Hindawi
Scientific Programming
Volume 2019, Article ID 9404323, 19 pages
https://doi.org/10.1155/2019/9404323

Research Article

Concurrency Bug Avoiding Based on Optimized Software Transactional Memory

Z. Yu,[1,2] **Y. Zuo**,[1,3] **and W. C. Xiong**[1,3]

[1]*Department of Mathematics and Big Data, Guizhou Education University, Guiyang 550018, China*
[2]*Guizhou Science and Technology Information Center, Guizhou Science and Technology Department, Guiyang 550002, China*
[3]*Big Data Science and Intelligent Engineering Research Institute, Guizhou Education University, Guiyang 550018, China*

Correspondence should be addressed to Z. Yu; yuzhen_3301@aliyun.com

Received 10 August 2018; Revised 25 November 2018; Accepted 2 January 2019; Published 3 February 2019

Academic Editor: Antonio J. Peña

Software transactional memory is an effective mechanism to avoid concurrency bugs in multithreaded programs. However, two problems hinder the adoption of such traditional systems in the wild world: high human cost for equipping programs with transaction functionality and low compatibility with I/O calls and conditional variables. This paper presents Convoider to solve these problems. By intercepting interthread operations and designating code among them as transactions in each thread, Convoider automatically transactionalizes target programs without any source code modification and recompiling. By saving/restoring stack frames and CPU registers on beginning/aborting a transaction, Convoider makes execution flow revocable. By turning threads into processes, leveraging virtual memory protection and customizing memory allocation/deallocation, Convoider makes memory manipulations revocable. By maintaining virtual file systems and redirecting I/O operations onto them, Convoider makes I/O effects revocable. By converting lock/unlock operations to no-ops, customizing signal/wait operations on condition variables, and committing memory changes transactionally, Convoider makes deadlocks, data races, and atomicity violations impossible. Experimental results show that Convoider succeeds in transparently transactionalizing twelve real-world applications with averagely incurring only 28% runtime overhead and perfectly avoid 94% of thirty-one concurrency bugs used in our experiments. This study can help efficiently transactionalize legacy multithreaded applications and effectively improve the runtime reliability of them.

1. Introduction

The currently ubiquitous multicore architecture necessitates concurrent programming, especially multithreaded programming [1–3]. Unfortunately, writing concurrent programs is challenging: multithreaded programs often suffer from concurrency bugs such as deadlocks, data races, atomicity violations, and order violations [4], which are notoriously difficult to expose [5, 6], detect [6–8], debug [8], and repair [9, 10] because of their nondeterminism nature. Thus, avoiding concurrency bugs at runtime is an attractive complementary approach to bug detection and fixing.

Software transactional memory (STM) [11–21] is a software-level concurrency control mechanism with which programmers can partition code into transactions and ensure them to execute atomically and in isolation with respect to each other. A transaction either commits at a commit point trivially or aborts, in which case it conflicts with other transactions, by revoking any effects it has made. STM can simplify concurrent programming by easing data protection, permitting sequential reasoning among transactions, and disabling part of concurrency bugs. Although STM is a promising technique for achieving easier/safer concurrency and avoiding concurrency bugs [22–25], it has drawbacks hindering its adoption in the wild world:

(i) *High human cost for equipping programs with transaction functionality*. Most STM systems are implemented as programming libraries with rich APIs for use. For moving legacy code from lock-based to transaction-based, programmers have to check the code carefully and insert low-level STM

API calls at proper points. The calls to STM APIs are used to demarcate transactions and identify potential shared accesses in them. Moreover, employing transactions also requires changes to the data/control structures [22, 25]. Although people have proposed methods [15] to alleviate this burden, they do not remove it all.

(ii) *Low compatibility with I/O calls*. While user-space memory within a transaction is under the STM system's control, memory in the kernel may not be. As a result, the atomicity and isolation properties are not automatically enforced for changes to kernel data structures. In addition, some I/O operations, such as printing a message onto screen, cannot be reversed on abort. So most STMs, such as Grace [11], Intel STM [25], and Haskell STM [26], simply prohibit using I/O calls within transactions. Analyses of multithreaded programs written with locks show that I/O calls are a regular occurrence in critical sections [20, 27]. Also, a study of real-world concurrency bugs shows that about 15% of concurrency bugs' recovery involves revoking I/O calls' effects [22]. Hence, forbidding I/O calls in transactions reduces the usability of STM and threatens its validity as a solution to concurrency bugs.

(iii) *Low compatibility with condition variables*. A condition variable (or "condvar") does not follow an atomical and isolated specification: an invocation of the wait method cannot return without paring with an intervening signal by another thread. When a transaction that executes wait/signal operations aborts, there is no safe way to revoke effects generated by them. For example, revoking/re-executing a not-finished-yet wait may lead to missing a signal signaled by another thread and further block the calling thread forever. So lots of STMs [11–19, 23, 28, 29] do not support using condvars in transactions. This makes STMs fail to avoid quite a few concurrency bugs [22] and has also been recognized as an obstacle to transactionalizing legacy code [22, 30, 31].

To overwhelm these deficiencies, this paper presents a new and optimized STM system Convoider, which tries to transparently transactionalize applications without any manual effort, avoid various types of concurrency bugs, and support revocable I/O as well as proper condvar handling simultaneously. Figure 1 overviews Convoider. It controls applications in both linkage and runtime phases.

For applying Convoider to an application, users first relink related object files, such as relocatable object files and shared object files, with the linker ld, against a customized linker script file, which is used to control the memory layout of the output executable file. By way of this control, Convoider specifies the start address where global data are stored in the address space. This information is then used at runtime to make global memory revocable.

At runtime, Convoider instruments five kinds of operations: interthread synchronization operations, memory access operations, input/output operations, condvar signal/wait operations, and lock/unlock operations. By intercepting interthread operations and designating code among them as transactions in each thread, Convoider automatically transactionalizes target programs without any source code modification and recompiling. By saving/restoring stack frames and CPU registers on beginning/aborting a transaction, Convoider makes execution flow revocable. By turning threads into processes, leveraging virtual memory protection, and customizing memory allocation/deallocation, Convoider makes memory manipulations revocable. By maintaining virtual file systems and redirecting I/O operations onto them, Convoider makes I/O effects revocable. By converting lock/unlock operations to no-ops, customizing signal/wait operations on condvars, and committing memory changes transactionally, Convoider makes deadlocks, data races, and atomicity violations impossible.

We have implemented Convoider in Ubuntu as a dynamic shared library for C/C++ applications. It runs with target applications in the same address space. We evaluated it against two benchmark suites: one real-world application suite including 12 real-world applications selected from PARSEC [32], SPLASH2 [33], and Phoenix [34] and one concurrency bug suite containing 31 concurrency bugs collected from Mocklinter [35], data-race-test [36], Maple [37], PSet [38], and Grace [11]. Evaluation results show that Convoider succeeds in automatically transactionalizing many more programs and perfectly avoiding many more concurrency bugs than Grace. Meanwhile, Convoider is efficient and only averagely incurs 28% runtime overhead to the target applications.

This paper makes contributions in the following points:

(1) An optimized STM system Convoider is proposed to transparently transactionalize multithreaded programs and avoid concurrency bugs.

(2) Revocable I/O support in Convoider enables making I/O calls in transactions.

(3) Proper condvar handling in Convoider allows using them in transactions.

(4) Experiments are carried out for evaluating Convoider's efficiency and effectiveness in transactionalizing real-world applications and avoiding concurrency bugs.

The rest of this paper is organized as follows: we review background works and describe our improvements on how to make execution flow and global/heap memory revocable in Section 2. We describe how to make I/O revocable in Section 3 and how to properly handle condvars in Section 4. Section 5 depicts the detailed execution of transactions. Section 6 evaluates Convoider and compares it with Grace [11], Dthreads [39], Dimmunix [40], and Slider [41]. Section 7 concludes this study.

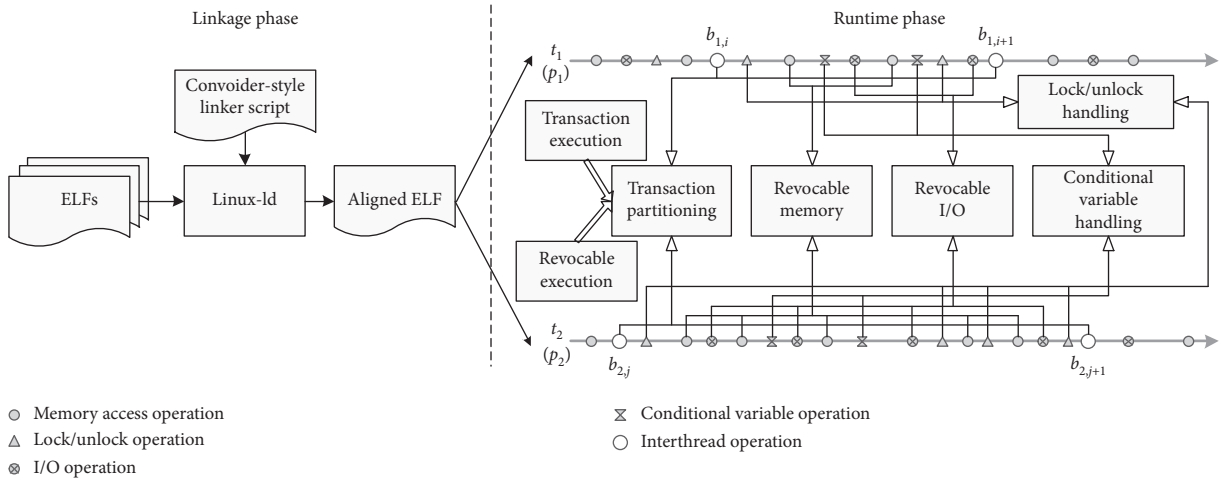

Figure 1: The overview of Convoider.

2. Background

Grace [11] is the first trial to automatically transactionalize traditional lock-based concurrent programs. It turns threads into processes, automatically partitions code into transactions, constructs a transactional memory by leveraging virtual memory protection, and avoids simple concurrency bugs. Dthreads [39] enhances Grace in adding support for proper condvar handling. Conversion [42] advances Dthreads by recreating an efficient virtual memory manager at the kernel level. We compare Convoider with Grace and Dthreads in Section 6. However, comparison between Convoider and Conversion is not made because they are not comparative in both implementation levels (user space level vs. kernel level) and usability (a shared library vs. an extended kernel).

We build Convoider on top of Grace [11] by reusing and enhancing its transaction partition, revocable execution, and transactional memory modules. This section will give background knowledge about how Grace demarcates transactions and makes execution flow and global/heap memory revocable, its deficiencies, and our improvements.

2.1. Transaction Partition and Revocable Execution. Grace intercepts thread creates and joins and designates code among them as transactions. Grace only targets fork-join concurrent programs. While Convoider aims at coping with more general concurrent applications, it also intercepts other interthread synchronization operations such as condvar wait, signal, and broadcast. For illustrating how Convoider automatically demarcates transactions, take t_1 as an example in Figure 1, which contains two immediate interthread operations $b_{1,i}$ and $b_{1,i+1}$. When $b_{1,i}$ is called, Convoider will try to commit t_1's current transaction and meanwhile start a new transaction, which lasts until $b_{1,i+1}$ is called.

