Dresden University of Technology

Department of Computer Science
Institute for System Architecture
Systems Engineering Group

Diploma Thesis

Transactional Execution of System-Library Functions

Thomas Zimmermann

October 28, 2009

Supervisor: Prof. Christof Fetzer
Advisor: Torvald Riegel

Word-based software transactional memory only supports memory operations. To serve as a full replacement for critical sections, arbitrary functions have to be supported as well. This work presents a generic and extensible way for the transactional support of operations that work on non-memory resources, so-called external actions. Each transaction is provided with transaction-safe abstractions of non-transactional resources and functions. The transactional memory system is extended by a transaction-local history, which holds any external actions that have been called within the transaction. The history is used for validating the consistency of resources used by the transaction, and for applying or undoing actions called from within the transaction. The work introduces transaction-safe versions of common resources and operations of the POSIX standard, such as memory allocation or file-descriptor I/O. It discusses trade-offs and problems when using POSIX functionality in transactional code. At the end a detailed evaluation is presented, as well as some ideas for future work on the topic.

Technische Universität Dresden / Fakultät Informatik

Diploma Thesis

Name, Vorname:	Thomas Zimmermann
Studiengang:	Diplominformatik
Matrikelnummer:	2939847
Thema:	***Transactional Execution of System Library Functions***

Software Transactional Memory (STM) provides transactional guarantees for accesses to main memory. This eases the implementation of concurrent data structures, for example. However, programmers thus cannot embed code into memory transactions that accesses other resources besides the application's main memory. Removing this limitation and allowing external actions would significantly expand the applicability of memory transactions (e.g., for failure atomicity).
Thus, STM implementations should support external actions. If there is already proper transaction support for such actions (e.g., file accesses in a transactional file system) they are straight-forward to combine with memory transactions. However, the majority of the actions provided by existing libraries is not transactional, so combining them with memory transactions requires additional support code that isolates them from other actions and makes them atomic if possible (by providing roll back mechanisms).
The aim of this diploma thesis is (A) to investigate how to provide transaction support for the external actions contained in a typical system library and (B) to implement a prototype of such support for the I/O functions in the GNU C Library (`glibc`).
The investigation should (1) cover which ways exist to provided transactional guarantees in this setting in general, (2) classify typical C system library functions regarding whether transactional guarantees can be reasonably provided, and how, and (3) discuss how error signaling and handling mechanisms of the functions affect transactional execution and how these issues can be solved. Furthermore, it should be studied how falling back to nontransactional execution (e.g., via irrevocability) can be avoided.
The prototype should be implemented as wrappers for selected `glibc` functions, focusing on low-level I/O, and optionally stream I/O and filesystem operations. Not all functions need to be supported, but the selection should be representative in the sense that it roughly covers all issues that may arise when supporting all the functions. The implementation should strive to provide a framework or helper functions to ease the process of implementing transaction support for other functions.
The evaluation of the thesis' results should focus on achieved features (e.g., how much the applicability of memory transactions has been expanded) and performance (e.g., single-thread overheads, scalability, and performance with multiple threads). Additionally, it should be evaluated whether transactional execution of system library functions conflicts with standards such as POSIX.

Betreuer:	Dipl.-Inf. Torvald Riegel
Verantwortlicher Hochschullehrer:	Prof. Christof Fetzer
Institut:	Systemarchitektur
Lehrstuhl:	Systems Engineering
Beginn am:	1.1.2009
Einzureichen am:	1.7.2009

Student　　　　　　　　　　　　Verantwortlicher Hochschullehrer

Declaration

Herewith I declare that this submission is my own work and that, to the best of my knowledge, it contains no material previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher education, except where due acknowledgment has been made in the text.

Dresden, October 28, 2009

Thomas Zimmermann

Contents

List of Figures

List of Tables

Listings

Acknowledgments

Several people helped me in the process of writing this diploma thesis. I thank Torvald Riegel for advising me during the creation of the work and for reviewing the text. The concepts of domains and actions are based on his work.

I also thank the other people of the transactional-memory group: Jons-Tobias Wamhoff, Martin Nowack, and Diogo Becker de Brum, who contributed their help. Diogo also reviewed several chapters of the text and pointed out an important detail I missed. Another reviewer was Martin Süßkraut. My thanks to him for reviewing the complete diploma thesis and providing me with valuable feedback.

1 Introduction

Word-based transactional memory systems only support memory operations. To be equally versatile as critical sections, arbitrary functions need to be handled as well. This work describes a generic and extensible framework for using C functions and non-transactional resources from within memory transactions. In contrast to previous work on this topic, it provides a general and flexible solution, and investigates the support for transactional versions of arbitrary components.

This first chapter introduces the topic and motivation in an informal way and gives some information about the problems that have to be solved.

1.1 Retrospection

During the first 30 years of modern computer technology[1] the execution speed of processor-bound computer programs roughly doubled every 18 months. With each new processor generation most old software ran faster than before; without having its programmer to optimize for the new processors.

This changed [Sut05] around 2004. Processor manufacturers were unable to significantly improve the performance of their products anymore, neither by increasing the clock speed, nor by architectural changes.

The solution was to double the number of cores per processor, such that there are now really two or more processors driving the computer. These multi-core architectures existed long before, but not in the mass market. Multi-core systems were used and programmed by experts on this field: people who often had very specific, computationally intensive problems to solve.

The introduction of multi-core systems to the mass market has kept the processors' theoretical performance doubling with each new processor generation. The drawback is that software performance does not double automatically anymore. Instead, programmers now have to actively modify their software to take advantage of the additional processor cores: they have to parallelize the software.

1.2 Critical Sections

The major problem when parallelizing software is the use of resources that are shared among several cores, such as global variables. The access to such resources is not automatically handled among processor cores. Instead, every shared resource needs to be protected by some form of concurrency control.

Concurrency control is the activity of coordinating the actions of processes that operate in parallel, access shared data, and therefore potentially interfere with each other [BHG87].

Two basic strategies for controlling concurrency exist. *Pessimistic concurrency control* ensures that once consistency has been validated, it holds true from the point of validation. This can be achieved by the use of locks, which block other transactions from accessing the respective resource.

[1]Let's assume *modern* computer technology started in 1971 with the invention of the first microprocessor, which made widespread use of computers possible in the first place.

Optimistic concurrency control ensures that consistency holds true up until the point of validation. This can be achieved by putting the resource under version control and comparing a transaction's local version with the global version of the resource.

The classic form of concurrency control is protection by the use of explicit locking. The piece of code where this takes place is called a *critical section.* In a system with explicit locking a critical section is implemented by

1. acquiring the locks of all concerned resources,
2. accessing these resources, and
3. releasing the locks.

Locking blocks other threads from accessing the lock's resource. This is called *mutual exclusion.* If a lock protects several independent resources at the same time, unrelated threads might be blocked unnecessarily, which negatively affects performance. Therefore, the parallel program should ideally contain one distinct lock for each independent resource. Such *fine-grained locking* minimizes interference among threads and allows for good scalability.

On the other hand, applications with fine-grained locking easily have problems with deadlocks. For example, thread T_1 wants to move items from a list x to another list y. For that purpose, it has to acquire the lock for x and the lock for y. Another, unrelated thread T_2 wants to do the same operation, but acquires the necessary locks in reversed order. When the locking operations of both threads interleave, the result is a *deadlock:* a state where two or more threads wait for each other to release their locks.

A possible solution is to reduce the number of locks to be acquired by a thread, possibly up to a point where only one lock protects all shared resources in the program. Such *coarse-grained locking* addresses the problem of deadlocks, but does not scale well if the number of concurrent threads increases. In the example above, there would be one lock for all lists in the application. Only acquiring this single lock prevents the lurking deadlock between T_1 and T_2, but also blocks an unrelated thread T_3 working on a distinct list z.

A real-world example is the *Big Kernel Lock,* a global lock which protects data structures within the Linux kernel from concurrent access. In [BH03] Bryant and Hawkes measured the impact of this lock on the file-system performance for parallel I/O. Besides other observations they found that in some cases up to 70 percent of the time was spend on waiting for the kernel lock to be released.

So with coarse-grained locking, it is hard to create software which is highly concurrent, whereas with fine-grained locking software is prone to deadlocks. Also, the explicit use of locks makes it hard to build composable components. Ideally components should be self-contained. If distinct components operate on a shared resource, they have to share the locks for the resource. This makes implementation details visible to the components' outside and breaks information hiding among them.

1.3 Transactional Memory

Transactional memory [ST95] promises a way to solve these problems. Informally, a transaction is a section of code that moves the system from one consistent state to another consistent state. In the context of this work, a *consistent state* is one that could be observed after each transaction during a serial execution of a set of concurrent transactions.

Transactions are often defined by a set of common properties: atomicity, consistency, and isolation [HR83].[2] *Atomicity* guarantees that either all or no updates of a transaction are taking effect. *Consistency* guarantees that the system is in a well-defined state before and after a transaction executes, independently of whether the transaction commits or aborts. *Isolation* guarantees that components outside of a transaction cannot see its intermediate states, but only its final state after it has been committed to the system.

Transactional code

1. signals the beginning of the transaction,
2. executes the transaction,
3. signals its wish to commit the updates.

The difference to critical sections is the implicit protection of resources. All memory operations go through the *transactional memory system:* a single software or hardware component which detects conflicts among concurrently running transactions. A *conflict* is a state where two or more transactions concurrently access the same data and at least one of them executes a write operation [BHG87]. If a conflict is detected, all but one conflicting transactions are aborted and have to restart.

The obvious advantage of this concept is that deadlocks can easily be prevented as all locking is centralized in one component. Also, by refining the implementation of the transactional memory system, the application can be scaled or adjusted to new workloads more easily than with critical sections.

The disadvantage is the inability to handle anything else than main memory.[3] With critical sections, access to a shared resource is protected by the use of locks. The nature of the resource is thereby unimportant. Transactional memory systems instead are hooked into the application's load and store operations on the main memory. This makes it impossible to handle other resources without additional work.

1.4 External Actions

An *external action* is a function that accesses or modifies the state of a resource which is not by default under the control of the transactional memory system. Therefore, the transactional memory system cannot detect conflicts on this resource; and in the case of an abort of the transaction, it cannot revoke updates to the resource's state.

Assume a transaction T_1 reads a value x from main memory and writes it to a file. Between the transaction's write operation and its commit, another transaction T_2 commits an update to x. The transactional memory system now aborts T_1 as it conflicts with the just committed transaction $T_{2.}$ However, T_1's update to the file is not revoked because the transactional memory system is not aware of it.

Several solutions to this problem are presented by Baugh and Zilles [BZ07]. The simplest solution is to forbid any external actions. Obviously, this is inconvenient to the application developer as it severely limits the applicability of transactional memory.

[2]The forth common property of transactions, durability, depends on the application and its environment. While it is ensured by database systems, durability is at most partially provided by transactional memory systems.

[3]More precisely said: anything else than physical main memory. If a memory page has been mapped from device memory, the application cannot access it with transactional semantics because writes to hardware devices are typically not meant to be revocable.

Another solution is to never abort a transaction that has executed an external action. Such a non-abortable transaction is called *irrevocable* or *inevitable.* Before becoming irrevocable a transaction has to validate that (1) it does not conflict with other transactions, and (2) there is not yet another transaction running irrevocable. If both premises are met the transaction becomes irrevocable; if not, the transaction is aborted. In the example above, T_2 would not have been able to commit because it conflicted with T_1, which would be running irrevocably after its write operation. The advantage of irrevocability is that it is easy to implement and works in all cases because access' to all resources is protected with respect to the boundaries of the irrevocable transaction. The disadvantage is that irrevocability works like a big lock for all resources in the system: It does not scale well and thus, when external actions are used extensively, the performance of a multi-core system might be similar to that of a single-core system.

A more scalable solution is the use of fine-grained concurrency control on the external resources and its integration with the transactional memory system. Additionally, for each external action there is a *compensation action,* which is called by the transactional memory system during a transaction's abort, It resets the resource to the state it had before the external action took effect. In the example above, when T_1 is aborted, the write operation is undone during the abort. The advantage of fine-grained concurrency control and compensation actions is the possibility to use external actions in multiple concurrently running transactions, which enables better scaling of the system. For example, if there are several transactions using a resource without conflicting, all of them can run concurrently. The disadvantage is that this is more complicated to implement than irrevocability. Also, it is not always possible to provide a compensation action. For example, if a new process is forked from within a transaction, it is not possible to decide whether it is safe to kill this process during an abort or not.

As a fourth possible solution, some actions can also be deferred into the transaction's commit phase, especially if no other actions depend on their results or the results can be simulated. In the given example, the write operation by transaction T_1 would not be done until T_1 actually commits.

1.5 Requirements

So far, this chapter only introduced the basic problems with external actions and their possible solutions. This section discusses the requirements that external-action support should fulfill. Looking at the transactional memory system on one side and a typical Unix system on the other side, the important design aspects are

- correctness,
- standards compliance,
- extensibility, and
- performance.

These points are roughly in the order of their importance, but sometimes exceptions are possible. Each is now discussed in more detail.

Correctness. Correctness defines the behavior of transactional code. The design presented in this work strives to provide atomicity, consistency and isolation for transactions with external actions. This is in line with the previous section on transactions.

Informally, it shall be possible to atomically move all resources from one consistent state to another consistent state. A consistent state is a state that could be observed after the successful commit of a transaction in a serial execution of a set of transactions.

Isolation shall be provided among concurrent transactions. After the commit of the last transaction of a set of concurrent transactions, it shall be possible to execute non-concurrent code. The presented framework does not attempt to isolate transactions from other, concurrent processes in the system. This would require support at the kernel level. In the case of this work, the criteria for isolation is serializability. A schedule of a set of transactions is defined to be *serializable* if its result is equivalent to the result of a serial schedule [BHG87]. Assume two transactions, T_1 and T_2, that access two individual resources x and y. T_1 reads the value of x and writes it to y. Between T_1's read and write operation, transaction T_2 updates the value in x. This schedule is not serializable. In the serial case, either T_1's write to y finishes before T_2's write to x occurs, or T_2's write finishes before T_1's read occurs.

The design does not strive for durability. The presented framework is a transactional implementation of POSIX functionality, so that applications can replace their critical sections with transactions. POSIX does not request durability itself, so this property is likely unnecessary for the target applications. Additionally, durability strongly depends on the setup of the underlying hardware and software; a transactional memory system cannot provide this by itself.

Standards compliance. *Standards compliance* requests that an operation that is executed within a transaction has the same semantics as its non-transactional counterpart. This also applies to series of operations.

Compliance with standards has two aspects. The transactional code should (1) formally comply to the relevant standards, and (2) allow for the use of common practices, which means to allow work to be done in the common (best) way.

The first aspect, formal compliance, specifies how an external action shall behave. To be of practical use to the application programmer, it has to provide the semantics of its non-transactional variant whenever possible. For example, a call from within a transaction to synchronize a file's content must result in the written data to be committed to disk if the non-transactional variant would have shown this behavior. In the case of this work, the relevant standard for formal compliance is POSIX, the *Portable Operating System Interface* [Ope08]. It defines the standard C interface and environment for modern variants of Unix and similar operating systems.

