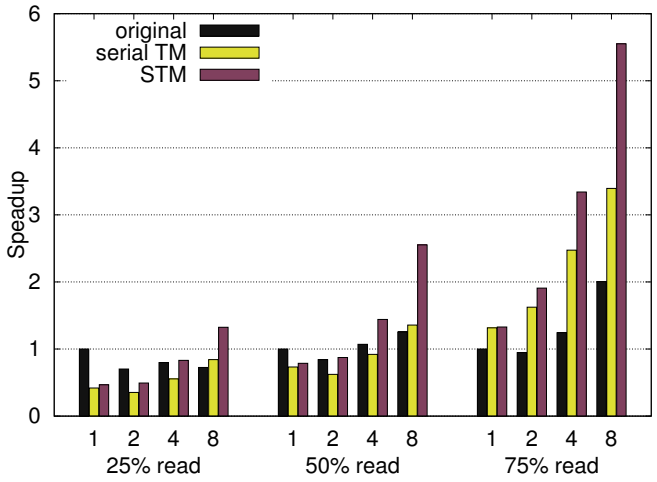


Figure 5: Evaluation of the red-black tree benchmark by 1, 2, 4 and 8 threads, original and with STM, all normalized to the original single-threaded execution.

their outputs to be sure that our modifications keep the correctness of a system library[6].

We run the benchmarks with STM, HTM and HyTM implementations. The STM library (TinySTM [20]) and its variants are a set of lightweight, highly efficient, word- and time-based STM implementations. The executions involving HTM [8] and HyTM [21] were conducted using a nearly cycle accurate CPU simulator PTLSim [25] with ASF extension from AMD. We emulate Intel's Haswell by changing the simulation mode to handle all memory accesses inside an atomic block as transactional. For benchmarks execution and to host the simulation environment we used the machines featuring 2 Intel Xeon E5405 processors each with 4 cores and a clock speed of 2.00GHz per core, with 4GiB RAM. The I/O experiments are conducted on a single 380 GB SATA hard drive. The sizes of the test files are 2GiB.

The main difference for an application programmer without a TM-aware system library would be going irrevocable (serial) before calling library functions. To compare our implementation with the previous approach, we compiled two versions of our TM-aware library: (1) transactional and linked with various TM implementations (HTM, STM and HyTM), and (2) transactional with switching irrevocable at the beginning of a transaction (serial TM).

The benchmarks we use for the TM-dietlibc evaluation include different file operations: `fgetc`, `fgets`, `fread`, `fputc`, `fputs` and `fwrite` (although the last two have the same implementation in diet libc). Multithreaded file accesses are done using a shared file descriptor and its associated `FILE` structure. They illustrate possible multi-threaded accesses to the same file and usage of the output of the file operations. Figure 4 shows that all transactional versions perform better than the irrevocable version, which is the only option without proper TM support in the system library. The benchmarks we use for the TM-dietlibc evaluation show high scalability and provide on average a 1.1, 2.6, 3.7 and 3.6x performance speedup for 8 cores for functions running with HyTM, STM, ASF and RTM, respectively. Two HTM implementations (ASF and RTM) perform similarly and we show only RTM.

The red-black tree benchmarks[7] (Figure 5) performs read, write and remove operations over the elements of a balanced tree structure, where reads are parallel and writes serial. For a smaller number of read operations, the number of write and

[6]This comparison is only for proving the correctness and is not shown in the paper

[7]We used a 32bit version, ASF hardware TM extension is only usable for 64bit.

remove operations including memory allocation/deallocation is higher. Therefore, general performance is lower in comparison to more read-dominated benchmarks. However, the differences for speedups between the transactional and irrevocable versions are larger for the benchmarks with many memory operations in comparison to the more lightweight runs.

VII. Related work

A preliminary version of our work [26] was presented at the non-archiving ACM SIGPLAN Workshop on Transactional Computing, and we showed the first progress in the transactification of a system library. The current work extends it by presenting different design choices for the integration of a system library and TM. We show how to support transactions with system calls in HTM, and we improve the evaluation of the library with various benchmarks and TM implementations including emulated Intel's RTM.