Grace saves/restores stack frames and CPU registers on starting/aborting a transaction. Thus, when a transaction conflicts with other transactions and aborts, it can roll back to its begin point and try again. Grace uses a pair of functions getcontext and setcontext to get/set the execution context, which contains CPU registers such as program counter and stack pointer. However, these two functions do not save/restore stack frames. To do that, Grace itself copies the stack frames of the current thread to a private memory region when a transaction starts and writes back the frames if this transaction aborts. The written-back frames together with the restored registers make the transaction re-execute in the same execution context as it begins in its last execution. However, Grace only copies part of stack frames of the current thread, which would lead to stack smashing (This is why Grace fails to apply to string_match in Section 6.3) when the partly stored stack frames are written back. Instead Convoider saves the whole stack rather than part of it by leveraging portable routines __builtin_return_address and __builtin_frame_address. By saving/restoring complete stack frames on beginning/aborting a transaction, Convoider can correctly make execution flow revocable.

2.2. Revocable Memory. Grace implements an efficient software transactional memory by treating threads as processes: instead of spawning new threads, Grace forks off new processes. Because each "thread" is in fact a separate process, it is possible to use standard memory protection functions and signal handlers to track reads and writes to memory. Grace presumes that an application uses only global memory and heap memory to share data among threads and tracks accesses to memory at a page granularity.

2.2.1. Revocable Global Memory. Grace assumes the size of the global data in the target application would not exceed 100 MB. It uses a customized linker script (Figure 1) to locate the start address of the global data with a linker variable "gracestart." Meanwhile, a symbol "_end" in ELF executable indicates the address of the first byte after uninitialized global data section (namely, .bss section). Therefore, at runtime, memory area between these two address stores the global bytes. The linker script is also used to instruct the linker to

page align and separate read-only memory from global read/write memory.

When an executable is loaded, Grace creates a 100 MB region to hold the global memory and establishes two memory mappings, one shared and one local, for the region. It maintains a version region for the global region and uses a word in the former to track the changes of a page in the latter. For the version region, Grace also creates two mappings: one shared and one local. Each pair of mappings created for a global or version region is correlated with the same on-disk temporary file. This correlation can make the content of local mapping consistent with that of shared mapping by simply remapping the local mapping from the file.

The mechanism to make global memory revocable is illustrated in Figure 2. Suppose that currently threads t_1 and t_2 are concurrently executing transactions tx_a and tx_b, and tx_b starts before and ends after tx_a. When tx_a starts, it performs several memory operations. Each read/write operation accesses memory through the private mapping (step ①), which permits reads to directly read data from the shared memory but redirects writes to a copy page of the corresponding "wanted" page in the shared memory (steps ② and ③). On starting, a transaction gets a private mapping for the global memory. Each page's permission in the private mapping is set to PROT_NONE. So, the first access issued by tx_a to an PROT_NONE page causes Grace to add this page to tx_a's read set (in page granularity) and then set the page's permission to PROT_READ. If a subsequent access to this page is a write, Grace adds this page to tx_a's write set and sets its permission to PROT_READ| PROT_WRITE. Such later accesses to it will not trigger page faults and run at full speed.

Each time a transaction starts, Grace also creates for this transaction a local copy of the global version numbers (steps ④ and ⑤). When the transaction tries to commit, it will first check whether its read bytes are still valid by comparing version numbers of pages in its read set against the global corresponding version numbers (step ⑥). If every local version number is equal to its global counterpart, the check is passed and the transaction commits by doing the following things: for each page in the write set, (1) incrementing the global corresponding version number by one, and (2) copying its content to the counterpart page in the shared memory (steps ⑥, ⑦, ⑧, and ⑨). Otherwise, the transaction aborts by abandoning memory changes buffered in its write sets, rolling back to its begin point, and trying to execute again (step ⑩). In Figure 2, tx_a and tx_b both access a page p: tx_b reads it, while tx_a writes it. When tx_b starts, it finds that p's global version number is 4 and gets a local copy of it. Then tx_a starts, it sees the same version number 4 of p and also gets a local copy of it. However, tx_a commits before tx_b and change p's global version number from 4 to 5. When tx_b tries to commit, it finds that p's local version number is less than p's global one. In such a case, tx_b knows some other transaction has already updated p's content, so, it aborts and re-executes to read the up-to-date content.

2.2.2. Revocable Heap Memory. When a target application starts, Grace claims a fixed-size 512 MB region from OS for it to hold its heap data. Grace maintains a version region for the heap region and uses a word in the former to track the changes of a page in the latter. Also, Grace creates shared and local mappings for both the heap region and the version region. And, each pair of mappings is backed with the same on-disk temporary file.

Grace instruments all basic C/C++ memory malloc/free family functions in order to steer the target application's memory claim/reclaim requests to the 512 MB heap region mentioned above. Figure 3 depicts the heap memory management mechanism. Grace embeds memory management metadata structures into the heap memory. This organization elegantly solves the problem of revoking effects of memory allocations/deallocations. Grace rolls back memory allocations/deallocations just as it rolls back ordinary updates to heap data or global data (Section 2.2.1).

Grace manages the heap region based on Hoard [43] and Heap Layers [44]. It divides the heap region into 16 subheaps. Each thread uses a hash of its thread identifier to claim a subheap for satisfying memory malloc/free requests issued by that thread. The isolation of each thread's memory manipulations from the others' allows threads to run independently most of the time. Each subheap is initially set as a 1024 KB zone. Each zone has an associated 16 B arena, which stores three kinds of information: the size of remained memory in the zone, the start address of remained memory, and the pointer to the next zone. Memory in zones is allocated linearly: the first allocated object is followed by the second one and so on. Each memory object occupies bytes of a power of 2 (at least 8 B) and has an 8 B object header used to record the object's size and alignment information. Each subheap can have multiple zones, and they are linked with each other by using the next pointer in each zone's arena. As long as a thread does not exhaust memory in the current zone, it will run independently with any other thread. If running out of memory, it will obtain another zone from the global allocator, whose size is 1024 KB or bigger if the memory request asks for more bytes than 1024 KB (*zone*2 in Figure 3). Zones are allocated linearly from the heap region. This zone allocation strategy would make the current thread conflict with another thread only if that thread also runs out of memory during the same period. In this situation, two threads will both view the newly allocated zone as their own zone.

Because each allocated object's size is a power of 2, when an object of size sz is freed, it is reclaimed and inserted into the head of bin $\log_2 sz$. Each subheap has 29 bins and bin_n ($0 \le n < 29$) is a double-linked list of free objects that each of them has a size of $2^3 * 2^n$. Each chunk in a bin is a user memory object, not including the management information such as the object header (Figure 3). When the application requests a memory object of length *len*, Grace acts as follows:

(1) If *len* is not larger than 8, it searches in bin_0 for a free chunk. If there exists one, Grace allocates it out. Otherwise, the logic sets *len* to 8 and goes to (4).

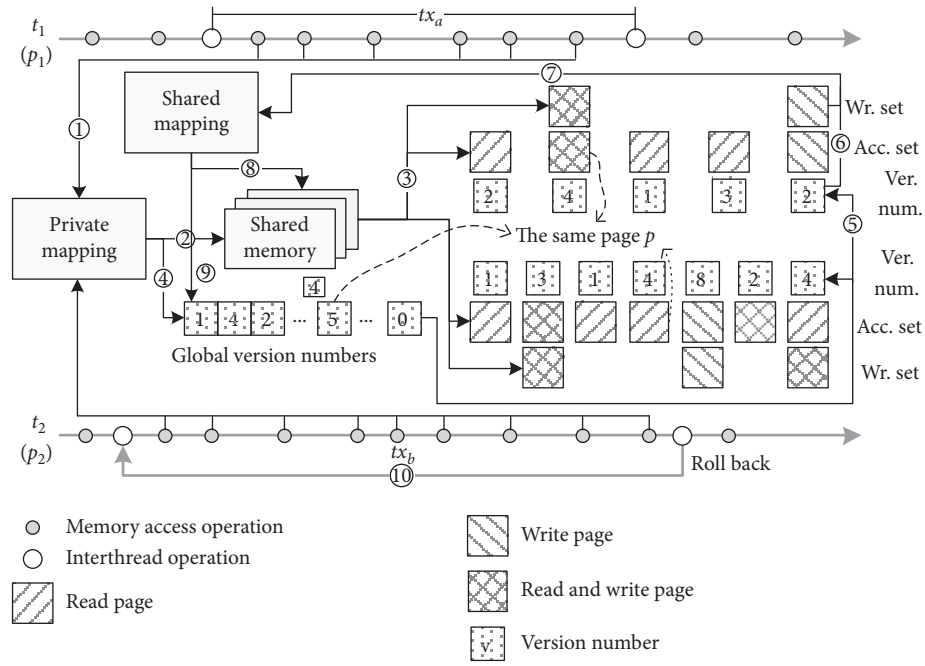

Figure 2: The revocable global memory mechanism.

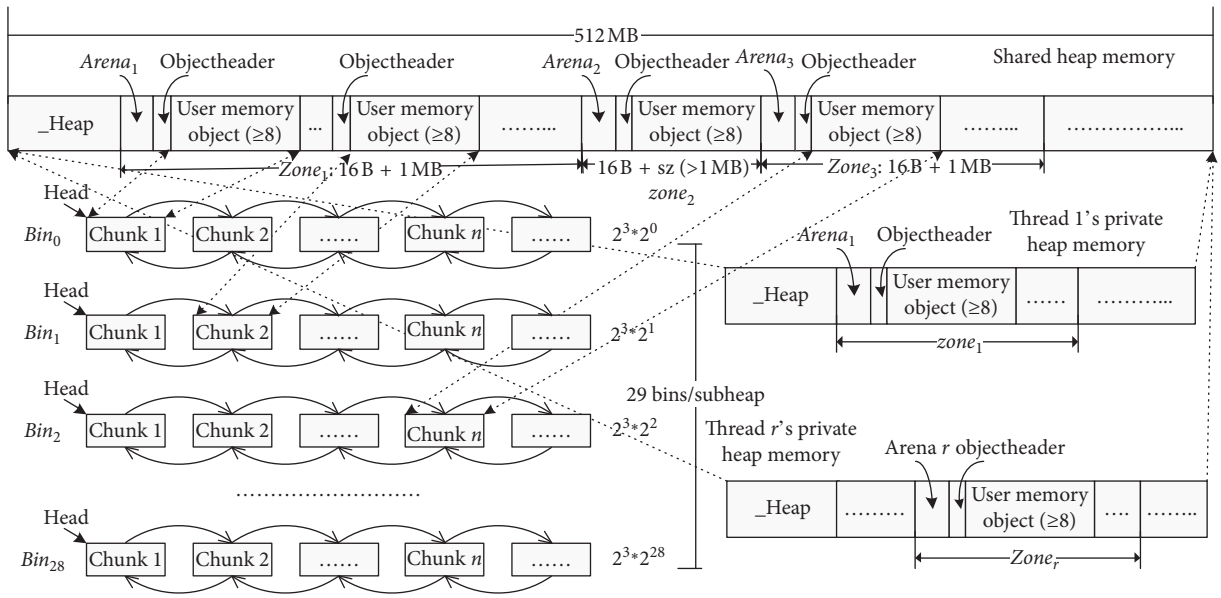

Figure 3: The heap memory management mechanism.