The second aspect, common practices, describes what an application developer should do, not merely want he or she can do. The support of common practices is equally important as formal compliance to technical standards. Especially on Unix systems, the correctness of software often depends on conventions, which the programmer is advised to follow. An example is the creation of new files with content. Write operations are not atomic, but rename operations are. Hence, the application first creates a temporary file and fills it with data. Afterwards the temporary is renamed to the final location. Since the rename operation is atomic, this allows for the instantaneous generation of files within the file system; even in the presence of concurrent writers that create files with the same final name. Many practices for programming in Unix and Linux environments are discussed in [Dre09, Whe03]. This includes dynamic memory allocation, string handling, file-system interaction, and others.

Extensibility. *Extensibility* refers to the ease of adding new features. The possibility to easily introduce new external actions is important for supporting future versions of the POSIX standard, or a new library's data structures. Extensibility allows transactions to keep up with critical sections in terms of versatility.

Assume an application uses some XML parser library. It should be possible for the programmer to integrate the library with the transactional memory system to allow for transaction-safe parsing and modification of XML trees without having to rewrite everything. Extensibility is also important for optimizing towards a specific problem. An application might have a very specific behavior during file I/O. If this is covered badly by existing strategies for concurrent I/O the programmer might want to implement a new strategy. This overlaps with the next point.

Performance. Performance is almost an imperative for system software. A software has high *performance* if its response time is minimal and its throughput is maximal. *Response time* is the time span between issuing a transaction and its successful completion; *throughput* is the number of successfully processed transactions per time unit [WV01]. In the case of transactional memory, performance can be optimized by using available processor cores as efficient as possible. Having external actions that allow for concurrent execution is therefore a plus.

1.6 Contribution

This work contributes a framework for using external actions from within transactions. The framework's design allows for the support of arbitrary resources and operations, at a fine-grained level of concurrency control. Previous work, which is discussed in Chapter 2, only supported resources with specific properties, like in the case of transactional boosting; or did only allow for a rather coarse-grained concurrency control, like xCalls. To meet the presented requirements the framework uses a hybrid approach for handling external actions. It either tries to defer actions or provides a compensation action for actions that have one. In cases where no compensation action is available or the semantics are unclear, the framework allows for the switch of a transaction to an irrevocable mode. To handle errors that happen at commit time, the presented work includes a simple mechanism to support their detection and handling.

Together with the design, an implementation is provided that is suitable for real software and features all important design aspects.

A set of components, which implement support for important parts of the POSIX system interface, is provided in addition to the basic framework. For each component a detailed description of the related resources and functions is given, together with a discussion of the possible problems of their transactional usage. Previous work only presented few information on supporting POSIX compatibility and interaction of transactional and non-transactional processes in the system. The components include memory management, file-descriptor I/O, file-system support, math, and string and memory functions.

1.7 Overview

The diploma thesis is organized as follows. Chapter 2 summarizes previous work on the topic of software transactional memory and external actions. Additionally, it points to relevant standards and software on which the work is based on. The framework's design is discussed in Chapter 3 and an implementation is presented in Chapters 4 and 5. Chapter 6 evaluates design and implementation with respect to the discussed requirements. Some ideas for future work on the topic are given in Chapter 7. The work finishes with a conclusion in Chapter 8 and a list of supported functions of the C standard library in Appendix A.

2 Related Work

This chapter provides an overview of research that is closely related to the use of external actions in transactional memory systems. It starts by introducing basic texts on the topic, and afterwards presents previous research on the support of external actions. At the end, the chapter gives some pointers to related standards and software.

Fundamentals

Transactional memory has its roots in the fields of concurrent programming and database research. An overview of concurrent programming can be found in [HS08] by Herlihy and Shavit. Basic information about database and serialization theory can be found in [BHG87] by Bernstein, Hadzilacos, and Goodman; and [WV01] by Weikum and Vossen. Transactional actions were first classified by Gray in [Gra81] and later in [GR93] by Gray and Reuter.

The paper that originated software transactional memory is [ST95] by Shavit and Touitou. Therein the authors describe a simple word-based transactional memory system. This early implementation is quite limited and does especially not contain facilities to execute external actions.

Exceptions and side-effects in atomic blocks

In [Har04], Harris presents a Java-based software transactional memory system that supports a simple mechanism for using I/O in transactions.

To support transactions, the used Java Virtual Machine implements *atomic blocks*, which execute with transactional semantics. The Java VM also provides an interface for an I/O library to query if it is running in an atomic block. In this case it can register callback functions for applying and undoing any updates to the non-transactional external state. During commit and abort either of these callbacks is executed by the transactional memory system. The paper makes the example of an I/O library that buffers all output. In the case of a commit of the atomic block, the buffered updates are flushed to the file; in the case of an abort, the updates are discarded.

External action support is implemented by the use of contexts: Java objects on which external actions execute. Context can be nested: an atomic block, which is itself a context, can contain several subcontexts. Each context implements two-phase locking.

A context can query the validity of its subcontexts; and, depending on the result, applies the contained external actions during the commit or undoes them. The apply operation is intended to merge the state of a subcontext into the state of the containing context.

The work at hand does not support nesting, but uses similar concepts for handling external action as the discussed paper. Any external actions can be deferred or compensated. The presented mechanism is generic enough to allow for the support of various different actions. What seems missing in Harris' approach is irrevocability, so non-deferable, non-compensatable actions might not be supported. Another difference is that even though an atomic block can contain several distinct contexts there is no history for the complete transaction. However, it might be possible to build this by implementing the history as a context of its own, and running each external actions as a subcontext of the history.

Unrestricted transactional memory: supporting I/O and system calls within transactions

One approach to using system calls and external actions is presented by Blundell, Lewis, and Martin[BLM06b]. The authors built a hardware-based transactional memory system that supports system calls via irrevocability.

The transactional memory system distinguishes between a restricted and an unrestricted mode of execution. A *restricted transaction* is highly concurrent, but limited by the constraints of the underlying hardware. It is also unable to perform system calls. An *unrestricted transaction* does not offer high concurrency, but can execute transactions of an arbitrary length and may contain system calls.

All transactions start in the limited, but highly concurrent, restricted mode. If a transaction exceeds this mode's limitations, it is switched to the unrestricted mode, where it is not bound to any limitations. Only one transaction can run unrestricted. The candidate transaction is aborted if there is already another transaction running unrestricted.

Two implementations are presented: a non-optimized and an optimized one. In the non-optimized implementation, once a transaction is running unrestricted, all restricted transactions are stalled. In the optimized implementation, restricted transactions are allowed to run concurrently with the restricted one. Restricted transactions are irrevocable: If a conflict between the restricted transaction and an unrestricted transaction occurs, the restricted transaction always wins.

The support for external actions that is presented in this paper is still quite limited. Especially the all-or-nothing strategy seems to limit high concurrency in the case of many transactions. The works at hand uses an approach where multiple transactions can execute system calls concurrently. It only switches a transaction to irrevocability if it executes an action that cannot be revoked.

An analysis of I/O and syscalls in critical sections and their implications for transactional memory

Baugh and Zilles [BZ07] present a detailed analysis of two widely used applications regarding their use of system calls and I/O within critical sections and the impact for transactional memory. These applications are *MySQL,* a multi-threaded database, and *Firefox,* a popular web browser. Both contain a non-trivial amount of system calls within their critical sections.

The authors first adapt the taxonomy by Gray and Reuter [GR93] to classify actions into (1) protected, (2) unprotected, and (3) real actions. *Protected actions* can be compensated completely by the transactional memory system, *unprotected actions* need some external compensation action, and *real actions* do not have a well-defined compensation action at all. Furthermore, the authors classify the strategies for handling actions as to (i) forbid them completely, (ii) defer them, (iii) go non-speculative, or (iv) compensate them. Refer to Section 1.4 for an introduction of these point together with some examples.

The authors analyzed the frequency in which actions of each class appear in critical sections. They found that most of the calls fall in one of the two first categories, protected or unprotected actions, and can therefore be compensated. With less than 10 percent, only a small number where real actions. An investigation of the distribution of system calls within the critical sections revealed that less then one percent of the critical sections actually execute system calls.

On the other hand, those critical sections that do tend to be very long in terms of duration. This makes it likely that transactions conflict with each other. Also, system calls tend to be called throughout the code and their results are further consumed within the critical section. Deferral is therefore often not an option. There is also a lot of overlapping among transactions, so going non-speculative, which is needed for some calls, results in a substantial loss of concurrency.

The results of Baugh and Zilles indicate that support for external actions in transactional code is at least worth trying. Only few critical sections, respective transactions, need to be considered. Also, only POSIX calls are relevant for most applications, memory-mapped I/O and port I/O were not used. Additionally, most of the calls either need no special handling or can be compensated. Then again, the results also yield some serious problems for the concurrency within applications.

In contrast to the work at hand, Baugh and Zilles only analyze and discuss the impact of system calls on transactions. They do not provide a working solution to the problem. The work at hand reuses the taxonomy; protected, unprotected and real actions; of Baugh and Zilles' paper to classify existing functions of the POSIX standard. It takes the presented approaches for handling external actions and applies them to the POSIX functions. It also discusses common functions and practices in more detail and identifies possible problems when using them in transactional code. In addition, the work at hand provides a complete design and implementation for supporting actions in transactional code.

Transactional boosting: a methodology for highly-concurrent transactional objects

Herlihy and Koskinen describe *transactional boosting* [HK08], a methodology for creating transactional objects from non-transactional, but linearizable objects.

Objects are treated as black boxes with an abstract state and a set of methods, which modify this state. Transactional objects are build by wrapping these non-transactional objects. Every transactional object provides synchronization and recovery.

Synchronization allows the detection of conflicts among methods calls of distinct transactions. Two methods do conflict if the result of their execution depends on the order of their invocation; the methods do not *commute*.

Each method of a transactional object has an associated abstract lock, which handles synchronization. Before a transaction calls a method it has to acquire the method's abstract lock. If any other transaction holds the abstract lock of a conflicting method, the acquiring transaction is delayed, and eventually aborted if it fails to acquire the lock in time.

Abstract locks provide *multi-level concurrency control* for the underlying object: the successful acquisition of an abstract lock is based on the method's semantics. This makes it possible to implement concurrency control beyond low-level read-write conflicts. The paper gives the example of two transactions adding distinct elements to a set. These operations do not have any intrinsic conflicts, yet they generate conflicts when reduced to read-write operations on the memory.

Recovery allows to roll back speculative updates to an object. In order to achieve this, each method m needs an *inverse* method $m^{'}$ that compensates its effects. Each transaction contains a log of inverses. Whenever a transaction executes a method call, the inverse is appended to the log. If the transaction has to abort it walks backwards through its log and calls each inverse method.

Transactional boosting allows for the use of many classes of objects in transactions, especially collection data structures, such as lists, sets, or arrays. Its use is limited to objects where some abstract semantics and commutative methods can be identified. Also, for each method an inverse method has to be available, which limits its applicability for system calls.

When comparing transactional boosting to the work at hand, both use multi-level concurrency control and a history to store executed actions. Transactional boosting seems less abstract in its concepts and less complex to implement. This makes it adequate for the support of data structures, such as the already mentioned collections, in transactional code.

On the other hand, the lower complexity hinders the support of arbitrary external actions substantially. Transactional boosting requires every non-transactional method to have an inverse method, so that it can be compensated in the case of an abort. Many external actions do not have an inverse, but can only be supported by deferral or irrevocability. An example is a write operation on a FIFO, which cannot be undone. In contrast to transactional boosting, the work at hand fully supports deferral and irrevocability; which makes it more complex, but suitable for arbitrary actions.

xCalls: safe I/O in memory transactions

A paper very similar to the work at hand is [VTG+09] by Volos, Tack, Goyal, Swift, and Welc. It presents *xCalls:* a programming interface for transactional support of common system calls, such as file-descriptor I/O, inter-process communication, and threading.

xCalls uses a combination of deferral, compensation, and irrevocability to execute system calls from within transactions. In a study of Linux system calls, the authors found that of 284 calls only 33 need irrevocability, the others can be deferred, compensated, or need no special handling at all. For controlling concurrency on the system calls, the software internally uses *sentinels:* revocable locks that protect kernel resources from conflicting accesses. It implements a two-phase locking protocol for the acquisition and release of sentinels. All sentinels are managed centrally to prevent deadlocks.

xCalls provides a variant of the POSIX programming interface. The programmer is responsible for using these functions instead of the actual POSIX variants. On invocation, each xCalls operation first acquires the required sentinels. Afterwards it either defers the operation, registers a compensation action, or makes the transaction irrevocable.

A simple mechanism for error detection is provided by xCalls. All non-deferred functions return errors within the transaction. All deferred functions provide an additional argument that returns the error state of the call. When the function is finally executed during commit and an error occurs, the error-state parameter is set according to the result of the function. After the transaction finished, the programmer can check the error-state variables to see whether the external calls succeeded. xCalls also allows the programmer to set several internal parameters regarding error handling, such as whether the system should retry any failed deferred call.

xCalls provides several common system interfaces, such as file I/O and file-system functions; socket and FIFO I/O; and threading support.

In the evaluation, the authors first converted the critical sections of three common application to use transactions with xCalls instead. These applications are *Berkeley DB*, a database; *BIND*, a server for the domain name system; and *XMMS*, a media player. The conversion was straightforward and no major problems are reported.

Some performance measurements were done with micro-benchmarks and the converted applications. The results show that transactional I/O can be faster than non-transactional I/O with coarse-grained locking, even though there is a considerable overhead by the use of transactional memory.

When compared with the work at hand, xCalls is very similar: both share the principle ways of handling external actions by deferral or compensation; both provide some facility for handling commit-time errors, and both are contained in user space.

xCalls mostly differs in the lower elaborateness of the presented solution. It works at the level of system calls. The sentinels provide simple lock-based concurrency control on so-called kernel objects, such as file descriptors.[1] Acquiring a sentinel locks out other, concurrent transactions from using the protected resource.

[1]The paper incorrectly speaks of file descriptors as kernel objects.

Protecting access at the level of system calls possibly eases the conversion of low-level critical sections to transactions, because of the low amount of necessary wrapper code. Other functions or complex constraints for the consistency of resources seem hard to be supportable this way. Maybe the sentinel approach is flexible enough to handle them as well, but this is not obvious. The work at hand supports fine-grained, object-level concurrency control and arbitrary functions. Adding new facilities, is likely more work for the programmer than the implementation of a simple wrapper with a sentinel; but allows for concurrent transactions on the same resource, and the implementation of different strategies for protecting each resource.

Another difference is the handling of commit-time errors. xCalls only allows to check for errors after the transaction has finished. With the framework presented in work at hand, a programmer can register callback functions for error handling, and thereby actively influence the treatment of errors during the commit.

Also, it seems that the way in which xCalls handles certain operations can lead to deadlocks or inconsistencies. For example, sentinels protect individual file descriptors. If two file descriptors refer to the same file buffer, distinct transactions can concurrently write to the buffer without synchronization. The xCalls paper does not address such problems.