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. For instance, Volos et al. [11] 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. [12] provide a Java library with an API extended with several functions, which ensures that file changes remain local until commit time. Porter et al. [13] implement transactions on the operating system level which require invocation of specific system calls in the user transaction. On the other hand, we keep the standard C API so an application developer does not have to modify his code, and provide parallel execution of majority of transactions.

In related work [22], [27], [11] the authors suggest that some critical actions should be deferred until commit time. However, for nested transactions[8] 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 dynamically decide whether a critical section should be transactional or lock-based by Usui et al. [28] or for a new type of lock (*transaction-safe*) by Volos et al. [7]. Similar to that, Rossbach et al. [29] introduce *cooperative transactional locks*. Gottschlich and Chung [30] 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.

Ultimately, this is the first proposal that detects kernel space conflicts on 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 [22], [31] or irrevocable execution [32], [23] for handling system calls and I/O inside transactions. Pankratius et al. [33] analyzing 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 irrevocably limits concurrency and scalability.

VIII. Conclusion

This paper presents the first real-world TM-aware standard C implementation with the API unchanged. We describe various design choices for integrating a system library with TM and explain what decisions we made for diet libc. We propose static separation of locks and transactions as a safe way to handle the interaction of these two inherently different synchronization concepts. We discuss handling system calls inside transactions and reveal a pitfall in detecting kernel space conflicts which would require many transactions to execute only as irrevocable. For this case, we propose a technique which enables detection of such conflicts in the scope of the system library, rather than involving complex kernel modifications.

Our results for memory management and file operations show that we achieve much better performance in our proposal over lock-based and irrevocable execution, we provide support for HTM, STM and HyTM, and we provide the first comparison with emulated Intel's RTM from the upcoming processors.

Acknowledgments

We would like to thank the anonymous reviewers for their useful comments. This work was partially supported by the cooperation agreement between the Barcelona Supercomputing Center and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contracts TIN2007-60625 and TIN2008-02055-E, by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC) and by the European Commission FP7 project VELOX (216852). Vesna Smiljković is also supported by a scholarship from The Polytechnic University of Catalonia (BarcelonaTech).

References

[1] M. Herlihy and J. Moss, "Transactional memory: architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993, pp. 289–300.

[8] a very likely case - a TM-based applications invoking a call from a TM-based library

[2] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory (Synthesis Lectures on Computer Architecture)*, 2nd ed. Morgan & Claypool Publishers, June 2010.

[3] G. Kestor, V. Karakostas, O. S. Unsal, A. Cristal, I. Hur, and M. Valero, "RMS-TM: a comprehensive benchmark suite for transactional memory systems," in *Proceeding of the second joint WOSP/SIPEW international conference on Performance engineering*, ser. ICPE '11. ACM, 2011, pp. 335–346.

[4] F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero, "Atomic Quake: using transactional memory in an interactive multiplayer game server," in *PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2009, pp. 25–34.

[5] V. Gajinov, F. Zyulkyarov, O. S. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero, "QuakeTM: parallelizing a complex sequential application using transactional memory," in *ICS '09: Proceedings of the 23rd International Conference on Supercomputing*. ACM, 2009, pp. 126–135.

[6] D. Lupei, B. Simion, D. Pinto, M. Misler, M. Burcea, W. Krick, and C. Amza, "Transactional memory support for scalable and transparent parallelization of multiplayer games," in *Proceedings of the 5th European conference on Computer systems*, ser. EuroSys '10. ACM, 2010, pp. 41–54.

[7] H. Volos, N. Goyal, and M. Swift, "Pathological interaction of locks with transactional memory," in *TRANSACT '08: 3rd Workshop on Transactional Computing*, February 2008.

[8] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Riviere, "Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack," in *Proceedings of the 5th European conference on Computer systems*, ser. EuroSys '10. ACM, 2010, pp. 27–40.