(2) If *len* is greater than or equal to 2^{31}, Grace reports an error.

(3) If $2^3 * 2^{m-1} < len \leq 2^3 * 2^m (1 \leq m < 29)$, Grace searches in bin_m for a free chunk. If there exists one, Grace allocates it to the application. Otherwise, the logic sets *len* to $2^3 * 2^m$ and goes to (4).

(4) Grace allocates the requested memory from the current thread's zone.

When a transaction begins, Grace creates for this transaction a local copy of the shared heap memory by way of private mapping. All memory malloc/free operations are actually redirected to this private copy and only change pages in the current thread's own zone. For example, in Figure 3, a transaction executed by thread *r* only mallocs/frees memory in zone *r*. When the transaction tries to commit, if Grace finds any page in its subheap is invalid, it rolls back memory allocations/deallocations for that transaction by simply discarding its write pages, recreating a local heap memory copy, and trying again. Currently, there are only 16 subheaps in heap memory. If two threads that execute two concurrent transactions are

mapped to the same subheap, they may conflict with high possibility.

Convoider corrects a severe bug in Grace where Grace unnecessarily and wrongly aligns allocated memory to page size on responding to memory allocation requests. The bug may make a newly allocated memory chunk by Grace overlap with a currently being used memory object, leading to memory crashing. (This is why Grace fails to apply to kmeans and reverse_index in Section 6.3.) Convoider fixes the bug by allocating memory continuously, namely, not aligning memory to page size.

3. Revocable I/O

Convoider instruments 84 commonly used I/O operations, as listed in Table 1, to provide transactional input/output support for regular files and character files at three levels: system level, C level, and C++ level. Convoider currently does not support revocable I/O for directory files, so it does not instrument directory I/O routines such as opendir, readdir, chdir, and closedir.

To enable performing I/O operations in transactions, researchers have developed three strategies: deferral [20, 27], namely, deferring I/O operations until commit, compensation [20, 27], namely, performing I/O operations as usual during the execution of transactions and reversing their side effects on transactions aborting, and irrevocability [18, 45], namely, ensuring that transactions with I/O operations will never abort. However, each strategy itself is imperfect [20]. When two operations are deferred, the OS may not be able to guarantee that both of them will succeed finally, leading to an inconsistent state. When system calls must be reversed on transaction abortion, the side effect revoking may fail. To guarantee successful commit, irrevocable transactions (those ones with I/O operations) cannot execute concurrently, causing downgrading performance. Convoider leverages a combined strategy to make I/O operations reversible. Convoider integrates deferral and compensation with a new strategy: exclusiveness. Under Convoider, multiple transactions with I/O operations run concurrently most of the time because they privatize I/O changes within transactions. However, if an I/O operation that may change the current process's table of open file descriptors is going to be performed, the current transaction enters an exclusive mode by prohibiting other transactions from executing such operations until itself commits or aborts. The prohibited transactions abort themselves when they are allowed to progress. Meanwhile, transactions that do not change the open file table can run concurrently with the exclusive transaction. The exclusive transaction still could abort because it may read out-of-date memory or access files that have been changed by other already-committed transactions. In this case, Convoider will revoke the effects of operations already performed by the exclusive transaction.

I/O operations in C/C++ programs can be categorized into three levels: system level, C level, and C++ level. At the system level, I/O operations manipulate files through file names or file descriptors. For confining effects of I/O operations within a transaction before the transaction commits, Convoider creates for the transaction a private copy of the global virtual file system when the transaction starts and redirects all system level I/O operations onto it while the transaction executes. The global virtual file system is created after the launch of the target application and shared among processes forked afterwards. When a transaction tries to commit, it must check whether the global counterparts of files in its private file system have not been changed by other transactions.

Table 1: I/O operations instrumented by Convoider.

Level	Operation
System	open, close, creat, write, read, lseek, pread, pwrite, dup, dup2, fsync, fdatasync, sync, fcntl, ioctl, stat, fstat, lstat, access, umask, chmod, fchmod, chown, fchown, lchown, truncate, ftruncate, link, unlink, symlink, readlink, utime
C	remove, rename, printf, puts, scanf, fscanf, vscanf, vfscanf, fflush, __isoc99_scanf, __isoc99_fscanf, __isoc99_vfscanf, fwide, setbuf, __isoc99_vscanf, setvbuf, fopen, freopen, fdopen, fclose, getc, fgetc, getchar, ferror, feof, clearerr, ungetc, putc, fputc, putchar, fgets, gets, fputs, perror, fread, fwrite, ftell, fseek, rewind, ftello, fseeko, fgetpos, fsetpos, fprintf, vprintf, vfprintf, fscanf, vscanf, vfscanf, fileno, tmpfile, mkstemp

At the C level, I/O operations access file through file streams. Convoider maintains a global stream-to-descriptor map that takes the file stream pointer as key and the file descriptor as the value. When a transaction starts, Convoider creates for it a private copy of the global map. When C I/O operations are going to be performed, Convoider leverages this private map to make them access files directly through file descriptors instead of file streams. According to C standards, a file stream can be associated with multiple (only one at a time) file descriptors. A transaction may switch a stream from an old file to a new file, while another transaction could still use that stream to refer to the old file. So on committing, the transaction has also to check whether entries in its private map are equal to their counterparts in the global map.

C++ level I/O operations based on I/O operations listed in Table 1 are automatically revocable because their underlying operations are revocable. So, by making system I/O and C I/O operations revocable, Convoider makes C++ I/O operations revocable.

3.1. Revocable System I/O. Convoider makes system I/O operations revocable by way of deferral, compensation, or exclusiveness strategies, depending on their semantics. Besides these strategies, Convoider also allows immediate execution of read-only operations, such as read and access, or sets some operations as no-ops, like sync, fsync, and fdatasync. Table 2 categories system I/O operations into five groups according to the strategy used to make them revocable. Note that fcntl falls into different categories because it has different operation semantics with different commands. For example, if having F_DUPFD as command, fcntl may

Table 2: Categories of system I/O operations.

Strategy	Operation
No-ops	sync, fsync, fdatasync, ioctl, fcntl
Read-only	read, pread, access, utime, readlink, stat, fstat, lstat, fcntl
Deferral	close, write, lseek, pwrite, fcntl, umask, chmod, fchmod, link, chown, fchown, lchown, truncate, ftruncate, unlink, symlink
Exclusiveness	open, creat, dup, dup2, fcntl
Compensation	open, creat, dup, dup2, fcntl

change the table of open file descriptors, so it falls into the "Exclusiveness" group. However, if given F_GETFL as command, fcntl will belong to the "Read-only" group because it just reads file flags. In addition, Convoider will set fcntl as no-op for commands that it does not support.

3.1.1. Virtual File System. Convoider turns thread into process. So each thread is actually a process. In Linux, each process has its own file descriptor table, maintained by the kernel. Associated with each file descriptor are a file descriptor flag and a pointer to an open file table entry. The open file table is maintained for all open files and each entry in it has three fields: a file status flag (O_RDONLY, O_APPEND, and so on), a current file offset, and a pointer to an entry in the v-node table. Each open file has a v-node structure that indicates the type of file and other information. The v-node also contains an i-node for the file, which contains the owner of the file, the size of the file, and so on. For a process that forked from the main process, the forked process inherits its parent's file descriptor table.

However, if the forked process opens a file, reads or writes it, there is no way to know whether the other process has accessed that file or not. To catch such information, Convoider maintains a virtual file system. Because Convoider only cares about regular files and character files, not supporting revocable I/O for directory files, the virtual file system is in fact a vector of virtual files. As shown in Figure 4, each virtual file contains a version number, indicating the number of commit times for this file, and some file attributes, including file name, file descriptor, file mode (indicating file's type and access permission), file status flag, file size, current offset, file owner, and file group owner. Besides these two fields, a private virtual file also contains two additional parts: a write list, used to buffer write bytes, and special virtual file records, including flags that indicate whether this virtual file is opened, closed, created, linked, unlinked, or symbol-linked, and strings, used to buffer the new link or symbol-link names. A virtual file's attributes are obtained from the real-file system when the real file is created or opened.

When an application is launched and about to execute, Convoider establishes a global virtual file system for it by creating virtual files for standard input/output/error files and inserting them to the file system. Later, whenever a transaction starts, Convoider creates a private virtual file system for it by duplicating the global one and uses this private file system to buffer changes for that transaction.

3.1.2. Deferral and Read-Only Strategies. Figure 4 shows the mechanism that makes system I/O revocable. As shown in Figure 4, a process gets a private virtual file system when it begins to execute a transaction (step ①). For an I/O operation that writes bytes to a file or changes file attributes, Convoider buffers its write bytes or changes onto the private file system until the current transaction commits or aborts (step ②). For an I/O operation that reads bytes from a file, Convoider first reads specified bytes from the real file (step ③) and then checks whether there is an overlapping between the read bytes and the buffered write bytes of the virtual file corresponding to the real file. If true, Convoider replaces the read bytes in the overlapping range with the corresponding write bytes. At last, the adjusted bytes are returned as result. For an I/O operation that retrieves file attributes, Convoider directly returns the corresponding virtual file's attributes as result instead of accessing the real file (step ②). On committing, a transaction checks whether its private files are consistent with the global ones by comparing files' versions. If they all match with each other (step ④), the transaction does three things for each virtual file in its virtual file system: replacing the corresponding global file's attributes with this virtual file's attributes (step ⑤), writing buffered bytes or attribute changes into the real file (step ⑥), and incrementing the corresponding global file's version by one (step ⑤).

3.1.3. Exclusiveness and Compensation Strategies. I/O operations that open, create, or duplicate files, such as open, creat, and dup (Table 1), cannot be delayed during the execution of a transaction. However, if different processes concurrently execute these operations, some process may get a private virtual file system that is inconsistent with its real file system.