Related standards and software

The framework's implementation is build upon the ISO C standard [ISO04], the POSIX standard [Ope08], and the GNU C Library [GNU]. Information about common programming practices on Unix systems is given in [Dre09] and [Whe03].

A description of TinySTM and Tanger, the underlying transactional-memory software, is found in [FFR08] and [FFM$^+$07]. TinySTM is a word-based software transactional memory system written in C++. Tanger is a transformation pass for the LLVM compiler infrastructure [LLV] that allows the comfortable use of transactions in C applications.

3 Design

This chapter describes the basic concepts of transactional execution of external actions and the framework's general design. It starts by repeating the requirements for the design, then discusses how resources and operations are abstracted in a transaction-safe fashion, how external actions are performed in general, and finally how to handle errors during a transaction's commit.

3.1 Requirements

In Chapter 1 the requirements for the framework's design were defined as

- correctness,
- standards compliance,
- extensibility, and
- performance.

They are repeated here briefly. Correctness is defined by atomicity, consistency, and isolation. *Standards compliance* requests that an operation executed within a transaction should have the same semantics as its non-transactional counterpart. *Extensibility* refers to the ease of supporting new features. Finally, *performance* minimizes a transaction's response time. An evaluation of the framework with regard to these requirements is presented in Chapter 6.

Before introducing the design it is important to discuss isolation in more detail, as it defines how transactions correctly interact with each other, with non-transactional code in the same process, and with the rest of the system.

Isolation among transactions

Serializability has been chosen as the correctness criteria for isolating concurrent transactions that access shared resources within the same process. The execution of a set of concurrent transaction is modeled by a history H. H is a partial order of operations executed by the transactions. H is *serializable* if it is equivalent to a serial history $H^{'}$: a history of the serial execution of the same set of transactions [BHG87].

The equivalence between H and $H^{'}$ can be described in several ways. The two common ways are view equivalence and conflict equivalence. H is then called *view serializable,* respectively *conflict serializable.*

A history H is *view serializable* [Yan84] if all read operations of all transactions in H observe the same values of data items as in a serial history $H^{'}$. The problem of determining view serializability is NP-complete, so it is not feasible for real use.

Conflict serializability is more useful in practice. A history H is *conflict serializable* if there exists a serial history $H^{'}$ with the same order of conflicting operations [BHG87]. This is a special case of the more general view serializability. Conflict serializability can be guaranteed easily by serializing transactions with respect to detected conflicts.

Given the computational complexity of view serializability and the relative ease of implementing conflict serializability, the framework uses conflict serializability for its components.

Isolation with non-transactional code

Isolation also defines how the effects of a transaction interfere with concurrent non-transactional code in the same process. For transactional memory there are two principle choices: strong and weak isolation [BLM05].[1]

Strong isolation automatically converts all operations outside of a transaction into individual transactional operations [LK08]. Each non-transactional operation runs irrevocable and serialized with respect to concurrent transactions or operations.[2] So either a complete transaction or a single, non-transactional operation is executed. While this seems practical at first, it is rather useless for external actions. Assume non-transactional I/O code that (1) seeks to a specific offset in a file, and (2) reads at this offset. Obviously the programmer's intent is to execute these two statements atomically. But as both are independent, a transaction can commit an update to the file offset between their execution. In this case the read occurs at the newly committed offset instead of the one intended by the programmer.

The problem is that the program executed a set of stateful calls, `lseek` and `read` in this case, and depends on the assumption that this state does change in between. The only solution to guarantee this is to convert the non-transactional code to transactional one. Strong isolation does not provide any benefits here.

With *weak isolation,* a conflicting operation executed outside of a transaction may not follow the protocols of the transactional memory system [LK08]. Therefore, when using weak isolation, transactional and non-transactional code may not use the same resource concurrently. The application programmer is responsible for ensuring that this constraint holds. This is done by converting the non-transactional code to transactional one, or organizing the program in such a way that the non-transactional code never executes concurrently with the transactional one. When changing between transactional and non-transactional code, there has to be a period of *quiescence:* a period without pending method calls for the resource.

In any correct, non-transactional program, access to shared resources is either executed in non-concurrent parts of the code, or protected by the use of critical sections or lock-free algorithms. As reported in [VTG$^+$09], the transformation of such programs to weakly-isolated, transactional code is straightforward. In most cases the programmer only has to replace the critical sections and lock-free algorithms with transactions.

There exist some odd cases where the transformation from non-transactional to transactional code is not easily possible [BLM06a]. This can happen if non-transactional code needs to publicize its state to the outside; such as concurrently running code or external processes. If necessary and possible, a transaction can allow such behavior by violating the isolation rules. An example are consumer-producer scenarios, where a transaction might need to publicize an intermediate result. A counterexample is a file in write-only mode. To roll back writes to the file it is necessary to first read its previous content, which is not possible.

As an additional constraint, any asynchronous function calls are explicitly excluded here, as they might break isolation among transactional and non-transactional code. The most prevalent example are Unix signals. A *signal* is a mechanism by which a process or thread may be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes. The term signal is also used to refer to the event itself [Ope08]. For example, an alarm signal that was arranged in non-transactional code might be delivered while the thread is running transactional code. Under the assumption it executes in a non-transactional environment, the signal-handler function easily breaks the transaction.

[1]The cited paper uses the term *atomicity* instead of *isolation.*

[2]The original definition of strong isolation, given in [BLM05], does not require serializability, but only safe access to values shared with concurrent transactions.

The rest of this work uses weak isolation among transactional and non-transactional code blocks when accessing process-local resources. In practice, this is almost equally useful as strong isolation, but easier to implement. The application programmer has to ensure that transactional and non-transactional code does not overlap. Asynchronous function calls are prohibited.

The framework does not attempt to isolate transactions when accessing shared resources of the system. This seems preferable for external actions, as it allows for the support of inter-process communication. Also, such a facility needs kernel support for implementation.

3.2 Building Blocks

This section introduces the basic concepts on which the framework's design is build: domains, actions, a transaction-local history, and a simple component architecture. Domains and actions have also been described in the (unpublished) paper [RWZ09] by Riegel, Wamhoff, and Zimmermann.

To give you an intuitive notion of these concepts, a common word-based transactional memory could be represented like this: the main memory is a component. Each memory cell is an individual domain. Every domain has two associated actions: read and write. But actions are not strictly methods of domains as each action can work on more than one domain, such as writes to several memory cells at once. The executed read and writes form the transaction's local history; which is the analogue to the read set and write set in a transactional memory system.

3.2.1 Domains

Domains are a concept for representing shared external resources, such as file descriptors or the system's memory allocator, in a generalized and abstract fashion. Each domain provides two properties for its associated resource. These are

- information hiding, and
- independent consistency constraints.

The property of *information hiding* allows the usage of a resource without the need to be aware of the details of its underlying implementation, or its state. The domain of a resource encapsulates the resource and provides interfaces to interaction with it. The transaction then merely interacts with the domain, but not directly with the domain's resource. Domains can use each others resources by calling their respective interfaces; just like transactional code does.

For each domain in the system there is a global state and a transaction-local state. The global state reflects the real state of the underlying resource and contains data structures for doing concurrency control on the resource. For example, a domain that represents an open file might contain the current file offset and a set of locks for serializing access to the file.

The transaction-local state represents the updates of a single, uncommitted transaction to the domain. In the case of the open file, this might be a set of write operations, a new file offset, and a set of domain locks held by the transaction.

Independent *consistency constraints* for each domain allow for the usage of several domains within a transaction without interference among unrelated domains. The consistency constraints of one domain do not affect the consistency constraints of another one. Therefore, each can be treated individually and each can be provided in a fashion best fitting the application's needs.

For example, an open file might allow for the concurrent reading and writing of its content. The transactional memory system thereby detects conflicts for distinct regions within the file. Another domain might represent a FIFO, whose consistency constraints only allow for one reader at a time.

Domains support this by allowing for the implementation of object-level concurrency control for its resources. *Concurrency control* is the activity of coordinating the actions of processes that operate in parallel, access shared data, and therefore potentially interfere with each other [BHG87]. *Object-level concurrency control* provides concurrency control on abstract objects; domains in this case. It can be expressed in terms of *page-level concurrency control,* which operates on individual read and write operation on the storage back-end. A detailed presentation on both can be found in [WV01].

The notions of object-level and page-level concurrency control allow for a formal definition of domains. The page-level consists of a (finite) set D of data items, so-called pages:

$$D = \{x, y, z, ...\}$$

Pages are indivisible and do not overlap. An example of a page is a byte in a file. Each page can possibly be read or written. A domain (1) Dom_i is defined as a set of pages. To guarantee the properties of information hiding and independent consistency constraints (2) distinct domains may not overlap. Formally:

1. $Dom_i \subseteq D$, and
2. if i $\neq$ j, then $Dom_i \cap Dom_j = \emptyset$.

An example of a domain is a file buffer that contains a number of bytes. A notable detail is the role of the transactional memory. In practice, it is handled separately by the transactional memory system; but conceptually it can be seen as just another domain in the system, or a set of memory cells where each cell is represented by an individual domain.

3.2.2 Actions

The primary way of interacting with domains are actions. An *action* is a call to a set of external resources that is invoked from within a transaction.

Action classes

Following Baugh and Zilles [BZ07], actions are to be classified into

- protected actions,
- unprotected actions, and
- real actions.

Protected actions. *Protected actions* only update the state of the thread's processor registers or the main memory. Thus, they can be compensated completely by the transactional memory system. Protected actions can further be classified into stateless and stateful actions.

Stateless actions do not use resources besides the processor and the function call's frame on the stack. They do not need any special handling at all. For example, most of the C standard library's math functions fall in this category.

Stateful actions possibly access memory that is shared with other transactions in the system, typically via pointer arguments or global variables. The affected memory regions have to be announced to the transactional memory system, which then detects and resolves any conflicting accesses to these regions automatically. Examples of stateful actions are the string and memory functions in the C standard library.

Unprotected actions. *Unprotected actions* have the possibility to update some external state and can therefore not be compensated by traditional transactional memory systems. For example; memory allocation, or input and output operations on files fall in this category. To allow unprotected actions within a transaction, either (i) the transaction has be become irrevocable, (ii) the action has to be deferred into the transaction's commit phase, or (iii) the programmer has to provide a special compensation action, which is called during the transaction's abort.

Unprotected actions operate on domains other than main memory. The domain provides the necessary abstraction from the resource to allow for deferral of compensation of the action if possible.

Concurrent, unprotected actions working on the same domain have to be isolated from each other and the non-transactional code of the system. The proposed solution is weak isolation among participants. The application programmer is responsible to ensure that both do not overlap. Section 3.1 discussed this in further detail.

Real actions. *Real actions* can neither be deferred nor compensated, as they typically update some system-wide state. For example, a call to `fork` can not be deferred because the transaction's success might depend on the child process' existence. The call can also not be compensated, as the forked child process is not under the control of the transactional memory system. Simply killing the child in the case of an abort is not an option as this might lead to inconsistent resources, such as incompletely written files. Real actions can therefore only be supported in a transactional fashion with the cooperation of all participating processes and the operating system. Today's common operating systems do not provide this facility, so the only strategy to reliably support real actions is by making the calling transaction irrevocable.

Transactional execution

Typically each action encapsulates one non-transactional function. For example, there might be actions to read and write the content of a file domain. To support transactional execution, each action provides three methods for its associated function. These methods are:

- execute,
- apply, and
- undo.

These methods allow for the implementation of the previously proposed solutions for handling external actions: deferring or compensating an external action, or making the transaction irrevocable. The process of running an action from within a transaction is divided into two phases.

1. Invoking an action from within a transaction results in a call to the action's *execute* method. Its purpose is to enable the transaction to handle the action. This might include setting up some internal data structures, and preparing the action's deferral or compensation. It can also make the transaction irrevocable if necessary.

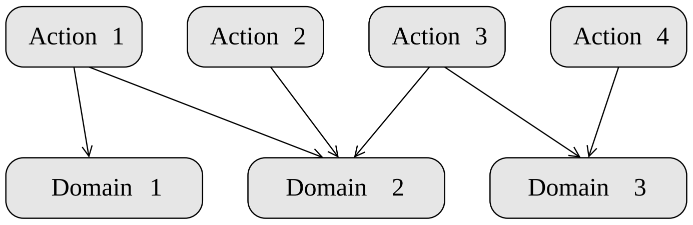


Figure 3.1: The relationship among domains and actions. Each action works on one or more domains, and each domain supports one or more actions.

2. a) The action's *apply* method is called during the commit phase of a succeeded transaction. It finishes the processing of the request by making the action's effects externally visible.
   b) The action's *undo* method is called during the abort phase of a failed transaction. This allows the action to cleanup occupied resources and revoke any externally visible effects.

There is an *m:n* relationship among domains and actions. Domains can have more than one action[3] and actions might depend on the state of more than one domain.[4] Figure 3.1 illustrates this. The nature of a domain's actions depends on the type of resource that is represented by the domain. The concrete activity of each action's methods depends on the nature of the action, the domains' strategy for concurrency control, and possibly other factors, such as some extra options set by the application programmer. For example, when executing file transactions speculatively, it is necessary to defer the effects of write actions to the apply phase, but if a transaction is running irrevocably it can make write actions externally visible during the action's execute.

Coming back to object-level concurrency control, an (1) action p that is executed on a set of domains $Dom_1, ..., Dom_n$ is equivalent to a partial order of read and write operations, $r[x]$ and $w[x]$, that are executed on the page-level data items x of these domains. The (2) order is established by a relation $<_p$ among conflicting operations. Formally:

1. $p[Dom_1, ..., Dom_n] \equiv \{r[x], w[x] \mid x \in Dom_1 \cup ... \cup Dom_n\}$, and
2. if $r[x], w[x] \in p$, then either $r[x] <_p w[x]$ or $w[x] <_p r[x]$.

Object-level concurrency control allows for taking an action's semantics into account. For example, two transactions T_1 and T_2 both execute a single `write` action on the same file. The `write` depends on the file offset and increments its value by the amount of written bytes. This is done implicitly, such that none of the transactions T_1 and T_2 has any actual dependency on that value. With object-level concurrency control it is possible to model the consistency constraints of the domain in such a way that both transactions are in a consistent state as long as they do not explicitly depend on the file offset; like they do after calling `read`. Hence, T_1 and T_2, can commit their `write` action without conflicting. With concurrency control at the page level, such optimization cannot be provided.

[3]A domain without actions is useless in practice.
[4]An action that depends on zero domains is a constant function.

3.2.3 Transaction-local history

Each transaction contains a history that holds external actions that were executed within the transaction. This section shows how this history is integrated with memory operations and how external actions are ordered.