[9] J. Reinders, "Transactional synchronization in Haswell," 2012. [Online]. Available: http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/

[10] F. von Leitner, "diet libc," 2001. [Online]. Available: http://www.fefe.de/dietlibc/talk.pdf

[11] H. Volos, A. J. Tack, N. Goyal, M. M. Swift, and A. Welc, "xCalls: Safe I/O in memory transactions," in *EuroSys '09: Proceedings of the 4th ACM European Conference on Computer Systems*. ACM, 2009, pp. 247–260.

[12] B. Demsky and N. F. Tehrany, "Integrating file operations into transactional memory," *Journal of Parallel and Distributed Computing*, vol. 71, pp. 1293–1304, October 2011.

[13] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel, "Operating system transactions," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. ACM, 2009, pp. 161–176.

[14] "TM-dietlibc source code," 2012. [Online]. Available: http://www.velox-project.eu/software/tmdietlibc

[15] M. Abadi, T. Harris, and K. F. Moore, "A model of dynamic separation for transactional memory," in *Proceedings of the 19th international conference on Concurrency Theory*, ser. CONCUR '08. Springer-Verlag, 2008, pp. 6–20.

[16] M. Abadi, A. Birrell, T. Harris, J. Hsieh, and M. Isard, "Implementation and use of transactional memory with dynamic separation," in *CC '09: Proceedings of the 18th International Conference on Compiler Construction*, 2009, pp. 63–77.

[17] M. Abadi, A. Birrell, T. Harris, and M. Isard, "Semantics of transactional memory and automatic mutual exclusion," in *POPL '08: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008, pp. 63–74.

[18] Intel, "Intel transactional memory compiler and runtime application binary interface," November 2008.

[19] G. Kestor, L. Dalessandro, A. Cristal, M. L. Scott, and O. Unsal, "Interchangeable back ends for STM compilers," in *TRANSACT*, June 2011.

[20] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008, pp. 237–246.

[21] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. 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*. ACM, 2011, pp. 53–64.

[22] L. Baugh and C. Zilles, "An analysis of I/O and syscalls in critical sections and their implications for transactional memory," in *TRANSACT*, August 2007.

[23] A. Welc, B. Saha, and A.-R. Adl-Tabatabai, "Irrevocable transactions and their applications," in *SPAA '08: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*. ACM, 2008, pp. 285–296.

[24] K. Fraser, "Practical lock freedom," Ph.D. dissertation, Cambridge University Computer Laboratory, 2003, also available as Technical Report UCAM-CL-TR-579.

[25] M. Yourst, "PTLsim: a cycle accurate full system x86-64 microarchitectural simulator," in *Performance Analysis of Systems Software, 2007. ISPASS 2007. IEEE International Symposium on*, April 2007, pp. 23 –34.

[26] N. Miletic, V. Smiljkovic, C. Perfumo, T. Harris, A. Cristal, I. Hur, O. Unsal, and M. Valero, "Transactification of a real-world system library," in *TRANSACT*, April 2010.

[27] R. Dias, J. M. S. Lourenço, and G. Cunha, "Developing libraries using software transactional memory," *Computer Science and Information Systems*, vol. 5, no. 2, pp. 103–117.

[28] T. Usui, R. Behrends, J. Evans, and Y. 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*. IEEE Computer Society, pp. 3–14.

[29] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. 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*. ACM, 2007, pp. 87–102.

[30] J. E. Gottschlich and J. 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.

[31] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun, "Architectural semantics for practical transactional memory," *SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 53–65, 2006.

[32] M. F. Spear, M. Michael, and M. L. Scott, "Inevitability mechanisms for software transactional memory," in *TRANSACT '08: 3rd Workshop on Transactional Computing*, February 2008.

[33] V. Pankratius and A.-R. Adl-Tabatabai, "A study of transactional memory vs. locks in practice," in *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '11. ACM, 2011, pp. 43–52.