To keep file systems of different processes consistent with each other, Convoider makes these I/O operations execute exclusively: in a transaction, once any such operation is going to be executed, the transaction acquires an interprocess lock if the lock has not been acquired and executes that operation under protection. This lock is held by the current process until the transaction commits or aborts. Thus, other transactions that execute such operations are all blocked from making progress. Between when the lock is acquired and when it is released, Convoider uses a list of interprocess file information objects to record each such operation's operation type and parameters. When the transaction eventually commits, for each object in the list, it sends to all other active processes a signal, taking the object as the signal's accompanying data, to tell them what operation has been performed in this transaction. However, these signals are not immediately received by the target processes. They are received and handled only when the targeted processes start/re-execute transactions. For a blocked process by the interprocess lock, if it is allowed to continue, it aborts its current transaction and re-executes the transaction again to receive signals. For an unblocked process, it executes its current transaction as usual. If the transaction finally aborts, the unblocked process re-executes it and

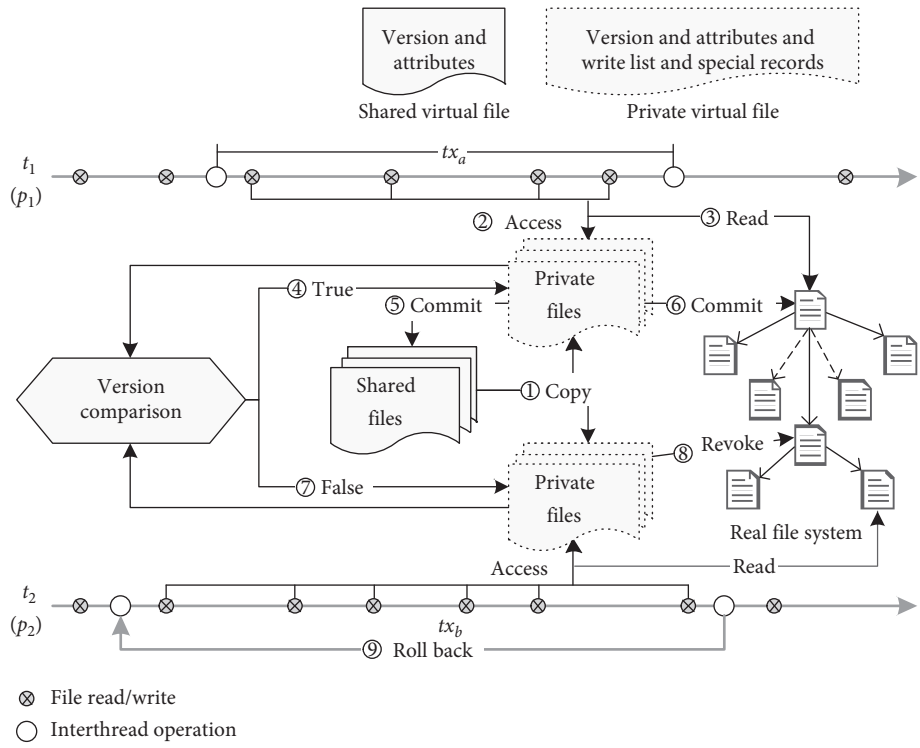

Figure 4: The mechanism to make system I/O revocable.

receives signals. Otherwise, the process receives signals when it begins a new transaction. Once a signal is received by a target process, the process carries out the operation with the parameters indicated in the accompanying file information object. After handling signals, the process gets a private copy of the global file system. This time, the private virtual file system is consistent with the underlying real file system.

When a transaction executes such operations exclusively, it opens or creates files in the real file system and also creates virtual files in its virtual file system. When the transaction commits, it compares all files except newly opened or created files in its private file system with their global counterparts. If they do not match all (step ⑦ in Figure 4), the transaction cancels out effects of such operations by closing or deleting corresponding files in the real file system (step ⑧) and rolls back to its starting point (step ⑨).

3.2. Revocable C I/O. Convoider instruments C I/O operations shown in Table 1 and reimplements them using revocable system I/O operations described in Section 3.1. For example, Convoider uses open and close to implement mkstemp and tmpfile, and utilizes link, unlink, and close to implement remove and rename. C I/O operations that take either file names or nothing as input can be simply reimplemented based on system I/O operations.

However, most C I/O operations manipulate files through file streams, which are generated by fopen, freopen, or fdopen. For these operations, Convoider cannot directly overwrite them using system I/O operations. To solve this problem, Convoider maintains a global map that each entry maps a file stream pointer to a file descriptor. When a transaction starts, Convoider creates for it a private copy of this global map. This transaction may use fopen or fdopen to create a new stream, or use freopen to change a stream's associating file, or use fclose to close a stream, or use other stream-based operations to access files. Convoider instruments fopen as follows: (a) calling open to open a specified file and return the file's descriptor, (b) converting the descriptor to a stream pointer and creating an entry to establish a mapping from this pointer to the descriptor, (c) inserting the entry to the private map of the current transaction, and (d) returning the pointer as a result. For fdopen, Convoider instruments it with the latter three steps used to instrument fopen. For freopen, Convoider reimplements it in a similar way to re-implementing fopen, except that instead of creating a new mapping entry, Convoider modifies a corresponding existing entry in the private map. For fclose, Convoider reimplements it by removing from the private map an entry taking the stream pointer specified in fclose as key. For other operations, Convoider leverages the private map to make them access files directly through descriptors instead of streams. This disables stream buffering, so Convoider sets operations setvbuf, setbuf, fwide, ferror, and clearerr as no-ops. When the transaction commits, it needs to check whether or not each entry in its private map matches with the corresponding global counterpart. Only when they all match, the transaction can successfully commit. The committing is simply done by copying contents of the private map back to the global map.

4. Condition Variable Handling

Convoider instruments all condvar operations and reimplements condvars in user space in a similar way to the

method of Wang et al. [46]. However, Convoider's solution is more portable because of needing no changes to condvar's programming interfaces.

When a condvar's initialization operation is performed, Convoider allocates a page memory from the shared memory pool reserved for condvars and stores the begin address of this page into the first word of the current condvar. All subsequent operations to the condvar are redirected onto this page. As shown in Figure 5, this page p is organized into 512 slots, one double-word per slot. These slots are further separated into two groups: the control group and the waiting thread group. The former group contains two slots: $slot_0$ and $slot_1$. The first word $p[0]$ of $slot_0$ is used to implement a customized lock: 1 indicating unlocked and 0 representing locked. Any access to this page should be performed under protection by this lock. The second word $p[1]$ indicates the position of the next available slot. The value of $p[1]$ monotonically increases from 0 and wraps around when it exceeds 511. For $slot_1$, its first word records the number of threads that waits on this condvar and its second word is unused. The remaining slots fall into the second group. Each slot in this group is a tuple $<tid, bsignaled>$, where tid represents the waiting thread and $bsignaled$ indicates whether this thread has been signaled by this "paged" condvar.

When a condvar's wait is performed, Convoider instruments it as follows: (a) committing the current transaction, (b) acquiring the lock of the paged condvar $p[0]$, (c) incrementing the number of waiting threads $p[2]$ by one, (d) setting the slot indicated by $p[1]$ to $<self, \text{false}>$, where $self$ represents the current thread's identifier, (e) spin waiting until the current thread is signaled on this condvar by another thread, (f) decrementing $p[2]$ by one, (g) releasing the lock $p[0]$, and (h) starting a new transaction.

When a condvar's signal is performed, Convoider instruments it as follows: (a) committing the current transaction, (b) acquiring the page lock, (c) checking whether there is any waiting thread, (d) if there is any one, randomly selecting a waiting thread and signaling it by setting its $bsignal$ to true, (e) releasing the lock, and (f) starting a new transaction. The broadcast operation of condvar is also instrumented in such a way, except that all waiting threads are signaled instead of only one thread being signaled.

When a condvar's destroy operation is performed, Convoider reclaims the page allocated for the condvar which is specified in the destroy's parameter.

5. Transaction Execution

A transaction can start, commit, or abort. When an application is going to execute, Convoider kicks off the first interesting transaction for it after necessary initializations are finished. Necessary initializations include saving execution context, creating memory mapping, creating version mapping and establishing virtual file system and stream-to-descriptor map.

On starting, a transaction first checks whether there is any blocked signals sent by other transactions. If true, the transaction receives and handles them as described in Section 3.1.3. Then, this transaction saves current stack frames and CPU registers, creates private memory/version mappings, and establishes the private virtual file system and stream-to-descriptor map from their corresponding global ones. The transaction also sets the protection of each page in its private mappings to PROT_NONE and clears its read/write page set. Additionally, for each file in the private virtual file system, this transaction initializes its special records (Section 3.1.1) with proper values and applies its attributes to the corresponding real file, thus keeping the real file consistent with the virtual file.

On committing, a transaction first performs consistency checks for memory and files it accesses during its execution. If all checks are passed, this transaction commits memory and files as described in Sections 2.2 and 3. Each transaction commits independently, needing not to wait for descendant transaction committing first. Due to committing transactions in no order, Convoider cannot prevent order violations from happening in one hundred percentage.

On aborting, a transaction discards any memory updates by calling madvise function with advice MADV_DONT NEED for all of the private mappings, and it abandons any buffered writes by clearing the write list of each file in the private file system. The transaction also empties the list of interprocess file objects and closes/unlinks any files opened/created during its execution. Then, it releases the interprocess lock mentioned in Section 3.1.3 if the lock is held. At last, the transaction restores stack frames saved on starting for the current process and rolls back to the start point.

6. Evaluation

We have implemented Convoider on Ubuntu-12.04 as a dynamic shared library. Its goal is to transparently transactionalize multithreaded programs at no human cost, meanwhile providing revocable I/O support for regular/character files, handling condvars properly, and avoiding concurrency bugs like deadlocks, data races, and atomicity violations. We evaluate Convoider against a real-world application suite and a concurrency bug suite to answer the following research questions:

(i) How effectively does Convoider transparently transactionalize real-world applications?

(ii) How efficiently does Convoider transparently transactionalize real-world applications?

(iii) How effectively does Convoider avoid concurrency bugs?

6.1. Experimental Setup. We perform our evaluation on a machine with Intel Core 2 Q8200 CPU and 2 GB RAM. This CPU has 4 cores, each of which runs at 2.33 GHz frequency and is equipped with a 4 MB L2 cache and a 4 × 64K L1 cache. The machine is running an OS of 32-bit Ubuntu 12.04 with kernel version 3.2.0. All avoiding tools and all benchmark programs are compiled with GCC 4.6.3 with the same level optimization.

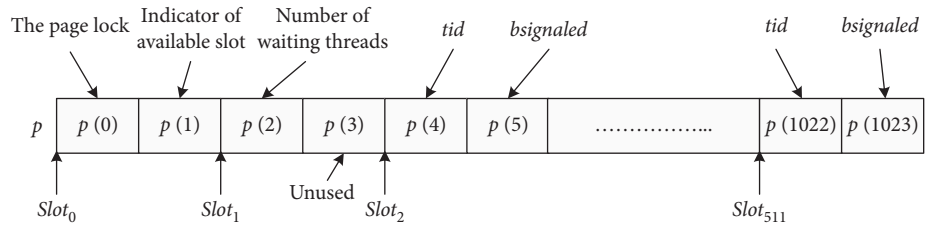

Figure 5: The condvar page organization.

6.2. Applicability. According to Sections 2.2, 3, and 4, Convoider works under three main assumptions: the target program (a) only consumes memory no more than 100 MB for global data and 500 MB for heap data (b) only uses I/O operations listed in Table 1 (or C++ I/O operations based on these operations), to access regular/character files, and (c) only uses pthread mutual exclusive locks, read/write locks, and condition variables to synchronize among threads. We test Convoider on applications collected from STAMP [47], PARSEC [32], SPLASH2 [33], and Phoenix [34] to evaluate its applicability.

STAMP is a benchmark suite designed for transactional memory research. It consists of eight macro applications and several microapplications, such as rbtree, list, and hashtable. All these applications are originally instrumented with low-level transactional memory API calls. However, Convoider is designed to transactionalize applications without needing any instrumentation. So, Convoider is not applicable to any STAMP application if we do not manually remove the transaction memory API instrumentation and convert these STM-based applications into lock-based applications.

PARSEC is a benchmark suite for shared memory on-chip multi-processor architectures and contains thirteen multithreaded and memory-intensive applications. In all applications, five applications (fluidanimate, raytrace, bodytrack, canneal, and streamcluster) use pthread barriers for synchronization; thus, Convoider is not applicable to them. Among other applications, one application (freqmine) is written with OpenMP (instead of pthread library) and four applications' (facesim, vips, x264 and ferret) linking procedures are too complicate for Convoider to cope with. For the remainder three applications (blackscholes, dedup, and swaptions), Convoider is only applicable to swaptions. Convoider cannot transactionalize dedup because it requires an amount of memory exceeding Convoider's limits. When running with Convoider, dedup reports a memory allocation failure warning and terminates. Convoider cannot transactionalize blackscholes because it smashes stack when running with Convoider.