[BHG87] defines a page-level transaction T_i, such as in a word-based transactional memory system, as a partial order with the relation $<_\mathrm{i}$ among conflicting operations. A transaction (1) consists of read, write, commit, and abort operations; (2) (3) its last operation is either a commit or an abort; and (4) if a transaction contains a read and a write operation to the same data item one precedes the other. Formally:

1. $T_i \subseteq \{r_i[x],\ w_i[x] \mid x \text{ is a data item}\} \cup \{a_i,\ c_i\}$;
2. $a_i \in T_i$, iff $c_i \notin T_i$;
3. if t is c_i or a_i, for any operation $p \in T_i$, $p <_\mathrm{i} t$; and
4. if $r_i[x],\ w_i[x] \in T_i$, then either $r_i[x] <_\mathrm{i} w_i[x]$ or $w_i[x] <_\mathrm{i} r_i[x]$.

The terms $r_i[x]$ and $w_i[x]$ denote read and write operations of T_i on a data item *x;* c_i and a_i denote commit and abort operations; p is a memory operation; and t is a commit or abort operation. To integrate external actions, this definition is extended slightly:

5. If p and q are conflicting operations and *p,* $q \in T_i$, then either $p <_\mathrm{i} q$ or $q <_\mathrm{i}$ *p;* and
6. if $p[x]$ is $r_i[x]$ or $w_i[x]$, and $p[x]$, $alloc[x]$, $free[x] \in T_i$, then $alloc[x] <_\mathrm{i} p[x]$ and $p[x] <_\mathrm{i} free[x]$.

The first addition (5) generalizes the notion of conflicts from memory operations to arbitrary operations p and q on memory or external resources. This integrates the previously defined domains and actions.

The second addition (6) describes the relationship among memory operations and external actions. The term $p[x]$ denotes a page-level operation on a memory data item *x,* $alloc[x]$ is an external action that allocates *x,* and $free[x]$ is an external action that frees *x.* The external actions update the state of the heap memory. This affects the behavior of subsequent memory operations on the affected locations. Therefore, allocation has to be done before any access at the affected location takes place, freeing memory has to be deferred until all access took place. This special case is necessary because it is not obvious that operations on the memory-allocator domain create conflicts with reads and writes on memory cells.

What is still missing is the integration of the actions' execute, apply, and undo methods. With the presented design, memory operations are handled separately from external actions: Memory operations are handled by the transactional memory system, external actions are handled by the framework. For every transaction T_i the framework adds a local history E_i. It should not be confused with a history *H.* E_i is the set of external actions p that were executed within T_i and need deferral or compensation, such as write or seek operations on a file. Other actions, such as offset-independent read operations, do not need to be contained in E_i. Formally:

$$E_i = T_i \cap \{p \mid p \text{ is an invocation of an external action}\}$$

The transaction-local history allows to traverse conflicting external actions in the correct order. To apply actions in the correct order, the elements in E_i have to be traversed in ascending order with respect to $<_\mathrm{i}$; to undo actions the elements have to be traversed in descending order. Actions that do not support deferral or compensation make the transaction irrevocable. In this case the content of E_i and all further actions are applied immediately.

3.2.4 Components

Critical sections can handle arbitrary data structures. For transactional memory to be equally useful, a flexible way of integrating new features into existing transactional code is needed.

The framework therefore provides a simple component architecture. Each *component* provides a set of related domains and actions. The component thereby acts as mediator between its domains and actions, and a transaction's local history. For example, it provides internal data structures that contain the local states of each transaction's domains and updates done by executed actions.

To integrate components with the history E_i, the definition of E_i is slightly changed: it does not contain actual external actions p, but generic events e that signal invocations. Each component contains a translation function f_{comp} that maps the events of the component to invocations of external actions. Formally:

$$E_i = T_i \cap \{e \mid f_{comp}(e) \text{ is an invocation of an external action}\}$$

Components communicate with the transaction's local history by a set of generic interfaces. This allows a component's actions to insert events into the transaction's history; and the framework to validate domains, and apply or undo actions.

3.3 Framework

The previous section presented the framework's building blocks: domains, actions, a history for each transaction, and a simple component architecture. In this section these blocks are put together to form the complete design, which is illustrated in Figure 3.2.

3.3.1 Outline

The base of the framework is the transactional memory system. It is hooked into the memory access of all running transactions. The transactional memory system internally checks for conflicting memory operations among the transactions and resolves them automatically. If two transactions conflict, one of them is aborted, or at least has to wait for the other transaction to disappear. To integrate external actions, the transactional memory system exports an interface to

- make it validate the consistency of the transaction's memory and external domains, and
- make a single transaction irrevocable.

Optimistic domains have to be validated to ensure their consistency. Irrevocability is necessary for the support of real actions: actions that can neither be deferred nor compensated.

The central part of the framework is the transaction's local history. It is build on top of the transactional memory system. It exposes an interface to

- make it validate the consistency of the transaction's external domains,
- lock and unlock domains used by the transaction,
- apply all external actions that have been deferred, and
- undo all external action that need compensation.

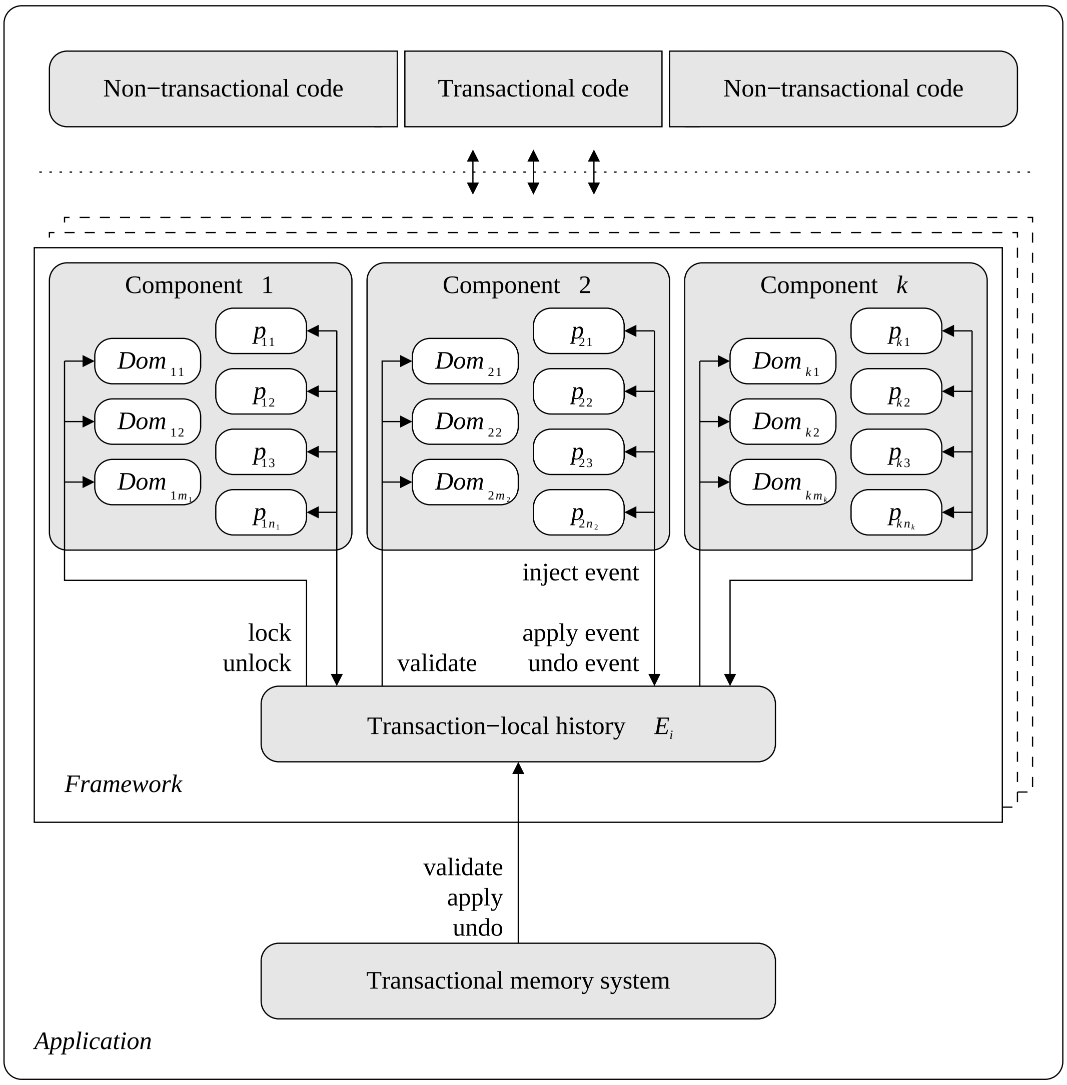


Figure 3.2: The design of the framework is founded on the transactional memory system. On top resides a transaction-local history E_i of action-related events, and a set of components. Each component provides a set of domains *Dom* and actions p. The transaction's application logic executes actions on the domains. The actions inject events into the history. During the transaction or the transaction's commit, the transactional memory system validates the transaction's domains. To support atomic commits, domains are locked before the commit and unlocked afterwards. For committing external actions, the transactional memory system instructs the history to apply all events' actions, otherwise to undo them.

The transactional memory system uses this interface during a transaction's commit and abort to control the external domains. Validation is necessary to guarantee consistency, and the apply and undo methods handle external actions' deferral and compensation. The lock and unlock interfaces are used to implement atomic commits. It is the minimal set of functionality needed, actual implementations might need additional interfaces, such as clean-up functions.

Components are build on top of the history. Each is registered with the history to allow the component to insert events into the history, and the history to call functions from the component's exported interface. The components interface provides means to

- validate the consistency of the component's domains that are in use by the transaction,
- lock and unlock the component's domains,
- apply an event that was added to the history by the component, and
- undo an event that was added to the history by the component.

Again, this is a minimal set of functionality. Any implementation is likely to need more interfaces for specialized tasks.

Each component encapsulates domains and actions. The interfaces for validation and locking are related to domains, whereas the interfaces for applying and undoing are related to actions. So each domain provides its component with an interface that allows for the locking and validation of the domain. Each action provides its component with an interface that allows for the application of the action, in case it was deferred, and an interface that allows to undo the action, in case it needs be compensation. This follows the discussion of domains and actions in Sections 3.2.1 and 3.2.2

3.3.2 Domain validation

Validation ensures the consistency of an optimistic domain's transaction-local state. This can become necessary during an action's execution, for example to check whether a read from a file returned consistent data; or during a transaction's commit, before applying the transaction's deferred action's.

A pessimistic domain provides valid consistency once its respective data item has been first accessed by the transaction. An optimistic domain's transaction-local state can become inconsistent if another transaction updates the domain's global state. Therefore, optimistic domains have to be re-validated after each successive read operation. Assume two transactions T_1 and T_2. Transaction T_1 reads two data items x and y. The read on x is executed on a domain with optimistic concurrency control. The operation is first performed, then the consistency of the read data is validated. Before the read on y is performed, T_2 commits an update to x and y. This is possible for T_2, because x uses optimistic concurrency control. If T_1 now reads and validates y only it holds data in an inconsistent state: the old value of x and the new value of y. To prevent this from happening, T_1 reads and validates y, and afterwards validates all of its optimistic domains, which is x in this case. Data item y could be a memory location, so read operations within the transactional memory system use this strategy as well.

Validation is triggered from within the call of an action's execute method, a memory read, before becoming irrevocable, or during a transaction's commit. It starts within the transactional memory system. This calls the validation interface of the transaction's history. The history contains a list of components that were used from within the transaction: exactly those components that inserted events into the history.

For each of these components the history calls the respective validation interface and each component calls the validation interface of those domains that have been used within the transaction: exactly those domains on which actions have been executed. The independent validation of domains is possible because each domain has its own independent consistency constraints, as defined in Section 3.2.1. Memory validation is handled completely within the transactional memory system. If the validation fails for at least one domain, the transaction aborts.

As an example, assume a file domain that is used by several reader and writer transactions. Each transaction contains a history of events of external actions that have been executed from within. For validation, the history obtains the components of these events and instructs each component to validate the domains used by the transaction. A component then instructs all its domains to validate their transaction-local states. Pessimistic domains need no validation. Optimistic domains have to validate all of the file's content read by the transaction, and possibly the file offset. This succeeds if no updates to these data items have been committed by another transaction. It fails if any data item has been updated. The results of each domain's validation is returned to the transactional memory system, which decides whether the transaction can proceed or has to abort.

3.3.3 Execution of actions

A call to an action from within a transaction invokes the action's execute method, which prepares the possible deferral or compensation of the action and inserts an event into the history if necessary.

When an action is executed that can neither be deferred nor compensated, the execute method requests irrevocability for the transaction. If this does not succeed the transaction aborts.

The execute method is also responsible for controlling concurrency and validating the consistency of any retrieved values. The exact strategy is specific to the action and its domains. Typically this means for the transaction to acquire some locks for a domain, or compare a domain's transaction-local timestamps to its global timestamps.

3.3.4 Commit of transactions

The design of the framework has to ensure the atomicity of a transaction's commit and the consistency of the committed values.

The commit starts within the transactional memory system. It first calls the lock interface of the transaction's history. This in turn calls the lock interface of all components that participated in the transaction, and the components lock the transaction's domains. The locking can be fine-grained, to allow for the parallel commit of unrelated transactions on the same domain. The locking signals all domains that a commit starts and is mostly necessary for domains with optimistic concurrency control. The pessimistic domains have lock semantics by default. Locks have to be acquired in a fixed order to prevent deadlocks. This order is established by locking all components in a fixed order, making each component lock its domains in a fixed order, and making each domain lock its data items in a fixed order.

After all locking is finished, the commit continues with the validation of the transaction's memory and external domains, as described in the previous section. If the validation fails, the transaction releases its locks by calling the history's unlock interface, and aborts.

If the validation has been successful the transactional memory system first commits the transaction's memory-write operations. This has to be done before the commit of any external actions, because the memory operations might work on resources that are unavailable afterwards, such as dynamically allocated memory regions that are freed by an external action's apply method.

After the memory write set has been committed, the transactional memory system starts to apply the transaction's external actions by calling the history's apply interface. The history iterates over the set of events in the order of their appearance within the transaction. For each event it calls the apply interface of the component that added the event. The component calls the apply interface of the action that generated the event. The action's apply method finally commits any deferred updates to the affected domains. It is assumed that both, event and component, store information to allow for this lookup, such that for each event the correct action's apply method is called. Section 4.3 discusses this in more detail. The application programmer can assign call-back functions for handling errors that occur during applies. This is discussed in Section 3.4.

To finish the commit, the transactional memory system calls the history's unlock interface, which releases any locks acquired by the transaction.

3.3.5 Abort of transactions

The consistency of a domain could not be verified if its validation fails. The affected transaction has to abort.

The abort process starts in the transactional memory system by reversing the effects of the transaction's memory operations Afterwards the transactional memory system calls the transaction-local history's undo interface. The history iterates over the set of events in reverse order and calls the undo interface of each event's component. The component in turn calls the undo interface of the action that generated the event. The action's undo now compensates the effects of any updates to a domain made during the action's execution.

Locking is not necessary during an abort if no updates have been made to a domain's global state. If updates have been made that need to be revoked, the action's undo methods can internally lock individual data items while reversing the updates.

3.4 Error Handling

One major problem of transactional execution is error handling at commit time. When deferring actions, it can happen that an error occurs within the action's apply method. The problem is that the action itself takes place within the transaction, but the error occurs during the transaction's commit. This is a difference to non-speculative execution where errors are reported immediately; like in traditional POSIX systems.

Sometimes it is possible to prevent commit-time errors from happening in the first place. For example, an action's execute method can detect whether the supplied parameters are valid and fail with an error code if so. Or the execute method might try to pre-allocate necessary resources, such as disk space for write operations.

In most cases it is not possible to reliably detect errors during an action's execution. For example, the success of a write operation to a file cannot be guaranteed until the write actually happens within the action's apply method. If the apply method fails, the application has to

- be informed that an error happened, including an error description,
- be informed which invocation failed, and
- decide on how to proceed.

To inform the application that an error occurred, the framework provides a simple callback mechanism, were the application programmer sets a callback during a transaction's execution.

The callback setup inserts an event into the transaction's history. When the history is replayed during the commit, the setup event's apply method makes the event's callback the current error handler. The framework also provides a way for removing the error handler, so that it does not handle errors of unrelated actions.

If an error occurs, the framework calls the last applied error handler with the event's parameters. The parameters are implementation-specific. Section 4.5 discusses this in detail. In the case that no error handler is set, a default strategy is applied; like exiting the process.

When the error handler is invoked, the application processes the error. It could, for example, output an error message. When finished, it decides on how to proceed further. This decision is passed back to the framework via the error handler's return value. Some possible strategies include

- EXIT,
- AGAIN,
- ABORT, or
- IGNORE.

In this example, returning EXIT simply terminates the program with a failure code. AGAIN signals the framework to retry to apply the action. This is useful to handle temporary failures, such as full disk drives. ABORT aborts the transaction, and IGNORE makes the framework ignore the error and go on with the next event in the history.

As an example for error handling with this facility, assume the apply method of a deferred write operation fails with `ENOSPC`, which reports that there is no space left on the disk. The write's apply saves the error code in some component-internal data structure, and signals the occurrence of an error back to the history. The history calls the transaction's currently registered error handler. The error handler checks the supplied parameters to see which invocation failed, and reads `ENOSPC` from the thread's `errno` variable. To handle the error, it tries to free some disk space by deleting obsolete files, such as temporary files or local caches. Afterwards it sets a bit in its internal data structures to marks the invocation as failed, and returns AGAIN. When receiving AGAIN, the framework tries again to apply the action. If this succeeds, because enough disk space has been freed by the error handler, the history continues by applying the next event. Otherwise, if the action's apply fails again, the history again calls the current error handler. During its previous run, the error handler has already marked this invocation as failed, so it could now return IGNORE to ignore the error or EXIT to exit the process.

In the case that transactions need to handle errors that occur during aborts, a similar scheme could be used. The programmer thereby registers functions for handling abort errors. The push and pop operations on the error-handler stack would be contained in the respective actions' undo methods, so that they are invoked during an abort.

The provided way of handling commit-time errors is very flexible. It allows for the use of more than one error handler within a transaction, or even to provide each action invocation with its own, specialized error handling. Components can provide a specialized error handler for each of their actions. It is also conceivable that an action's execute method automatically registers the action's respective error handler. The execute method therefore inserts three events into the history. It first adds an event to set the error handler, then it adds an event for the action itself, and finally it adds an event to remove the error handler.

The problem with this approach to error handling is that the application developer still has to do most of the necessary work. For ideas on how to further automatize error handling and making it transparent to the application, refer to Section 7.3.

4 Implementation

This chapter describes *Taglibc,* the Transactional GNU C library, which is the framework's implementation. It is a set of tools and wrappers to allow the use of external actions from within transactions. Nested transactions are not supported.

Taglibc is completely contained in user space and no kernel changes have been made. However, kernel support would greatly ease the implementation of user-space transactions in some cases. A transaction could start system calls, and have the kernel handle the details of the call and the scheduling of transactions. Additionally, kernel support could to some extend avoid the occurrence of commit-time errors, as the kernel can often give guarantees for the success of deferred operations. Support from the operating system also allows the isolation of concurrent processes in the system when interacting with system-wide resources, such as the file systems. A proposal of how kernel transactions can be supported and integrated with transactional memory is found in [PHR+09].

The big disadvantage of a kernel-based implementation is (the absence of) portability. Supporting the same set of features on a wide range of operating systems likely requires a specific implementation for each system. A user-space solution, like presented in this chapter, can be ported with less effort, ideally by simply recompiling for the new system.

The chapter starts by presenting the underlying transactional memory system and how it is connected to Taglibc. Afterwards it describes Taglibc's internal implementation, its connection to the application, and the error handling.

4.1 Basic Setup of Transactions

The framework uses the C++ implementation of *TinySTM* [FFM+07], a word-based transactional memory system for C applications. *Tanger*, a transformation pass for the intermediate representation of the LLVM C compiler [LLV], is provided together with TinySTM. It is executed during the build process of the application where it transforms normal byte code into a transaction-aware form. The build process is outlined by the following steps.

1. All source files are compiled to LLVM byte code and linked together to form one large file. The byte code contains function calls that signal the beginning and ending of transactions.

2. LLVM runs transformation passes on the byte code, including Tanger. This replaces the non-transactional code by transaction-aware code. When the Tanger pass has finished, all memory references within transactions have been replaced by calls to TinySTM's programming interface, and some code for setting up and committing transactions has been added.

3. Finally, the transaction-aware byte code is converted to machine code, and linked with TinySTM as well as other external dependencies. This creates the final application.

Tanger is not strictly needed to use TinySTM, but it adds a lot of comfort. Otherwise, the application programmer had to use TinySTM's interface directly.

4.2 Interaction between TinySTM and Taglibc

Taglibc is the framework's prototype implementation. It is a set of wrappers and tools to support external actions. Taglibc is written in C, an external code generator is implemented in Perl.

When a transaction calls its first external action, Taglibc registers itself with TinySTM. In addition to the standard STM interface for memory operations, TinySTM exposes a few new interfaces to allow for the integration of Taglibc. These are

- `bool tanger_stm_is_noundo()`,
- `bool tanger_stm_go_noundo()`,
- `void tanger_stm_abort_self()`,
- `bool tanger_stm_validate()`,
- `void tanger_stm_store_mark_written(offset, length)`,
- `void tanger_stm_set_ext_calls(<callback functions>)`,
- `void tanger_stm_set_optcc(bool)`,
- `bool tanger_stm_get_optcc()`,
- `void tanger_stm_set_validation_mode(mode)`, and
- `mode tanger_stm_get_validation_mode()`.

The function `tanger_stm_is_noundo` is provided for querying the irrevocability state of a transaction. The function `tanger_stm_go_noundo` allows a transaction to request irrevocability. If the transaction became irrevocable, the call returns success, otherwise it returns an error code. The transaction is then responsible for cleaning up its resources and aborting itself. Aborting is achieved by calling `tanger_stm_abort_self`.

It is sometimes necessary to re-validate the consistency of domains with optimistic concurrency control. Data read from such a domain can hold inconsistent values when used in a transaction. This happens if an update is committed between the time of the validation and the actual use of the read value. An example is given in Section 3.3.2. Optimistic domains announced their presence of to TinySTM with a call to the function `tanger_stm_set_optcc`. The function `tanger_stm_get_optcc` retrieves this value. As long as no optimistic domains are present, validation is not performed.

In the case that optimistic domains are in use, the function `tanger_stm_set_validation_mode` allows to limit the amount of validation. Three modes are currently supported:

- OP,
- DOMAIN, and
- FULL.

The value OP, which is the default, signals that only the result of the current action's operation needs to be validated, DOMAIN signals that only the optimistic domains on which the action was executed need to be validated, and FULL signals the validation of all optimistic domains. This allows the application programmer to reduce the overhead of validation if inconsistencies can be tolerated.

An action, executed on optimistic domains, retrieves the current validation mode with a call to `tanger_stm_get_validation_mode`. If the mode is OP or DOMAIN the action instructs its domains to validate the operation's values or themselves; if the mode is FULL the action calls `tanger_stm_validate` to instruct TinySTM to start validation of all optimistic domains that are in use by the transaction. TinySTM then validates the transaction's memory and instructs Taglibc to validate its domains. This is done via function callbacks, which are described below.

During commit TinySTM always validates all domains, so even if a transaction can execute on inconsistent values, it cannot commit any inconsistent results.

The call `tanger_stm_store_mark_written` originally was an internal interface of TinySTM, but has been made public. It allows a transaction to request ownership of a region of main memory. This is necessary for memory management.

Finally, `tanger_stm_set_ext_calls` allows for the connection of Taglibc with the memory transaction. This function takes a set of hooks, which are called by TinySTM during the lifetime of the transaction. These hooks are

- `void lock()`,
- `void unlock()`,
- `bool validate()`,
- `void commit()`,
- `void abort()`, and
- `void finish()`.

All hooks are implemented as function pointers, where each transaction has its own set of pointers.

Calls to the hooks `lock` and `unlock` enclose each transaction's commit phase, where they allow for the locking and unlocking of framework-internal data structures and optimistic domains.

The `validate` hook is called to validate the state of the domains with optimistic concurrency control at the beginning of the commit process, after a transaction became irrevocable, or after a read operation. It instructs Taglibc to validate the consistency of the transaction's external domains. Taglibc knows about the components that inserted events into the history. These components in turn keep track of the state of their domains. The return value signals the success or failure of the validation. Memory validation is handled by TinySTM itself.

If validation during commit is successful TinySTM calls the `commit` hook, which instructs Taglibc to apply the transaction's pending actions to its respective domains. If the validation is not successful, TinySTM calls the `abort` hook, which makes Taglibc undo all pending actions.

Finally, `finish` is always called at the end of each transaction to clean up remaining framework-internal data structures.

4.3 Framework-Internal Details

This section describes the core of Taglibc, which is based on the changes to TinySTM.

4.3.1 The transaction-local history

The callback functions, used by TinySTM, are provided by Taglibc's history. The history is the lowest level of the software stack within Taglibc. It contains event that have been executed by the transaction. The history is implemented as an *event log* that represents the history of executed actions in the order of their execution.[1]

As illustrated in Figure 4.1, the log is aware of all included components. Similar to TinySTM, it exposes a programming interface for components to interact with. The elements of the interface are

- `void log_register_component(index, <callback functions>)`,
- `void log_inject(component, call, cookie)`,
- `void log_push_commit_error_handler(<callback function>)`, and
- `void log_pop_commit_error_handler(<callback function>)`.

The functions `log_push_commit_error_handler` and `log_pop_commit_error_handler`, are related to commit-time error handling. They are described in Section 4.5.

The function `log_register_component` allows components to register themselves with the log. When registering, each component selects a unique index by which Taglibc distinguishes among individual components. At the moment, indexes are assigned statically. The register function also receives a set of hooks, which allow any component to become part of the transaction. These hooks are

- `void lock()`,
- `void unlock()`,
- `bool validate()`,
- `bool apply_event(events, nevents)`,
- `bool undo_event(events, nevents)`,
- `void updatecc()`,
- `void clearcc()`, and
- `void finish()`

Most of these hooks behave like their TinySTM counterparts. The hooks `apply_event`, and `undo_event` allow for applying or undoing the actions of events in the log. Both functions retrieve an array of events, so that a component that inserted several events in a row can handle them as one. This reduces function call overhead and allows to merge events. For example, the component for file-descriptor I/O uses this facility to merge successive write operations at consecutive offsets in a file buffer.

[1]Saving the undo actions instead raises several problems with functions that have to be deferred in any case. Assume all file writes were committed from within the action's execute method and compensated during an abort. Therefore, the write's execute had to obtain and save the previous values of the written area. This is not possible if the file had been opened write-only. Also, for readable files the extra read operation during the execute would impose a performance overhead. Problems also occur with memory management: It is not possible to regain a once-freed memory region during an abort.

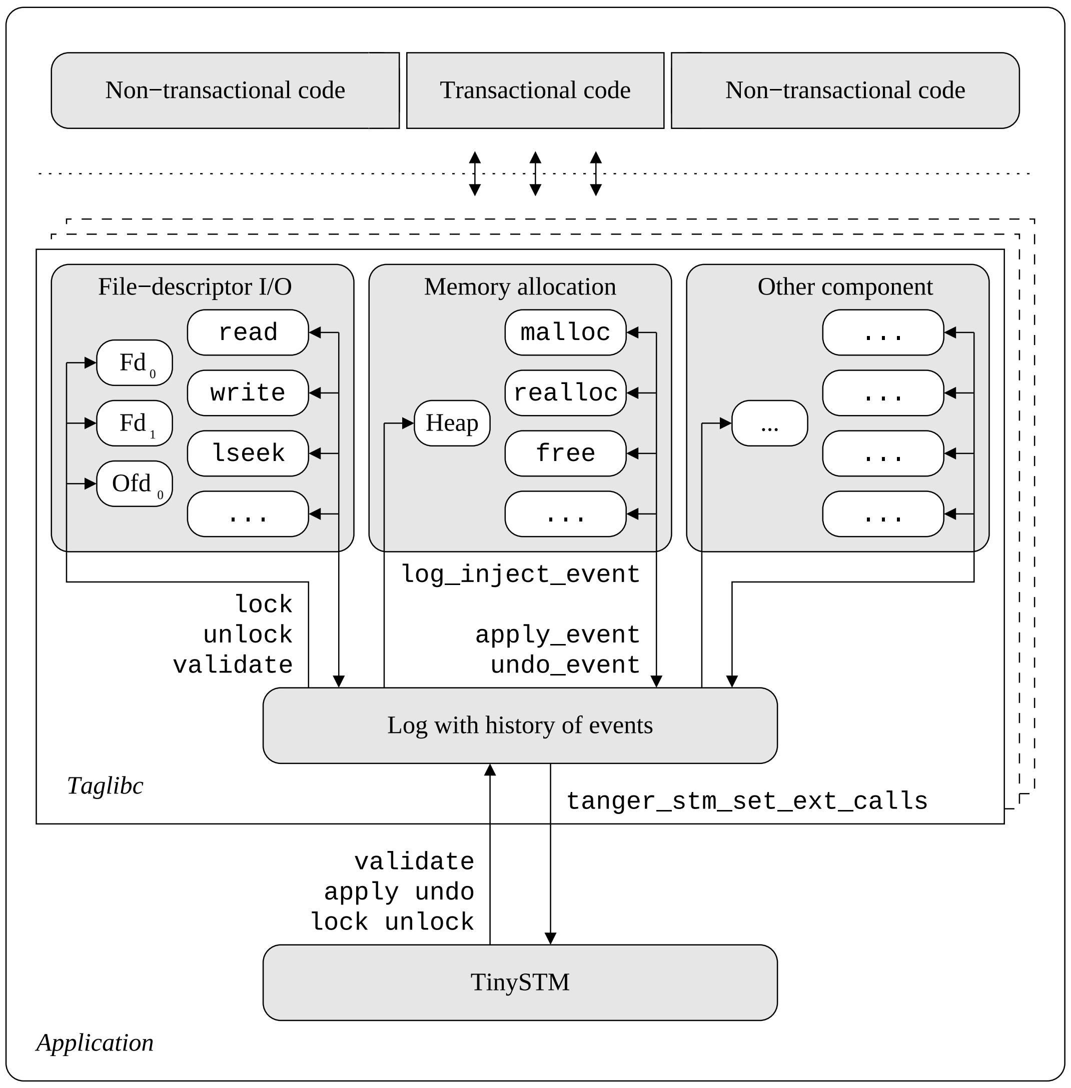


Figure 4.1: The interaction within Taglibc for the most important interfaces. The implementation is based on TinySTM. Each transaction's event log contains the history of events. It connects with TinySTM using `tanger_stm_set_ext_calls`. Several components, such as file-descriptor I/O and memory allocation, are provided with Taglibc. Each component provides a set of domains and actions: transaction-safe variants of non-transactional resources and functions. The transaction executes actions, which append events to the log via `log_inject_event`. Before the commit starts the optimistic domains are locked by TinySTM via `lock`. During the commit or abort TinySTM uses the `validate`, `apply`, and `undo` callbacks for interacting with the event log. The log itself uses validate, `apply_event`, and `undo_event` to interact with the components. At the end of the commit the domains are unlocked via `unlock`.