For SPLASH2, we test Convoider on its 4 kernel applications: lu, fft, radix, and cholesky. These applications are also memory-intensive and CPU-intensive, and they synchronize among threads by using pthread mutexes and condvars. Results show that Convoider can work well with lu, fft, and radix but does not work with cholesky. For cholesky, Convoider is applicable to its serial version. However, for multithreaded cholesky, Convoider causes it to be blocked after it partitions the input into blocks and launches multiple threads to deal with these blocks. After careful check, we find that, besides standard pthread synchronization facilities, Cholesky also uses ad hoc synchronization constructs (for example, reading/writing shared flags) to synchronize threads. Convoider privatizes memory changes into a transaction until it commits. If two threads that run two transactions use shared flags to communicate, the read thread may wait forever because it cannot see the new value updated by the write thread without aborting/re-executing.

We also test Convoider on Phoenix which totally contains eight applications: histogram, kmeans, liner_regression, matrix_multiply, string_match, word_count, reverse_index, and pca. All applications are memory-intensive and CPU-intensive. Two of them synchronize with pthread mutexes. Testing results show that Convoider is applicable to all applications. Note that although the reverse_index application performs directory manipulation operations such as opendir and readdir, however, Convoider still can be applied to it because those operations are called in the first transaction which will definitely succeed in committing.

Table 3 lists all twelve applications with which Convoider can work. Among these applications, swaptions is written in C++ and others are written in C. Although all applications are multithreaded and memory-intensive, seven of them (swaptions, histogram, kmeans, linear_regression, matrix_-multiply, string_match, and word_count) are embarrassing parallel, meaning that there is no memory shared among threads and each thread runs independently without synchronizing with others. And, although all applications read/write files, only five of them (lu, fft, radix, string_match, and reverse_index) perform concurrent I/O operations, meaning that file accesses are carried out in threads that may run simultaneously.

6.3. Efficiency on Real-World Applications. In this section, we evaluate Convoider's performance on twelve real-world applications listed in Table 3. For each application, we compare Convoider with pthread, Grace [11], and Dthreads [39] on the speedup incurred over the sequential version of the application. Dthreads is a deterministic concurrency system and an improver of Grace. Dthreads improves Grace in handling conditional variables, just as Convoider, but in a different way. Not like Convoider, Dthreads neither supports revocable I/O nor avoids concurrency bugs. The input parameter for each application has two versions: sequential

Table 3: Applications to which Convoider is applicable.

Benchmark suite	Application	Language	Synchronization	Shared memory	Concurrent IO
PARSEC-3.0	swaptions	C++	None	No	No
SPLASH2	lu	C	mtx, cond	Yes	Yes
	fft	C	mtx, cond	Yes	Yes
	radix	C	mtx, cond	Yes	Yes
Phoenix	histogram	C	None	No	No
	kmeans	C	None	No	No
	linear_regression	C	None	No	No
	matrix_multiply	C	None	No	No
	pca	C	mtx	Yes	No
	string_match	C	None	No	Yes
	word_count	C	None	No	No
	reverse_index	C	mtx	Yes	Yes

and multithread, as shown in Table 4. For each application, the pthread-version, Grace-version, Convoider-version, and Dthreads-version share the same multithread input parameter.

Figure 6 displays experiment results in the form of histogram. Values greater than 1 indicate better performance than the sequential version. From Figure 6, we see that Convoider performs worse than pthread and better than the sequential execution. Averaged over twelve applications, the speedup ratio between pthread and Convoider is 2.58 : 1.86, and, namely, Convoider incurs 28% runtime overhead to the original multithreaded applications. Although Convoider brings large slowdown for the target applications, it succeeds in automatically transactionalizing all of them with almost no human efforts and runs them correctly without wrong outputs or crashes. Among twelve applications, Convoider can be directly applied to swaptions, lu, fft, radix, kmeans, and pca. For the remainder, we need to make minor (one or two) modifications to replace file mapping operation mmap with two operations: memory allocation malloc and file access operation read/write. Each such modification only needs three lines of codes. Convoider does not intercept mmap and has no perception of memory allocated by it, so without such modifications Convoider will not provide revocability for the memory.

In contrast, Grace is only perfectly applicable to two applications: swaptions and pca. For these two applications, Grace incurs about the same speedup as Convoider: the ratio is 3.19 : 2.85 averagely. Grace performs a little better than Convoider because it does not support revocable I/O while Convoider provides revocable I/O support for three system files: STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO and three file streams: stdin, stdout, and stderr. For histogram, linear_regression, matrix_multiply, and word_count, although Grace runs them without runtime errors or crashes, they terminate with wrong running results. So, the speedup incurred by Grace for these applications may be overestimated. For the other applications, Grace makes lu, fft, and radix hang forever because they synchronize with condition variables that Grace does not handle. For kmeans or reverse_index, Grace causes them to crash because of failed assertions or illegal memory accesses. Grace's problematic memory management is to blame for this crash. Lastly, Grace is also not applicable to string_match because when running with Grace, this application instantly terminates without any meaningful output.

Dthreads is perfectly applicable to nine applications: swaptions, histogram, kmeans, linear_regression, matrix_multiply, pca, string_match, word_count, and reverse_index. For these nine applications, Dthreads runs 7.3% slower than Convoider: the speedup ratio is 1.78 : 1.92. Dthreads performs a litter worse than Convoider because it uses a single global token to guarantee determinism. In Dthreads, on performing a lock/unlock or condvar wait/signal operation, a thread is asked to first exclusively acquire the global token, leading to performance downgrading. For lu, fft, and radix, Dthreads makes them hang forever although it is declared to be able to handle locks and conditional variables properly.

6.4. Effectiveness on Avoiding Concurrency Bugs. To evaluate Convoider's concurrency bug avoidance capability, we create a bug suite including totally thirty-one concurrency bugs which are classified into four categories: deadlock, data race, atomicity violation, and order violation. All bugs are listed in Table 5, together with their corresponding buggy programs, sources, descriptions, and categories. Among these bugs, fifteen bugs (bug#5, bug#6, bug#18, and bug#20–bug#31) are drawn from actual bugs described in the previous work on concurrency bug detection [35, 37] and avoidance [11, 38], while two bugs (bug#3 and bug#4) are real bugs without any changes. Our suite also includes four toy bugs [11, 35, 37, 38] (bug#1, bug#2, bug#16, and bug#17) created for research purpose. The remainder nine bugs (bug#7–bug#15 and bug#19) are all collected from Google's data-race-test suite. Because concurrency errors are by their nature nondeterministic and occur only for particular thread interleaving, we insert delays (via usleep) at key points in the code. The delays makes these bugs occur in an almost 100% likelihood. Thus, all bugs can be easily triggered by directly running the corresponding buggy programs, except bug#3 and bug#4. For these two real bugs, we write and run triggering code to trigger them.

Table 4: Input parameters for 12 real-world applications.

Application		Input parameter
swaptions	Sequential	-ns 32 -sm 10000 -nt 1
	Multithreaded	-ns 32 -sm 10000 -nt 4
lu	Sequential	-n2048 -b16 -p1
	Multithreaded	-n2048 -b16 -p4
fft	Sequential	-m20 -n65536 -l4 -p1
	Multithreaded	-m20 -n65536 -l4 -p4
radix	Sequential	-n33554432 -r8 -m524288 -p1
	Multithreaded	-n33554432 -r8 -m524288 -p4
histogram	Sequential	./histogram_datafiles/small.bmp
	Multithreaded	./histogram_datafiles/small.bmp
kmeans	Sequential	-d 400 -c 20 -p 100000 -s 10
	Multithreaded	-d 400 -c 20 -p 100000 -s 10 -t 4
linear_regression	Sequential	./linear_regression_datafiles/key_file_100MB.txt
	Multithreaded	./linear_regression_datafiles/key_file_100MB.txt
matrix_multiply	Sequential	512
	Multithreaded	-r 512 -t 4 -c 1
pca	Sequential	-r 800 -c 640 -s 200
	Multithreaded	-r 800 -c 640 -s 200 -t 4
string_match	Sequential	./string_match_datafiles/key_file_100MB.txt
	Multithreaded	./string_match_datafiles/key_file_100MB.txt
word_count	Sequential	./word_count_datafiles/word_50MB.txt
	Multithreaded	-f ./word_count/datafiles/word_50MB.txt -d 10 -t 4
reverse_index	Sequential	./testdir1
	Multithreaded	./testdir1

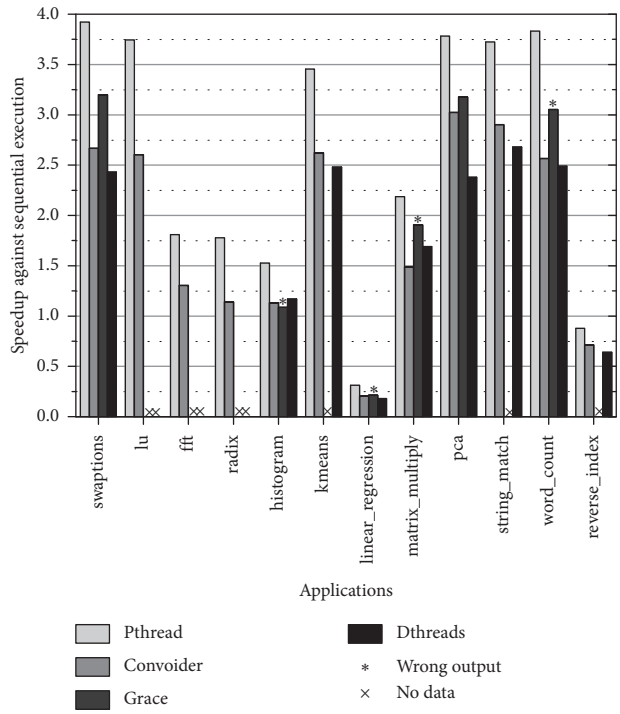

Figure 6: Performance of real-world applications running with Pthread, Convoider, Grace, and Dthreads.

In our experiments, we apply Convoider as well as other avoidance tools on these bugs. Given a bug, we apply a tool on it 10 times. Among 10 times of avoidance testing, only if one time the tool fails to avoid that bug, we say that the tool cannot avoid that bug.

6.4.1. Effectiveness on Avoiding Deadlocks. We uses Convoider to avoid deadlock bugs bug#1–bug#9 and compare Convoider's avoidance capability with three state-of-the-art deadlock avoidance tools: Grace [11], Dimmunix [40], and Slider [41]. We do not compare Convoider with Sammati

Table 5: 31 concurrency bugs and their related information.