The hooks `updatecc` and `clearcc` are related to concurrency control. The former is called at the end of a successful commit to update the state of the concurrency-control data structures; the latter is called at the end of an abort for the same purpose. When registering it is also possible to supply a pointer with component-specific data structures.

Each component is aware of the domains for which it is responsible. After it has registered itself with the log via `log_register_component`, it can start appending events by using `log_inject`. The event receives as its parameters the index number of the calling component, a component-internal index of the executed action, and a component-internal cookie value. These values are later used to map the event to a specific invocation of an action.

4.3.2 Setup of actions, components, log, and transactional memory system

Each component interacts with the framework's event log via the log's exported interfaces and the callback hooks. The code for doing this is located within the component and its actions.

When a transaction executes its first action, neither has the component been registered with the log, nor has the log been registered with TinySTM. The action's execute method therefore sets up some component-specific data structures and calls the log's interface `log_register_component` to register its component. This in turn executes `tanger_stm_set_ext_calls` to register the logging facility with TinySTM. Any further action of this component does not process this setup, but retrieves the component's data from the log. The first action of any other component only has to do its component-specific setup.

Having retrieved the component's data, the action's execute method proceeds with its specific purpose. Afterwards it can instruct its component to append an event to the log by calling `log_inject`. The event's parameters include the component's index, the action's component-intern index, and an invocation-specific cookie. Later this allows to reconstruct the necessary information to apply or undo the action.

The lazy setup is used to minimize the overhead for transactions that do not use external actions, or only use specific components. It also eases the integration of new components, as there is no need to specifically announce them beforehand to the framework; at compile time or application start-up for example.

4.3.3 The commit and abort process in detail

During a transaction's commit phase, TinySTM first attempts to lock all of the transaction's external domains via the `lock` callback. This makes the log's implementation call the `lock` callback of all registered components, which, if necessary, start locking the domains used by the transaction, either partially or completely. Locking prevents concurrent transactions from accessing the domains' fields and ensures atomic commits.

Having locked the domains, the next step is validation. Besides validating its memory locations, TinySTM validates the external domains via the `validate` callback. The log's implementation of the `validate` callback validates all external domains, using each component's `validate` callback. The details of validation are specific to each component and its domains.

If the validation completes successfully, TinySTM first commits its local memory updates and then commits the updates to the external domains. This order is important. Assume that a transaction wrote to some area of heap memory and later released this memory by calling `free`. To make `free` revocable it gets delayed until commit time. If the external actions were committed first, the free'd memory would be released, and afterwards the updates to these now unavailable memory locations would fail to be committed.

A transaction's external actions are applied in order of their execution. The log's implementation of `apply` walks over the transaction's set of events and calls each event's `apply_event` function with the parameters supplied with the event.

Afterwards the log calls `updatecc` to update any data structures that are related to concurrency control. At the end of the commit process all domains are unlocked by the `unlock` callbacks and any remaining resources are cleaned up by the `finish` callbacks.

If a transaction has to abort, TinySTM calls the `abort` callback. Its implementation walks through the history in reverse order and calls each event's `undo_event` callback, and afterwards resets the concurrency-control state by calling `clearcc`. The transaction then starts anew.

4.3.4 Serial mode

TinySTM has been enhanced to allow one transaction to run exclusively. This becomes necessary if a non-revocable action is executed, or if an action encounters a state where revocability cannot be provided.

The interface `tanger_stm_go_noundo` allows a transaction to request serial mode. The function waits until all other running transactions have either committed or aborted. Also, when a transaction has called this function no transaction can start or retry.

As soon as the transaction is running exclusively, is starts the validation and commit process for the domains it has used so far. If the validation and commit succeeds, the transaction applies its external actions and continues exclusively; otherwise the transaction has to undo its actions and start anew, but stays in the serial mode.

4.3.5 Progress and contention management

Concurrent code that uses mutual exclusion is subject to deadlocks. Assume two transactions T_1 and T_2, and two shared resources x and y. T_1 first locks x and T_2 first locks y. Afterwards T_1 attempts to lock y and T_2 attempts to lock x. This results in both transactions blocking each other and none is able to make progress.

The absence of deadlocks is called *freedom from deadlock:* If some thread attempts to acquire the lock then some thread will succeed in acquiring the lock [HS08]. With this principle it is guaranteed that at least one thread always makes progress.

The framework's implementation requires its components to never block while acquiring locks. If acquisition fails, the component should instead abort the transaction. This prevents the case of blocked transactions waiting for each other.

It can still be the case that T_1 and T_2 both repeatedly fail to acquire one of the other transaction's locks after having aborted. This is called *livelock.* To ensure freedom from deadlocks the framework counts the number of aborts per transaction. When a transaction reaches a certain limit, it is executed exclusively. At this point it cannot possibly conflict with other transactions anymore.

The implementation does not guarantee freedom from starvation. *Freedom from starvation* requires that every thread that attempts to acquire the lock eventually succeeds [HS08]. This principle ensures that every thread makes progress. In the framework, when a thread requests to run exclusively, there could always be some other exclusive thread that is preferred.

At the moment the decision which transaction to prefer in the case of conflicts happens by chance: the transaction that detects the conflict has to abort. Ideally it should be made by a *contention manager*: a centralized component that resolves conflicts among transactions. It can use an optimal strategy to select the best transaction to win the conflict.

Listing 4.1: Declaration of a wrapper around `write`

```
static ssize_t
tanger_wrapper_tanger_stm_std_write(
    int fildes,
    const void *buf,
    size_t nbyte) __attribute__ ((weakref("write")));
```

Additionally the contention manager can take various parameters into account; such as transaction length, size of the read and write sets, or the number of previous aborts. The framework does not contain a contention manager, but some ideas are presented in Section 7.1.3.

4.4 Connecting Framework and Application

Having described the changes to TinySTM and the internal structure of Taglibc, the final missing piece is the execution of actions from within transactions. For this to work, non-transactional function calls have to be redirected to their framework-internal execute function, and each call's arguments need to be made transaction-safe. For example, C strings that are passed as arguments need to be announced to the transactional memory system, so that conflicts on them are detected. The call stack is illustrated in Figure 4.2.

The framework's components provide a set of entry points to the application. Each *entry point* provides an interface of the POSIX specification. An application can make use of an entry point by including the respective header files. The framework provides a code generator, which automatically creates the public headers and entry points from some simple declarations. It covers most of the POSIX interfaces that the framework supports. The interfaces that are not covered by the code generator have been added manually. They often have unusual semantics, which are not easy or worth to formalize for automatic generation. An example is the function `posix_memalign`, which returns the address of a pointer via a pointer argument.

4.4.1 Redirection of function calls

The process of linking entry points into an application is a combination of different techniques. A declaration of an entry point uses GCC's function attribute `weakref`, which makes a declaration a weak reference to some target function. When using the non-transactional function's name as target, Tanger redirects all calls from the non-transactional interface to the framework-internal entry point during the byte-code transformation. As an example, the declaration of the framework's entry point for `write` is shown in Listing 4.1. If this declaration is included in the program's source code, a transaction that calls `write` actually calls `tanger_stm_std_write`.

Declarations need to have one of two prefixes. The prefix `tanger_wrapper_` means that the declaration is a wrapper around a framework-internal function; the prefix `tanger_wrapperpure_` means that the external action can be used directly. The latter is especially useful for stateless, protected actions, which can be called directly.

When the byte-code is linked during the last step of the build process, the framework is linked statically into the application. In this program, when a non-transactional function is called from within a transaction, the entry point and the action's execute method are invoked.

Tanger only works on transactional code blocks. Calls from non-transactional code still work as before: any call is linked to its original POSIX interface.

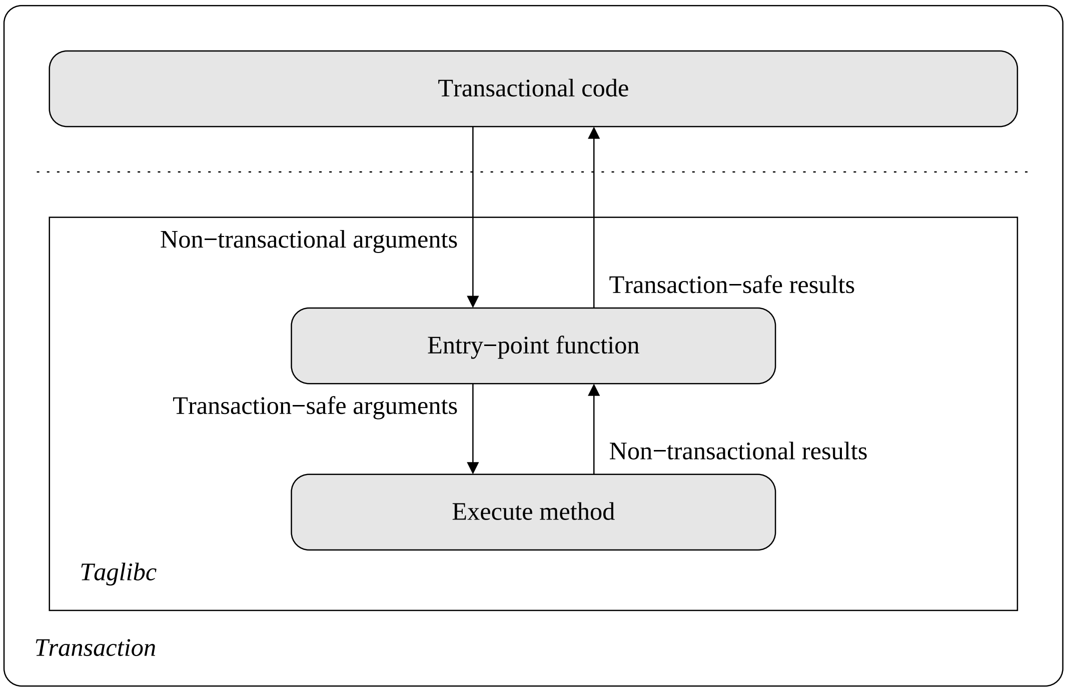


Figure 4.2: A call of an execute method. The entry point is called from within a transaction. It converts the arguments to transaction-safe values before it passes them on to the execute method. On return, it converts the results to transaction-safe values as well. The distinction between entry point and execute method allows for the automatic generation of the conversion code.

4.4.2 Transaction-safe arguments

An entry point's implementation has to check the arguments for validity, regarding concurrent access by other transactions. It is the entry point's responsibility to make TinySTM aware of the access and copy the referenced memory's content to a transaction-local storage. TinySTM provides the necessary interfaces.

The entry point then hands over control flow to an action-specific execute method. Afterwards it announces any pointers that have been returned by this function to TinySTM before it returns to the transaction. For example, a call to `write` receives a pointer to a memory buffer, which has to be written in a transaction-safe fashion.

For saving transaction-safe copies of supplied buffers it is necessary to dynamically allocate memory. If the buffer is not larger than one kilobyte, the code generator automatically applies an optimization where it allocates temporary memory on the thread's stack instead of the heap.

4.5 Error Handling

Commit-time errors are handled by callback functions, which the programmer assigns during the transaction's execution. When an error occurs in an apply method, the apply has to save the error state and return an error indicator to the framework. The framework then calls the currently active error-handler callback.

Listing 4.2: Registering an error handler

```
static enum stm_error_action
error_handler(int component, int call, int cookie)
{
    /* Exit on commit error */
    return STM_ERR_EXIT;
}

static void
do_transaction(void)
{
    tanger_begin();

    /* Push error handler */
    tanger_stm_push_error_handler(error_handler);

    /* Some function calls here
     * ...
     */

    /* Pop error handler */
    tanger_stm_pop_error_handler();

    tanger_commit();
}
```

A callback's arguments are the parameters of the event that generated the error. These consists of the component number, a component-internal call number, and an invocation-specific cookie. These are the values that have been passed when the respective event was injected into the history. They allow the component to retrieve information about the failed invocation. The return value of the callback determines the further processing: ignore the error, try again, or exit the process.

Internally, the error-handling facility is implemented as a stack of function pointers. The public interface `tanger_stm_push_error_handler` allows the application programmer to push a function pointer onto the stack, the interface `tanger_stm_pop_error_handler` allows the removal of the top-most function pointer. When the application developer executes such a function, it either appends a *push* or a *pop* event to the transaction's log. During commit these actions are applied: the specified function pointer is pushed onto the stack or the top-most pointer is removed from the stack. When an error occurs during commit, the current top of the stack is used as the error handler.

Listing 4.2 shows an example usage of the interface. The transaction is implemented in `do_transaction`. It executes `tanger_stm_push_error_handler` for making the function named `error_handler` the commit-time error handler. It then executes some function calls and executes `tanger_stm_push_error_handler` to pop `error_handler` again. If the apply method of a function call fails, the framework calls `error_handler` with the parameters of the event whose function call caused the error. This allows the application to search information about the respective invocation and handle the error. This is done before the transaction is aborted, so it is still possible to continue the commit if the error could be resolved.

With the stack-based design, it is possible to support nested error handlers: functions called from within a transaction can push and pop their own error handlers onto the stack without interfering with the callers error handling.

The error-handler function is not transaction-safe. Thus, it should not attempt to access any shared resources, which might be in use concurrently by transactional code. It should rather use the interfaces provided by components or domains to obtain the transaction's state. Although this is not implemented at the moment, each component can provide a set of standard error handlers for its actions, or provide the application with a way to query extra information about the failed action.

5 Components

The previous chapters only discussed the core of Taglibc. The framework also includes components that provide functionality of the POSIX standard in a transaction-safe fashion. This chapter presents the domains and actions of the available components and covers important details of their implementation.

5.1 Memory Allocation

The memory-allocation component provides transaction-safe allocation of heap memory. Its only domain is the system's memory allocator. A basic version of transactional memory allocation has already been provided by a combination of TinySTM and Tanger. Moving this upwards in the software stack leads to a cleaner and more generic solution.

5.1.1 Actions

The basic primitives are `posix_memalign` and `free`. All other actions, namely `malloc`, `realloc`, or `calloc`, are just variants of these primitives and build upon them.

Posix_memalign. A call to `posix_memalign` allocates memory on the heap. It is executed immediately within the transaction. The address of the allocated memory is stored in the component for use with the log's entry. The undo method of `posix_memalign` frees this memory, the apply method does nothing. In contrast to normal `malloc`, `posix_memalign` allows to set the alignment of the newly allocated memory. When emulating `malloc`, the alignment can be chosen within some boundaries. This allows for an alignment that fits the number of bytes covered by the transactional memory system's internal locks, and thereby minimizes the amount of false sharing of memory regions among independent transactions.

Free. Allocated heap memory is returned to the system by calling `free`. The function `free` cannot be executed immediately from within the transaction, because the memory to be released might have been allocated in non-transactional code. In case the transaction has to restart, there would be no chance of getting back the memory at the specific address. Instead, the deferred apply method of `free` returns the memory to the system. Its undo method does nothing.

An important implementation detail is the interference of `free` with other running transactions. TinySTM exports the interface `tanger_stm_store_mark_written`, which allows to mark arbitrary areas of memory to be written. The execute method of `free` uses this interface to detect conflicts on the supplied memory region.[1] If any other running transaction already uses the referenced memory, the freeing transaction is aborted. This is a problem with irrevocability. If a transaction runs irrevocably and tries to free memory used by some other transaction, it would be aborted. The only way to reliably prevent this from happening is to disallow for any concurrent transactions while one transaction is irrevocable. Thus, an irrevocable transaction is executed exclusively with the current implementation.

[1]The size of the memory region is determined by using glibc's internal interface `malloc_useable_size`.

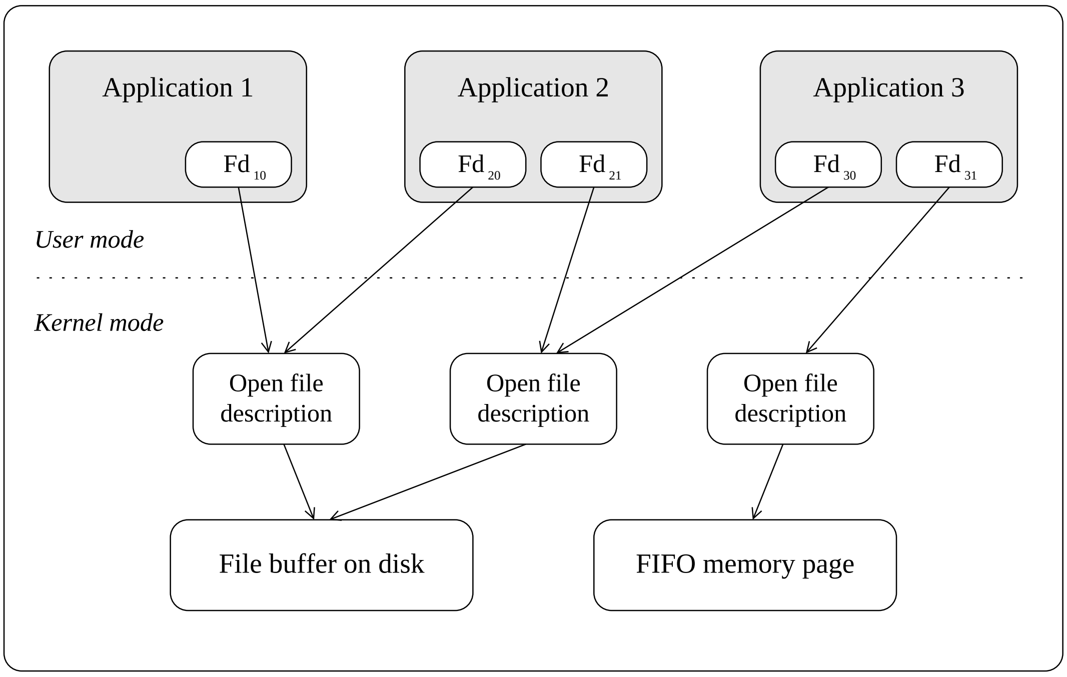


Figure 5.1: The model of file-descriptor I/O in Unix. Each application contains a set of file descriptors, which refer to in-kernel open file descriptions. These in turn refer to low-level buffers, which holds the file content or a FIFO's data. The file descriptors are specific to each application, whereas the open file descriptions and file buffers are shared among all running processes.

5.2 File-descriptor I/O

File descriptors are the low-level handles for file access in Unix. However, file descriptors are not restricted to files only, but are used to access FIFOs, sockets, and other buffers as well. The I/O component strives to isolate the effects of transactions within the same process, isolation from other processes in the system is not provided.

5.2.1 Unix' model of file-descriptor I/O

When it comes to file-descriptor I/O, Unix-like operating systems distinguish between three layers of abstraction. The two lower layers are typically only accessible when running in kernel mode, whereas the top-most layer is accessibly from within an application in user mode. This is illustrated in Figure 5.1.

The lowest layer is a *buffer,* which contains the actual data. This can be a file on the hard drive or a memory page containing the data of a FIFO.

On top of the file buffer is an *open file description.* This is a record containing information about the current access, such as the file offset or read-write mode. It is created by function calls that provide buffers to the application, such as `open` in case of file buffers. An open file description refers to exactly one underlying buffer, but a buffer can be referenced by multiple open file descriptions, which then share the buffer's content.

Strategy	Approach	Implementation	Revocable	Implemented for
NOUNDO	Pessimistic	Global lock	No	All
2PL	Pessimistic	Two-phase locking	Yes	Regular files, writes on FIFOs and sockets
TS	Optimistic	Timestamps	Yes	Regular files, writes on FIFOs and sockets

Table 5.1: Overview of available strategies for concurrency control on file descriptors.

The top-most structure in the hierarchy is a *file descriptor.* It is the only structure that is directly visible to the application and acts as a handle for doing interactions with the underlying open file description and buffer. A file descriptor refers to exactly one open file description, but an open file description can be referenced by more than one file descriptor. For example an application successfully opens a file with a call to `open`. This returns a file descriptor, which refers to a newly created open file description. The application now calls `dup` on the file descriptor. This creates a new file descriptor that refers to the same open file description as the old one.

5.2.2 Concurrency control on files

Before the handling of different file types is discussed, this section first presents the I/O component's options for controlling concurrency on files. Each file buffer is protected by one of three strategies: NOUNDO, two-phase locking, and time stamps. Table 5.1 contains an overview. In addition to protecting the file's content, these strategies also determine of how to handle concurrency on the open file's state, such as the file offset.

The two latter strategies, two-phase locking and timestamps, allow fine-grained concurrency control on the file buffer. In order to achieve this, each file buffer is divided into a set of records. Each *record* spans an area of a fixed size and provides concurrency control for its content.

Noundo. The first strategy, NOUNDO, makes a transaction execute exclusively, and hence needs no file-buffer concurrency control at all. When this has been selected, a transaction doing file I/O requests irrevocability before executing the actual I/O operation. The operation is then executed directly without deferral or compensation, which reduces the overhead of this mode.

The strategy NOUNDO is needed for situations where some action needs irrevocability. It does not allow for concurrent transactions; but supports arbitrary non-transactional operations, since these can simply be wrapped and executed directly. Also, errors are generated immediately, so no commit-time error handling is necessary.

Two-phase locking. The second strategy, 2PL, provides pessimistic concurrency control. It employes *strong strict two-phase locking* [BHG87] for each individual file record. Before a transaction reads a record, it acquires the respective reader lock if no writer is already present. Before a transaction writes a record, it acquires an exclusive writer lock if no reader is present. A reader lock can also be upgraded to a writer lock, but not if other readers are present. The transaction aborts in cases where it cannot acquire a lock because of conflicts.

The actual write operations are deferred into the transaction's commit phase. During the commit, all write operations are applied in the order of their execution and the open file description's global offset is updated. Read operations only need to update the file offset. At the end of the commit, all locks are released.

The 2PL strategy is indented for files where many conflicts are expected or a transaction executed many read operations. It generates false conflicts if concurrent transactions write to overlapping regions in the file buffer. Each transaction tries to write-lock the region. One writer succeeds, whereas the other writers observe a conflict and abort themselves. Real conflicts only occur among reading and writing transactions, so the transactions aborted for no reason.

Timestamps. The third strategy, TS, provides optimistic concurrency control by associating a timestamp with each record. For read operations TS uses eager validation to ensure that the read data is consistent. Before a record is first read, the transaction retrieves a copy of the record's global timestamp that is current at this point in time. Then the read operation is executed. Afterwards the local timestamps of all records in the transaction's read set are compared to their global counterparts. If any global timestamp is higher than its local timestamp, some other transaction committed an update to this record's region in the file buffer. The transaction's snapshot is inconsistent and the transaction has to abort. A detailed discussion on how validation is implemented is given in Section 4.2.

All write operations are deferred and applied during the commit. On commit, first all regions in the read and write sets are temporarily locked to guarantee atomicity. The read set is again validated to prevent inconsistencies. If successful the write set is applied, the timestamp of each written record is incremented, and the locks are released.

A problem of timestamps is their potential overflow. The current implementation uses 64-bit version numbers to minimize the probability of this to happen. To safely handle overflows in all situations, a future revision of the I/O component could handle this explicitly; for example by first aborting all transactions on a domain when the domain's timestamp values reach some upper bound; then setting all timestamps to zero; and afterwards restarting the aborted transactions.

The strategy TS is intended for files where transactions mostly write values, but seldom read. With TS, a transaction only validates its read set. If several transactions write to overlapping regions in the file buffer, none of them has to abort. This prevents a shortcoming in the implementation of 2PL: Writing transactions can observe false conflicts with other writers, due to the eager acquisition of locks. In cases where only few real conflicts are to be expected, the additional work of validation might be passable.

Each file has its own strategy for concurrency control. The strategy is based on the file buffer's type, and can be selected by the application programmer while the buffer is not in use. Any other shared resource that refers to the buffer, such as the file offset in the open file description, inherits the buffer's strategy. Switching a buffer's strategy while it is in use requires coordination among the affected transactions and the conversion of concurrency-control data structures. Because of the inherent complexity of this procedure, strategies cannot be switched once they are established. The only exception is that a buffer selects NOUNDO or one of the executed actions requires irrevocability. This turns the transaction irrevocable and all previous updates on all files are applied. As soon as the transaction is running irrevocably, NOUNDO is used for all actions on all buffers.

5.2.3 Handling of different file types

Unix systems distinguish among a number of different file types, such as regular files, FIFOs, or sockets. This section provides details on the handling of the most important of these types.

Regular files

Regular files are mostly used for storing data within the file system. This includes persistent data, such as data created by the user or some system program, as well as transient data, such as the web browser's local cache file. Regular files also serve as advisory locks to signal the presence of processes within the system. For example, a system daemon might create an advisory lock file at a fixed location in the file system. Whenever a second instance of the daemon is started, it recognizes the lock file and exits immediately, as the service is already running.

Problems. Transactions encounter problems when using regular files for inter-process communication. A deadlock occurs if there is a cyclic dependency among them. This is the case if the transactions read and write to the file buffer, and the output of either transaction is the input of the others.

Assume two transactions T_1 and T_2 communicate by reading and writing within a file buffer. Transaction T_1 is waiting for a message from T_2 by reading from the buffer. Transaction T_2 writes to the file buffer and afterwards waits for an answer from T_1 by reading as well. As described in Section 5.2.6 revocable write operations cannot be compensated in the general case, so they have to be deferred until the transaction commits. Thus, the message's write operation does not happen until T_2 commits, which never happens because it waits for T_1 to answer. So, both transactions are blocked by waiting for each other.

Handling. The I/O component allows for fine-grained control of concurrency on a file's buffer. All three strategies are provided. The described problem of inter-process communication is of little relevance, as it rarely occurs in practice. POSIX systems provide better suited methods of inter-process communication.

Special attention is given to the file offset. The file offset determines the offset in the file buffer where I/O operations take place. There are 3 basic operations that depend on the file offset. These are reads, writes, and seeks. Each seek operation sets the file offset relative to some position, which is either the current offset, the file's beginning, or the file's end; and returns the new absolute file offset. Each read operation returns the content at the current file offset and afterwards increases its value by the amount of read bytes. Both, seek and read, depend on the value of the file offset during their execution. The write operation does not depend on the file offset during its execution, but only during its appliance. In write-only transactions the file offset is not made available within the transaction; neither explicitly via seek nor implicitly via read. Thus, the transaction logic does not see its value.

This allows the framework to enable non-conflicting writes for the TS strategy. Assume two transactions only executing write operations. Both are ready to commit. The framework first atomically commits one transaction at the current file offset. Afterwards it commits the other transactions to the new file offset. In a different case, where a transaction contains read or seek operations, there exists a dependency on the file offset. Such a transaction has to abort if this dependency generates any conflicts.

FIFOs

A *first-in-first-out buffer*, abbreviated FIFO, provides a half-duplex channel for connecting distinct processes in the system. Each FIFO has a read end and a write end. Writing to a FIFO's write end sends data over the channel to the process at the read end, reading from a FIFO's read end receives data from the process at the write end.

Problems. A lot of problems arise when using FIFOs from within transactions. One problem is that reads cannot be compensated. Assume a transaction has read some bytes from a FIFO. If the transaction aborts, the read's compensation action had to return the read data to the beginning of the FIFO's memory buffer, so that it is available when the transaction retries. However, pushing back data to the FIFO buffer is not supported by the operating system.

An imaginable workaround is to store the read bytes in a framework-internal buffer when an abort occurs, and reading them during a retry. This does not work correctly with non-transactional code. Assume a transaction reads some bytes from a FIFO and aborts afterwards. It pushes the read bytes to the framework-internal buffer. During the retry, the transaction reads less bytes from the buffer than it pushed in before. So when it commits, the buffer is not yet empty. If the next read from this FIFO occurs in non-transactional code, it misses the data left in the framework's buffer. This violates the weak isolation among transactional and non-transactional code, which has been discussed in Section 3.1. In the presented example, transactional and non-transactional code cannot be interleaved reliably. A possible solution for this problem includes a new kernel interface for pushing back data into the FIFO's buffer in the case of an abort. The returned data could then be retrieved correctly by non-transactional code.

Other problems arises when using FIFOs to communicate between transactions in the same process. Assume two transactions communicating over a FIFO. Both transactions run in the same address space; so they are managed by the same transactional memory system. When the first read from the FIFO occurs, the reading transaction becomes irrevocable. It is then blocked by the read operation until some data arrives. This does not happen in the case that the sending transaction has not yet started, as the sender has to wait for the blocked receiver to commit. The occurring deadlock is again a violation of the specified rules for isolation.