Bug#	Buggy program	Description	Category
1	bank-transaction [35]	Two threads reversely acquire two mutexes	Deadlock
2	dining-philosophers [35]	Five threads reversely acquire five mutexes	
3	sqlite-3.3.3 [35]	http://www.sqlite.org/src/info/a6c30be214	
4	hawknl-1.6b3 [35]	nlShutdown() and nlClose()	
5	openldap-3494-kernel [35]	http://www.openldap.org/lists/openldapbugs/200501/msg00101.html	
6	mysql-37080-kernel [35]	http://bugs.mysql.com/bug.php?id=37080	
7	deadlock-test 01 [36]	Similar to bug#1	
8	deadlock-test 02	Similar to bug#2	
9	deadlock-test 03	Two threads wait on two conditional variables	
10	race-test 01	Concurrent writes	Data race
11	race-test 09	Unsynchronized read and write	
12	race-test 20	Concurrent writes with timeout synchronization	
13	race-test 46	Concurrent reads/writes because of incorrect locking	
14	race-test 47	Similar to bug#13	
15	race-test 50	Concurrent writes because of incorrect signal/wait	
16	logprocsweep	Accesses on a shared variable with no lock protection	
17	shared-counter	Similar to bug#16	
18	mysql-3596-kernel [11]	http://bugs.mysql.com/bug.php?id=3596	Atomicity violation
19	atm-violation-test01 [36]	Similar to bug#20	
20	stringbuffer-jdk-1.4	https://github.com/jieyu/concurrency-bugstree/master/stringbuffer-jdk1.4	
21	mysql-169-kernel	http://bugs.mysql.com/bug.php?id=169	
22	apache-21287-kernel	https://bz.apache.org/bugzilla/show_bug.cgi?id=21287	
23	apache-25520-kernel	https://bz.apache.org/bugzilla/show_bug.cgi?id=25520	
24	apache-45605-kernel	https://bz.apache.org/bugzilla/show_bug.cgi?id=45605	
25	cherokee-0.9.2-kernel	A local read/write sequence interleaves with a remote one	
26	memcached-127-kernel	Similar to bug#25	
27	mysql-12848-kernel	http://bugs.mysql.com/bug.php?id=12848	
28	mysql-791-kernel	http://bugs.mysql.com/bug.php?id=791	
29	httrack-3.43.9-kernel	Reference may come before definition	Order violation
30	transmission-1.42-kernel	Similar to bug#29	
31	mozilla-0.8-kernel	Similar to bug#29	

[48] because it can only be compiled and run in a 64-bit environment while our platform is a 32-bit OS. However, as far as we know, from [48], Sammati can only avoid mutex deadlocks and cannot provide revocability for I/O operations. The experimental results are shown in Table 6, where the second column lists the types of the corresponding deadlock bugs.

As seen from Table 6, Convoider ties for strongest with Slider in deadlock avoidance capability among four tools: all deadlock bugs except bug#9 are perfectly avoided and the buggy programs terminate with expected behaviors. Comparatively, Dimmunix and Grace only perfectly avoid five and three bugs, respectively. Dimmunix is an offline deadlock avoider. It can only avoid deadlocks caused by mutexes, such as bug#1, bug#3, bug#4, bug#7, and bug#8. However, although bug#2 is caused by mutexes, Dimmunix fails to avoid it. We carefully check Dimmunix's source code and find its implementation for the lock-free queue contains data race bugs, which make Dimmunix fail to avoid bug#2. Grace avoids deadlocks by nullifying lock/unlock operations on mutexes/rwlocks, the same as Convoider does. However, different from Convoider, Grace does not support revocable I/O or proper condvar handling. This causes that Grace cannot perfectly avoid bugs such as bug#1 and bug#4–bug#6. For these bugs, their corresponding buggy programs terminate normally when running with Grace but with wrong outputs. The reason is that threads involved in any such bug all print out messages onto screen during the construction of the bug. This will lead to conflicts when threads commits transactions. In such case, Grace will roll back victim threads but cannot revoke effects of print operations executed by those threads. So, under Grace, these deadlocks are avoided but wrong output is generated. For Grace, we also find with surprise that Grace fails to avoid bug#2, a simple mutex deadlock: the buggy program cannot terminate when running with Grace. It seems likely that Grace has trouble in revoking memory effects when rolling back victim threads involved in this bug.

We note that all tools cannot avoid bug#9. As shown in Figure 7, this bug actually is an acyclic deadlock [49]. Each

Table 6: Deadlock avoidance results by Convoider, Grace, Dimmunix, and Slider (✔: success; ✘: failure; ✿: wrong output).

Bug	Deadlock type	Avoiding result			
		Convoider	Grace	Dimmunix	Slider
Bug#1	mutex deadlock	✔	✿	✔	✔
Bug#2	mutex deadlock	✔	✘	✘	✔
Bug#3	mutex deadlock	✔	✔	✔	✔
Bug#4	mutex deadlock	✔	✿	✔	✔
Bug#5	mixed deadlock	✔	✿	✘	✔
Bug#6	rwlock deadlock	✔	✿	✘	✔
Bug#7	mutex deadlock	✔	✔	✔	✔
Bug#8	mutex deadlock	✔	✔	✔	✔
Bug#9	acyclic deadlock	✘	✘	✘	✘

t_1	t_2
L_1: pthread_mutex_lock(&mtx_1);	L_6: pthread_mutex_lock(&mtx_2);
L_2: while (...){	L_7: while (...){
L_3: pthread_cond_wait(&cv_1, &mtx_1);	L_8: pthread_cond_wait(&cv_2, &mtx_2);
L_4: }	L_9: }
L_5: pthread_mutex_unlock(&mtx_1);	L_{10}: pthread_mutex_unlock(&mtx_2);

Figure 7: The pseudocode of bug#9: two threads wait on two condvars.

thread is waiting on a condvar, but there are no threads to send signals onto these condvars; thus, each thread will wait forever. For this bug, because Dimmunix and Grace do not handle condvars, they can do nothing to avoid the bug. Although Convoider and Slider rewrite condvars in user space, they keep the rewritten condvars the same semantics as the original ones. So, they either cannot avoid this bug.

We conclude that, at least to deadlock bugs listed in Table 6, Convoider can successfully avoid deadlocks except the acyclic ones and is one of the most powerful deadlock avoidance tools.

6.4.2. Effectiveness on Avoiding Data Races. We apply Convoider on data races bug#10–bug#17 and compare Convoider with Grace according to their avoidance capabilities. For these races, each thread except bug#17 prints a prompt by calling fprintf when finishing. The experiment results are shown in Table 7.

From Table 7, we see that Convoider correctly avoids all races, while Grace gets wrong output for bug#16 although it also avoids all races. However, according to our expectation, Grace should cause wrong outputs to bug#13 and bug#14 and fail to avoid bug#15. We carefully check the buggy programs and find that the way the involved threads are created is responsible for Grace's successes in perfectly avoiding these races. We find in the data-race-test suite, after a child thread is created, the main thread will access the same memory page as the new child. This will trigger Grace's sequential commit protocol, which leads the main thread tries to commit if and only if the child thread finishes and commits. Thus, all threads complete in their creation order and outputs correctly.

We are surprised that Grace could successfully avoid bug#15. This bug, as shown in Figure 8, involves condvars

Table 7: Data races avoidance results by Convoider and Grace (✔: success; ✿: wrong output).

Bug	Avoiding result	
	Convoider	Grace
Bug#10	✔	✔
Bug#11	✔	✔
Bug#12	✔	✔
Bug#13	✔	✔
Bug#14	✔	✔
Bug#15	✔	✔
Bug#16	✔	✿
Bug#17	✔	✔

t_1: Waker	t_2: Waiter
L_{01}: fprintf (stdout, ...);	L_{12}: fprintf(stdout, ...);
L_{02}: *GLOB* = 1;	L_{13}: pthread_mutex_lock(&*mtx*);
L_{03}: pthread_mutex_lock(&*mtx*);	L_{14}: while(*COND* != 1){
L_{04}: *COND* = 1;	L_{15}: pthread_cond_wait(&*cv*, &*mtx*);
L_{05}: pthread_cond_signal(&*cv*, &*mtx*);	L_{16}: }
L_{06}: pthread_mutex_unlock(&*mtx*);	L_{17}: pthread_mutex_unlock(&*mtx*);
L_{07}: usleep(1000);	L_{18}: *GLOB* = 2;
L_{08}: pthread_mutex_lock(&*mtx*);	L_{19}: usleep(1000);
L_{09}: *GLOB* = 3;	L_{20}: fprintf(stdout, ...);
L_{10}: pthread_mutex_unlock(&*mtx*);	
L_{11}: fprintf(stdout, ...);	

t_{main}

L_{21}: t_1 = pthread_create(...,Waker, ...);
.....
L_{22}: t_2 = pthread_create(...,Waiter, ...);

Figure 8: The pseudocode of bug#15.

which Grace cannot handle. In Figure 8, the main thread creates two child threads t_1 and t_2: t_1 writes global variables *GLOB* and *COND* and sends a signal to a condvar *cv*, while t_2 waits on *cv* until *COND* becomes 1 and then sets *GLOB* as 2. Because of the sequential commit protocol, Grace makes t_2 execute after t_1 finishes. When t_2 runs to line L14, it finds *COND* is 1. Thus, it continues to execute remainder instructions without waiting on *cv*. Therefore, data races on *GLOB* are avoided and correct prompts are output. However, if we change code to let t_1 be created after t_2, then Grace will make the program hang on *cv* forever while Convoider still can work to avoid data races because it permits concurrent execution of these two threads.

Bugs such as bug#10–bug#12 are simple races: for each race, there are two threads in it that each thread performs a read or a write on a shared variable. For these races, Convoider and Grace avoid them in different ways. Grace avoids them by executing threads sequentially as stated above. Thus, races are impossible. However, Convoider executes threads concurrently. Because each thread only performs one access onto the page where the shared variable locates, Convoider will only record the page into the read sets and find there are no conflicts between threads (or transactions) when committing (Section 2.2.1). So Convoider commits these transactions one by one, thus avoiding races.

Bugs such as bug#16 and bug#17 are not only races but also atomicity violations. Once they happen, the corresponding buggy programs will end with a segmentation fault

or an assertion failure. Both Convoider and Grace successfully avoid such crashes. However, Grace gets wrong output for bug#16 because it does not support revocable I/O.

We conclude that, at least for race bugs listed in Table 7, Convoider can perfectly avoid races. And owing to its revocable I/O support, Convoider does not cause wrong outputs when avoiding races for racy programs.

6.4.3. Effectiveness on Avoiding Atomicity Violations. We apply Convoider to atomicity violations bug#18–bug#28 and compare Convoider with Grace in terms of their avoidance capabilities. If no avoidance measures are carried out, bug#18 and bug#27 will cause segmentation faults to the buggy programs containing them, bug#20–bug#22 and bug#24 will cause assertion failures, bug#19, bug#25, and bug#28 will cause unexpected outputs, bug#23 will cause a buffer overflow error, and lastly bug#26 will cause a double free error. After leveraging Convoider and Grace to avoid these bugs, we list the experiment results in Table 8.

As shown in Table 8, Convoider perfectly avoids all atomicity violations, while Grace only perfectly avoids four bugs. Grace causes wrong outputs for bug#19–bug#21, bug#23, bug#27, and bug#28 and fails to avoid bug#24. For the former case, we take bug#27, as shown in Figure 9, as an example to illustrate how Grace causes wrong output for applications because of its no support for revocable I/O. Threads t_1 and t_2 in Figure 9, respectively, print prompts at their entry/exit points. The bug#27 occurs when thread t_1 is performing cache resizing while another thread t_2 is storing SQL queries into the same cache. The cache resizing (in function resize) is not atomic to both *query_cache* and *query_cache_size*, making it possible that the intermediate status of *query_cache_size* is read by other threads. For example, after t_1 calling free_cache to free *query_cache*, it executes L04 to temporarily set *query_cache_size* to *arg* before calling init_cache to set it as meaningful values. However, during this gap (L05), thread t_2 may interleave in and finds *query_cache_size* is not 0, so it calls write_block_data to write the memory pointed by *query_cache*, triggering a segmentation fault.

When running under Convoider and Grace, threads t_1 and t_2 are treated as two concurrent transactions. Suppose t_2 runs immediately after t_1. At beginning, both transactions observe non-NULL *query_cache* and nonzero *query_cache_size* (L32-L35). Then t_1 and t_2 concurrently read/write these two global variables. When finishing, suppose t_1 reaches its commit point before t_2. Because there is no already-committed transaction conflicting with t_1, it successfully commits and terminates. However, when t_2 tries to commit, it finds it conflicts with t_1 because of reading shared variables updated by t_1. So it rolls back and re-executes. At this time, Convoider will remove all messages buffered in t_2's output buffer while Grace can do nothing to revoke the already-printed messages. At last, t_2 will output its prompts again, leading to wrong outputs.

Another noteworthy point is that Grace cannot avoid bug#24 (shown in Figure 10) while Convoider can. This bug lies in Apache MPM worker subsystem. In this subsystem, there are a listener thread, which accepts socket connection and dispatches connections, and some worker threads, which get connections from the listener thread and do the actual job. The listener thread and the worker threads communicate through a queue. When a connection is accepted by the listener thread, an element will be pushed into the queue and the worker thread will pop the element from the queue and do the job. The queue keeps track of the number of idle worker threads in the system. If the number of idlers reaches to 0, the listener will stop pushing elements into the queue and wait on a condvar *queue_info->wait_for_idler* (L09-L15). Whenever a worker thread finishes its job, it will raise a signal if it finds the number of current idlers is 0 (L25-L30). This bug happens when the following steps are taken:

(1) Initially, the listener thread just accepted a connection and set idler as 0.

(2) A worker thread finishes its job, calls ap_queue_info_set _idle to increment idlers from 0 to 1.

(3) The listener thread sees that the idlers is 1, so decreases it to 0, gets another connection, and then waits for idle worker threads in function qp_queue_info_wait_for_idler.

(4) The worker thread resumes its execution and issues a conditional signal.

(5) The listener thread is waked up by the signal just issued by the worker thread sets idler as -1, causing an assertion failure.

Grace fails to avoid bug#24 because it does not properly handle condvar operations in the transaction environment. Due to not deeming conditional signals/waits as demarcation points of transactions, Grace would make a transaction keep updates to shared variables private until committing and prevent other transactions seeing up-to-date values of shared variables. Looking at bug#24's code in Figure 10, we suppose that initially *queue_info->idlers* is 2 before t_1 and t_2 run. When t_2 runs, it consecutively accepts two connections, pushes two elements into the worker queue, and decreases *queue_info->idlers* to 0. When another connection comes, t_2 will wait on *queue_info->wait_for_idlers* for idle workers because it finds *queue_info->idlers* is 0. Suppose at this time, t_1 runs. It will see *queue_info->idlers* is 2 because

Table 8: Convoider's avoidance ability against atomicity violations compared with Grace (✔: success; ✘: failure; ✿: wrong output).

Bug	Avoiding result	
	Convoider	Grace
Bug#18	✔	✔
Bug#19	✔	✿
Bug#20	✔	✿
Bug#21	✔	✿
Bug#22	✔	✔
Bug#23	✔	✿
Bug#24	✔	✘
Bug#25	✔	✔
Bug#26	✔	✔
Bug#27	✔	✿
Bug#28	✔	✿

t_1: resize

```
L01: ulong resize(ulong arg){
L02: fprintf(stdout, "resizing begin…\n");
L03: free_cache(0);
L04: query_cache_size = arg;
L05: usleep(100);
L06: query_cache_size = init_cache(...);
L07: fprintf(stdout, "resizing end.\n"); }

L08: void free_cache(...){
L09: if (query_cache_size > 0){
L10:   pthread_mutex_lock(&guard_mutex);
         ......
L11:   if (query_cache){
L12:     free(query_cache);
L13:     query_cache = NULL;
L14:     query_cache_size = 0;}
         ......
L15: pthread_mutex_unlock(&guard_mutex);
L16:   }
L17: }
```

t_2: store_query

```
L18: void store_query(...){
L19: fprintf(stdout, "storing begin…\n");
L20: pthread_mutex_lock(&guard_mutex);
L21: if (query_cache_size == 0){
L22:   pthread_mutex_unlock(&guard_mutex);}
       ......
L23: write_block_data(...);
L24: pthread_mutex_unlock(&guard_mutex);
L25: fprintf(stdout, "storing end.\n");
L26: }

L27: void write_block_data(constchar*query){
L28: int size = min(strlen(query), query_cache_size);
L29: memcpy(query_cache, query, size);
L30: }
```

t_{main}

```
L31: int main(){
L32: query_cache = malloc(128);
L33: query_cache_size = 128;
L34: t1 = pthread_create(…, resize, (void*)100);
L35: t2 = pthread_create(…, store_query, ...);
       ......
L36: }
```

Figure 9: The pseudocode of bug#27.

t_1

```
L01: void* worker_thread(...){
       ......
L02: while (!workers_may_exit){
         ......
L03:   ap_queue_info_set_idle(...);
         ......
L04:   ap_queue_pop(worker_queue, ...);
         ......
L05:   process_socket(...);
         ......
L06: }
L07: }

L08: apr_status_t ap_queue_info_set_idle(...){
       ......
L09: prev_idlers = queue_info->idlers;
L10: queue_info->idlers = prev_idlers+1;
L11: usleep(100);
L12: if (prev_idlers == 0){
L13:   pthread_mutex_lock(queue_info->idlers_mutex);
L14:   pthread_cond_signal(queue_info->wait_for_idler);
L15:   pthread_mutex_unlock(queue_info->idlers_mutex);
L16: }
L17: }
```

t_2

```
L18: void* listener_thread(...){
       ......
L19: while (1){
         ......
L20:   ap_queue_info_wait_for_idler(...);
         ......
L21:   ap_queue_push(worker_queue, ...);
         ......
L22: }
L23: }

L24: apr_status_t ap_queue_info_wait_for_idler(...){
       ......
L25: if (queue_info->idlers == 0){
L26:   pthread_mutex_lock(queue_info->idlers_mutex);
L27:   if (queue_info->idlers == 0){
L28:     pthread_cond_wait(queue_info->wait_for_idler,
           queue_info->idlers_mutex);
L29:   }
L30:   pthread_mutex_unlock(queue_info->idlers_mutex);
L31: }
L32: apr_atomic_dec(&(queue_info->idlers));
L33: assert (queue_info->idlers >= 0);
L34: }
```

Figure 10: The pseudocode of bug#24.

t_2's update to the variable is now still invisible to t_1. Therefore, t_1 increments *queue_info*->idlers by 1 and does not issue a signal onto *queue_info->wait_for_idler*, leading t_2 to wait forever.

In contrast to Grace, Convoider can perfectly avoid bug#24 due to its appropriate condvar handling described in Section 4. Convoider views condvar signal/wait operations as demarcations of transactions. So, in Figure 9, when t_2 decreases *queue_info->idlers* to 0 and waits on *queue_info->wait_for_idler*, it has committed its updates to shared variables to the global memory. Therefore, when t_1 runs, it will find *queue_info->idlers* is 0 and then issue a signal onto *queue_info->wait_for_idler*, making t_2 proceed.

We conclude that, at least to atomicity violation bugs listed in Table 8, Convoider can perfectly atomicity violations. And owing to its revocable I/O support and proper condvar handling, Convoider can correctly avoid bugs involving condvar and I/O operations.

6.4.4. Effectiveness on Avoiding Order Violations. We apply Convoider on order violations bug#29–bug#31 and compare Convoider with Grace in terms of their avoidance capabilities. If no avoidance measures are taken, bug#29 and bug#30 will trigger segmentation faults while bug#31 will lead to unexpected outputs. After applying Convoider and Grace to them, we list the experiment results in Table 9.

Table 9: Order violation results by Convoider and Grace (✔: success; ✘: failure; ✿: wrong output).

Bug	Avoiding result	
	Convoider	Grace
Bug#29	✘	✘
Bug#30	✔	✔
Bug#31	✔	✔

As shown in Table 9, both Convoider and Grace successfully avoid bug#30 and bug#31 but fail to avoid bug#29. According to Lu et al. [4], an order violation occurs when two operations (or two groups of operations) from two different threads are executed in an undesirable order. Convoider does not impose order control on transactions, so it only can avoid order violations with probability 0.5. Grace either does not guarantee transactions to finish in some specific order in most cases. The only exception is that if a parent thread accesses any global or heap memory pages after creating a child thread, Grace's sequential commit protocol will be triggered and the parent thread will pause until the child thread successfully commits. In that case, the child thread definitely finishes before the parent thread.

Taking bug#29 and bug#30 as examples to illustrate how Convoider and Grace fail and succeed in avoiding them, respectively. For bug#29 shown in Figure 11, the global variable *global_opt* may be referenced by thread t_2 (L07-L09) before it is initialized with valid values in t_1 (L04), leading to a segmentation fault. This bug cannot be avoided by Convoider and Grace because t_2 almost always runs concurrently with t_1 and reads uninitialized value NULL of *global_opt*. For bug#30 shown in Figure 12, the global variable *h->bandwidth* also may be referenced by t_2 (L08) before it is initialized with meaningful values in t_1 (L04). However, both Convoider and Grace are able to avoid this bug because t_1 always finishes before t_2. If t_1 and t_2 run concurrently, bug#30 will be triggered and cannot be avoided by Convoider and Grace.

We conclude that, according to the experiment results, Convoider cannot avoid order violations in one hundred percentage.

7. Conclusion

This paper presents Convoider, a runtime STM system with proper condvar handling and revocable I/O support, for transparently transactionalizing multithreaded C/C++ applications and dynamically avoiding concurrency bugs such as deadlocks, data races, and atomicity violations.

We perform three experiments to evaluate and compare Convoider with Grace, Dthreads, Dimmunix, and Slider in terms of the applicability, efficiency, and effectiveness. Evaluation results show that Convoider succeeds in transparently transactionalizing twelve real-world applications with averagely incurring only 28% runtime overhead and perfectly avoids 94% of thirty-one concurrency bugs used in our experiments. Comparison

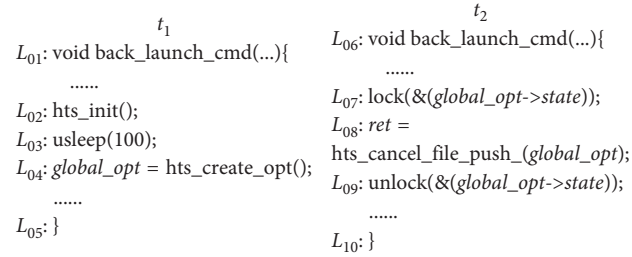

Figure 11: The pseudocode of bug#29.

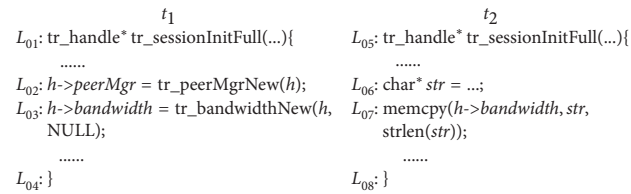

Figure 12: The pseudocode of bug#30.

results show that Convoider could correctly transactionalize many more applications and avoid many more concurrency bugs than other tools. This study can help efficiently transactionalize legacy multithreaded applications and effectively improve the runtime reliability of them.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This paper was supported by Key Supported Disciplines of Guizhou Province–Computer Application Technology (No. QianXueWeiHeZiZDXX[2016]20), Specialized Fund for Science and Technology Platform and Talent Team Project of Guizhou Province (No. QianKeHePingTaiRenCai[2016]5609), Major Research Projects of Innovation Group of Guizhou Provincial Department of Education (No. QianJiaoHeKY[2016]040), Engineering Research Center of the Higher Education Institutions of Guizhou Province (No. QianJiaoHeKY[2016]015), 2018 Doctoral Foundation of Guizhou Education University (No. 2018BS004), and Service Platform Construction for Science and Technology Resources in Guizhou Province–Internet Plus Based Science and Technology Resource Platform Construction (No. QiankeZhongYinDi[2016]4009).

References

[1] H. Sutter and J. Larus, "Software and the concurrency revolution," *ACM Queue*, vol. 3, no. 7, pp. 54–62, 2005.

[2] J. Wang, W. S. Dou, Y. Gao et al., "A comprehensive study on real world concurrency bugs in Node.js," in *Proceedings of 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 520–531, Urbana, IL, USA, November 2017.

[3] J. Wang, Y. Y. Jiang, C. Xu et al., "AATT+: effectively manifesting concurrency bugs in Android apps," *Science of Computer Programming*, vol. 163, pp. 1–18, 2018.

[4] S. Lu, S. Park, E. Seo et al., "Learning from mistakes: a comprehensive study on real world concurrency characteristics," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 1, pp. 329–339, 2008.

[5] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multi-threaded programs," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 446–455, 2007.

[6] S. Park, S. Lu, and Y. Y. Zhou, "CTrigger: exposing atomicity violation bugs from their hiding places," *ACM SIGPLAN Notices*, vol. 44, no. 3, pp. 25–36, 2009.

[7] K. Sen, "Race directed random testing of concurrent programs," *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 11–21, 2008.

[8] W. Zhang, J. Lim, R. Olichandran et al., "ConSeq: detecting concurrency bugs through sequential errors," *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 251–264, 2011.

[9] Z. Yin, D. Yuan, and Y. Y. Zhou, "How do fixes become bugs?ąła comprehensive characteristic study on incorrect fixes in commercial and open source operating systems," in *Proceedings of 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 26–36, Szeged, Hungary, September 2011.

[10] M. Zhang, Y. Wu, S. Lu et al., "AI: a lightweight system for tolerating concurrency bugs," in *Proceedings of 22nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 330–340, Hong Kong, China, November 2014.

[11] E. D. Berger, T. Yang, T. Liu et al., "Grace: safe multi-threaded programming for C/C++," *ACM SIGPLAN Notices*, vol. 44, no. 10, pp. 81–96, 2009.

[12] J. T. Wamhoff, C. Fetzer, P. Felber et al., "FastLane: improving performance of software transactional memory for low thread counts," in *Proceedings of 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 113–122, Shenzhen, China, February 2013.

[13] M. F. Spear, L. Dalessandro, V. J. Marathe et al., "A comprehensive strategy for contention management in software transactional memory," *ACM SIGPLAN Notices*, vol. 44, no. 4, pp. 141–150, 2009.

[14] P. Felber, C. Fetzer, P. Marlier et al., "Time-based software transactional memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 12, pp. 1793–1807, 2010.

[15] M. Feng, R. Gupta, and I. Neamtiu, "Programming support for speculative execution with software transactional memory," in *Proceedings of 27th International Symposium on Parallel and Distributed Processing*, pp. 394–403, Boston, MA, USA, October 2013.

[16] A. Matveev and N. Shavit, "Towards a fully pessimistic STM model," in *Proceedings of 7th ACM SIGPLAN Workshop on Transactional Computing*, pp. 1–9, New Orleans, LA, USA, February 2012.

[17] D. Dice, O. Shalev, and N. Shavit, "Transactional locking II," in *Proceedings of 20th International Symposium on Distributed Computing*, pp. 194–208, Stockholm, Sweden, September 2006.

[18] A. Welc, B. Saha, and A. Adl-Tabatabai, "Irrevocable transactions and their applications," in *Proceedings of 20th Symposium on Parallelism in Algorithms and Architectures*, pp. 285–296, Munich, Germany, June 2008.

[19] M. F. Spear, M. Silverman, L. Dalessandro et al., "Implementing and exploiting inevitability in software transactional memory," in *Proceedings of 37th International Conference on Parallel Processing*, pp. 59–66, Portland, OR, USA, September 2008.

[20] H. Volos, A. J. Tack, N. Goyal et al., "xCalls: safe I/O in memory transactions," in *Proceedings of 4th ACM Euro Conference on Computer Systems*, pp. 247–260, Nuremberg, Germany, April 2009.

[21] A. Dragojevic, P. Felber, V. Gramoli et al., "Why STM can be more than a research toy," *ACM Communications*, vol. 54, no. 4, pp. 70–77, 2011.

[22] H. Volos, A. J. Tack, M. M. Swift et al., "Applying transactional memory to concurrency bugs," in *Proceedings of 17th International Conference on Architectural Support for Programming Language and Operating Systems*, pp. 211–222, London, UK, March 2012.

[23] V. Pankratius and A. R. Adl-Tabatabai, "Software engineering with transactional memory versus locks in practice," *Theory of Computing Systems*, vol. 55, no. 3, pp. 555–590, 2014.

[24] C. J. Rossbach, O. S. Hofmann, and E. Witchel, "Is transactional programming actually easier?," in *Proceedings of 15th ACM SIGPLAN Symposium on Principle and Practice of Parallel Programming*, pp. 47–56, Bangalore, India, January 2010.

[25] F. Zyulkyarov, V. Gajinov, O. S. Unsal et al., "Atomic quake: using transactional memory in an interactive multiplayer game server," in *Proceedings of 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 25–34, Raleigh, NC, USA, February 2009.

[26] T. Harris, S. Marlow, S. Peyton-Jones et al., "Composable memory transactions," in *Proceedings of 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 48–60, Chicago, IL, USA, June 2005.

[27] L. Baugh and C. Zilles, "An analysis of I/O and syscalls in critical sections and their implications for transactional memory," in *Proceedings of 2008 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 54–62, Austin, TX, USA, April 2008.

[28] T. Harris and K. Fraser, "Language support for lightweight transactions," *ACM SIGPLAN Notices*, vol. 38, no. 11, pp. 388–402, 2003.

[29] D. Nicacio, A. Baldassin, and G. Araujo, "Transaction scheduling using dynamic conflict avoidance," *International Journal of Parallel Programming*, vol. 41, no. 1, pp. 89–110, 2013.

[30] A. Skyrme and N. Rodriguez, "From locks to transactional memory: lessons learned from porting a real-world application," in *Proceedings of 8th ACM SIGPLAN Workshop on Transactional Computing*, pp. 1–9, Houston, TX, USA, March 2013.

[31] W. Ruan, T. Vyas, Y. Liu et al., "Transactionalizing legacy code: an experience report using GCC and Memcached," in *Proceedings of 19th International Conference on Architecture Support for Programming Languages and Operating Systems*, pp. 399–412, Salt Lake City, UT, USA, March 2014.

[32] C. Bienia, S. Kumar, J. P. Singh et al., "The PARSEC benchmark suite: characterization and architectural implications," in *Proceedings of 17th International Conference on Parallel Architectures and Compilation Techniques*, pp. 72–81, Toronto, Canada, October 2008.

[33] The Modified SPLASH2 Home Page, 2018, http://www.capsl.udel.edu/splash/.

[34] C. Ranger, R. Raghuraman, A. Penmetsa et al., "Evaluating MapReduce for multi-core and multiprocessor systems," in *Proceedings of 13th International Symposium on High Performance Computer Architecture*, pp. 13–24, Phoenix, AZ, USA, February 2007.

[35] Z. Yu, X. H. Su, and P. J. Ma, "Mocklinter: linting mutual exclusive deadlocks with lock allocation graphs," *International Journal of Hybrid Information Technology*, vol. 9, no. 3, pp. 355–374, 2016.

[36] Google's Data Race Test, 2018, http://code.google.com/p/data-race-test/.

[37] J. Yu, S. Narayanasamy, C. Pereira et al., "Maple: a coverage-driven testing tool for multi-threaded programs," *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 485–502, 2012.

[38] J. Yu and S. Narayanasamy, "A case for an interleaving constrained shared-memory multi-processor," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 325–336, 2009.

[39] T. Liu, C. Curtsinger, and E. D. Berger, "Dthreads: efficient deterministic multithreading," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pp. 327–336, Cascais, Portugal, October 2011.

[40] H. Jula, D. Tralamazza, C. Zamfir et al., "Deadlock immunity: enabling systems to defend against deadlocks," in *Proceedings of 8th USENIX Symposium on Operating Systems Design and Implementation*, pp. 295–308, San Diego, CA, USA, December 2008.

[41] Z. Yu, X. H. Su, and P. J. Ma, "Slider: an online and active deadlock avoider by serial execution of critical sections," *International J. High Perf. Syst. Archi.*, vol. 6, no. 1, pp. 36–50, 2016.

[42] T. Merrifield and J. Eriksson, "Conversion: multi-version concurrency control for main memory segments," in *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 127–139, Prague, Czech Republic, April 2013.

[43] E. D. Berger, K. S. Mckinley, R. D. Blumofe et al., "Hoard: a scalable memory allocator for multi-threaded applications," in *Proceedings of 9th International Conference on Architectural Support for Programming Language and Operating Systems*, pp. 117–128, Cambridge, MA, USA, November 2000.

[44] E. D. Berger, B. G. Zorn, and K. S. Mckinley, "Composing high-performance memory allocators," in *Proceedings of 22nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 114–124, Snowbird, UT, USA, June 2001.

[45] C. Blundell, E. C. Lewis, and M. Martin, *Unrestricted Transactions Memory: Supporting I/O and System Calls within Transactions*, University of Pennsylvania, Philadelphia, PA, USA, 2006.

[46] C. Wang, Y. Liu, and M. Spear, "Transaction-friendly condition variables," in *Proceedings of 26th ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 198–207, Prague, Czech Republic, June 2014.

[47] C. C. Minh, J. W. Chung, C. Kozyrakis et al., "STAMP: stanford transactional applications for multi-processing," in *Proceedings of 2008 IEEE International Symposium on Workload Characterization*, pp. 25–46, Seattle, WA, USA, September 2008.

[48] H. K. Pyla and S. Varadarajan, "Avoiding deadlock avoidance," in *Proceedings of 19th International Conference on Parallel Architecture Compilation Techniques*, pp. 75–86, Vienna, Austria, Septembrer 2010.

[49] T. Shimomura and K. Ikeda, "Two types of deadlock detection: cyclic and acyclic," *Intelligent System for Science and Information*, vol. 54, no. 2, pp. 233–259, 2016.