A similar situation occurs if there is a cyclic dependency among transactions in the same address space. Each transaction reads from one of the FIFOs, but only one of them can become irrevocable. The other transactions have to abort. This again results in a deadlock as the irrevocable transaction waits forever for its communication partner.

Handling. Despite all these problems, the framework provides FIFO support for transactions. It defers all writes until the commit happens, and makes a transaction irrevocable when it attempts to read from a FIFO. The problematic case that a FIFO is used for communication within the same address space is very uncommon in real-world applications, as there are better and faster alternatives to FIFOs, such as process-local message queues.

Because FIFOs are not seekable, it is only possible to read and write at the beginning, respectively end, of the data buffer. Concurrency control on the FIFOs content is therefore not necessary, respectively possible. As for writing to regular files, the three strategies for concurrency control are provided. The optimistic strategy TS detects conflicts based on the timestamp of the FIFO's file offset. However, with the current implementation, it is not necessary to have any timestamp at all. All reads are executed irrevocably, and all writes are deferred into the transaction's commit. Thus, there are neither any overlaps with other transactions; nor any internal variables, such as an offset, to maintain. The pessimistic strategy 2PL detects conflicts by locking the FIFO for a transaction. The strategy NOUNDO makes a transaction irrevocable before accessing the FIFO. Table 5.1 summarizes this.

Sockets

Connection-oriented sockets are bi-directional communication channels between processes in a computer network. Data written to one end of the communication channel is transmitted to the other end of the channel and can be read there.

Sockets have similar properties as FIFOs. Like FIFOs, sockets are not seekable. Any reads and writes occur at the beginning, respectively the end, of the contained data.

Problems. Sockets also have the same problems as FIFOs: non-compensatable reads and cyclic dependencies among peers. The problem of cyclic dependencies among transactions in the same address space is more severe with sockets than with FIFOs, as socket communication within the same address space is common in today's software. For example, modern computer games often have a client-server architecture to ease the implementation of multi-player games. The server hereby provides the central management of the game and coordinates the actions of all players. Each player's client handles input, output, and communication with the server. In the case that client and server run on the same host they often share the same address space to save memory, but still communicate over a socket connection. This leads to the deadlocks described in Section 5.2.3 if server and client use transactions while communicating. A preliminary example of such an application is the game Atomic Quake [ZGU+09], which uses transactional memory in the server part.

Handling. Sockets and FIFOs share the same properties and problems. Hence, they are treated in exactly the same way. The framework does not provide connection-less sockets, but the discussion applies to them as well.

Devices and other file types

Other file types include device files for hardware access, shared memory objects, and possibly others. Most of these types cannot be used safely from within transactions. For example, when accessing a device file, the underlying hardware has to be prepared that any executed action might be compensated or deferred. None of today's consumer hardware devices provides such a facility. So if one of these file types is being used from within transactional code, the framework switches the respective transaction to irrevocability.

It is questionable whether any of these file types should be used from within transactions at all. By making a transaction irrevocable, the transactional memory system can provide serialized access, but it is highly context dependent whether this makes sense. For example, irrevocability might work well when interacting with the user's terminal. On the other hand, having several transactions writing to a sound card's device file is unlikely to produce meaningful output from this device.

5.2.4 The framework's model of file-descriptor I/O

The file-descriptor I/O component provides transactional semantics for file access. The design is based on how Unix internally handles file descriptor I/O and allows for the fine-grained abstraction of the related resources.

The file-descriptor I/O model is illustrated in Figure 5.2. Each individual file descriptor, open file description, and buffer is represented by a framework-internal domain. Two types of domains are provided: one domain type for representing a file descriptor, and one domain type for representing an open file description and its buffer. Each domain hold the process-global and transaction-local state of its resource. For each there is exactly one global state, and one state per transaction. To allow for weak isolation, the validity of all domain's data strucures ends with the commit of the last transaction in a set of concurrent transactions.

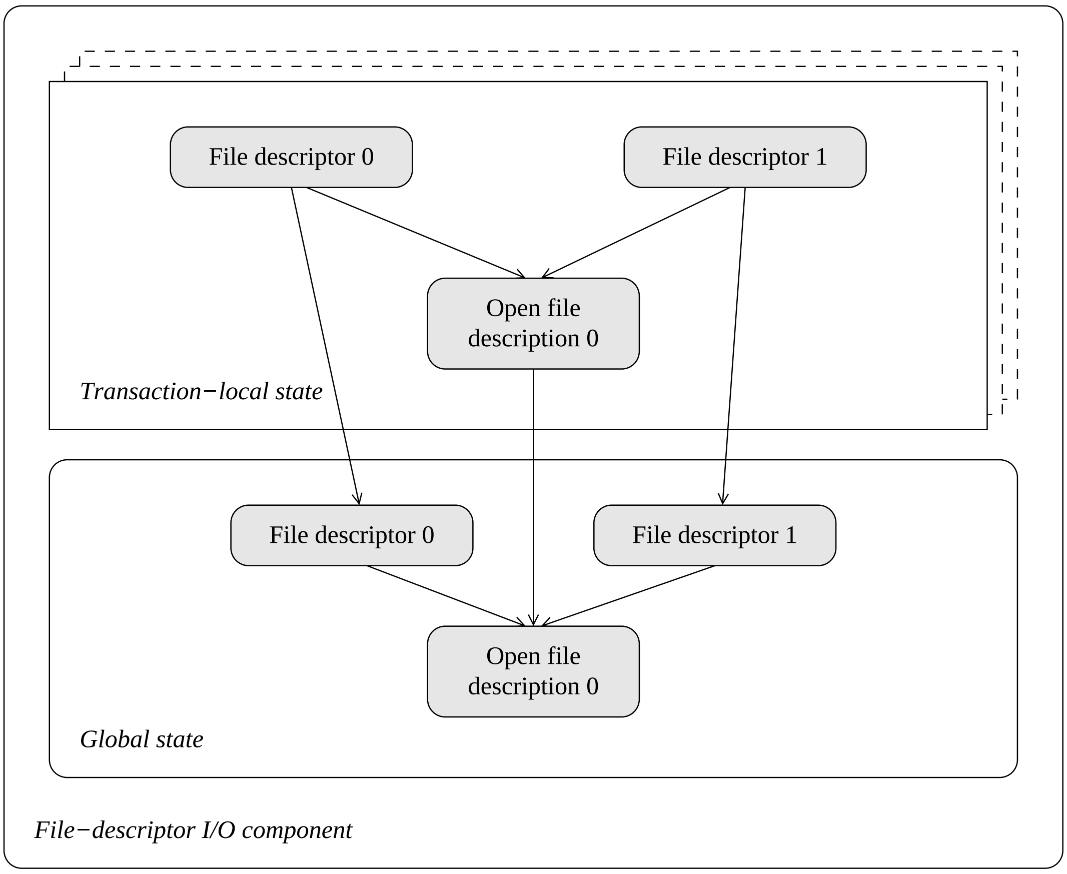


Figure 5.2: The connection among local and global file domains. The file domains are file descriptors and open file descriptions. File buffers are implicitly contained in open file descriptions. Each file-descriptor domain is connected to an open-file-description domain, either locally or globally. Each local domain is also connected to its global counterpart.

Buffers and open file descriptions

The kernel part of the I/O stack are the file buffers and open file descriptions. To reliably handled them within the framework, it is necessary to distinguish among individual instances of those resources. File buffers can be distinguished by their device ID and inode ID. The *device ID* represents the file system or communication protocol; the *inode ID* represents a device-internal identifier.

It is not possible to distinguish among open file descriptions. For this reasons an open file description and its underlying file buffer are treated as the same resource, and both use the file-buffer ID as their unique identifier. This leads to problems when transactional code uses two distinct open file descriptions that share the same file buffer. Assume a file is opened twice from within non-transactional code. Two distinct open file descriptions are created, which both share the same file buffer. When using the first open file description in a transaction, its domain is inserted into a framework-internal open-file-description table with the unique ID constructed from its file buffer's device ID and inode ID. When the second open file description is used in the transaction, it has the same unique ID as the first open file description, as it uses the same file buffer. Thus, its entry seems already present in the open-file-description table. The transaction now uses this wrong domain when it executes an action on the second file descriptor. This can cause problems when tracking the state of the open file description or its synchronization data structures. For example, if an executed action changes the file offset of the second open file description, the update would get applied to the first open file description.

At the moment, the only reliable way to handle this problem is to not allow the same file to be opened twice. The I/O component checks this case and aborts the transaction if a newly opened file descriptor's open file description is already present in the transaction's open-file-description table. What happens in non-transactional code is out of the component's scope. It is the application developer's responsibility to ensure that no two open file descriptions refer to the same file buffer.

A simple workaround for the problem is to check whether transactions already use an open file description via a different file descriptor. In this case the transaction could be switched to serial mode, which can avoid the internal state tables completely.

For each application, the kernel maintains an internal table that maps file descriptors to open file descriptions. A real solution to the problem is to add a unique ID for each open file description and make it available to the application. The ID could be returned by a call to `fcntl`, or by a special kernel interface for retrieving open-file-description IDs. The kernel could provide a system call to make this information available to the application or export it via the proc file system.

When using an open file description, and by extension a file buffer, within a transaction there is a global and a local state associated with its domain. The global state consists of

- a unique ID number,
- data structures for synchronization,
- a reference counter,
- the file type, and
- a set of properties for the type.

As described above, the *ID number* is used internally to distinguish among different open file descriptions. This is necessary for adding domains that represent new open file descriptions, and connecting the domains of file descriptors and open file descriptions.

The *synchronization data structures* are used for concurrency control on the open file description. This can be a lock, a timestamp, or something similar. The protection of the file buffer's content is described in detail in Section 5.2.5.

The *reference counter* holds the number of transactions using the open file description. It is increased whenever a transaction first uses the open file description, and decreased when the transaction commits or aborts.

The *file type* holds information whether the open file description refers to a regular file, a FIFO, or some other type. The *properties* are specific to the type of the open file description's buffer. For example, if the buffer is a regular file, the global state contains the file offset.

An open-file-descriptor domain's transaction-local state consists of

- an indication whether the open file description is valid,
- some properties specific to the type of the underlying buffer, and
- the transaction's local read and write sets for the buffer.

The *indicator* tells of whether the transaction holds a consistent reference on the open file description's global state. For example, if the global state uses timestamps for concurrency control, the indicator is the time when the reference on the file-descriptor domain was acquired.

The *properties* are again specific to the type of open file description's buffer. The *read and local write sets* contain the parameters of operations that have been executed on the open file descriptor, respective its buffer. This include actual write operations on the data, updates to the file offset, and read regions of the buffer.

File descriptors

The application's handle to a file is the file descriptor. It is the user-space end of the I/O stack. The global state of a file descriptor's domain consists of

- synchronization data structures,
- the ID of the associated global open-file-description domain,
- a reference counter, and
- a state flag.

Similar to the open file description, the *synchronization data structures* are used for detecting conflicts on the file descriptor.

The open file description's *ID* is used to connect the domains of the file descriptor to its open file description's domain. This is necessary as programs only operate on file descriptors, whereas most of the actual work is done in relation to an open file description.

The *reference counter* holds the number of transactions that use the file descriptor at any given point in time. When a transaction first uses a file descriptor it obtains a reference, and when it commits or aborts it releases the reference.

The *state flag* indicates whether the file descriptor

- is in use by some transaction (INUSE),
- is not in use (UNUSED), or
- has been closed by any transaction (CLOSING).

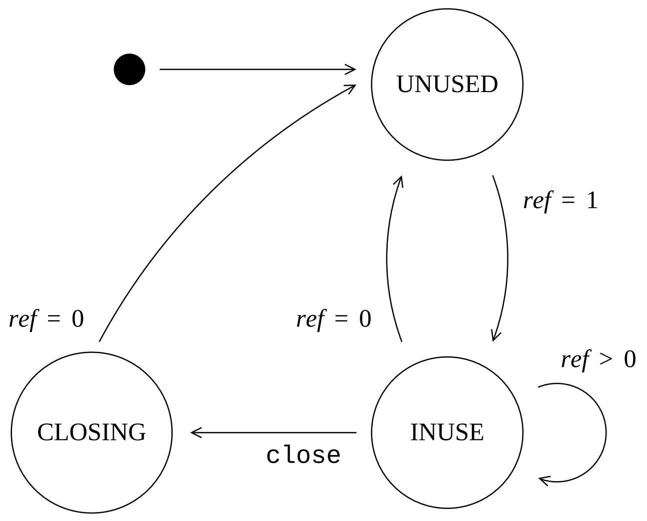


Figure 5.3: The state diagram for file-descriptor domains. Initially, each file-descriptor domain is in the state UNUSED. When it is first referenced it switches to the state INUSE, and back to UNUSED after the last reference has been released. When a close operation is committed, the state is immediately switched to CLOSING. This makes all conflicting transactions release their references and abort. After the last reference has been released, the file descriptor is actually closed and its domain's state is set back to UNUSED.

Together, the reference counter and the state flag prevent the file descriptor from disappearing behind the back of a transaction, as illustrated in Figure 5.3. By default a file descriptor is not referenced by any transaction and therefore its domain is in the state UNUSED. When the first transaction references the file descriptor's domain, its state is switched to INUSE. Every further reference keeps the domain in this state until all references have been released. When the last reference is released the domain is switched back to UNUSED.

A special case occurs when a transaction applies a close operation on a file descriptor. Then the file-descriptor domain's state is immediately switched to CLOSING. This signals all other transactions that refer to the domain to abort and release their reference. Also, no new references can be taken of a file-descriptor domain that is in the state CLOSING. When the last reference is released, the domain switches to UNUSED and the file descriptor is finally closed. Meanwhile the committing transaction can proceed. This protocol allows to reliably open and close file descriptors that are in use by several transactions at the same time.

Similar to open file descriptions, a file-descriptor domain's local state consists of

- the ID of the associated local open-file-descriptor state, and
- an indicator of the validity of the reference on the global file-descriptor state.

5.2.5 Data structures for serializing file-buffer access

Together with concurrency control on a file's data structures, concurrency control on the file's content is provided. This is implemented by dividing each file buffer into *records*: non-overlapping, equally-sized blocks of raw data.

Each individual record of a file buffer has an associated data structure that allows concurrency control for this record, such as a lock. The actual data that is protected by the record data structure, is stored in the file buffer or the transaction's read and write sets.

Organization of these record data structures imposes trade-offs to the framework's implementation. These are

- scalability,
- access time, and
- memory consumption of the data structure.

Scalability refers to the overall performance of the data structure when the file buffer is accessed concurrently. Retrieving a record should ideally not interfere with concurrent transactions working on the same file buffer.

Access time refers to the time a single, non-concurrent access takes. When a location in the file buffer is referred to, the average time to access its record's state should be minimal, in terms of algorithmic complexity and absolute number of processor cycles.

Finally, *memory consumption* refers to the amount of main memory needed to hold the record data structures. To fully cover large files, a lot of records might be necessary. Thus, records of unused portions of the file should ideally not be allocated at all.

Array-based implementation

For a naive approach, all concurrency-control data structures of a file buffer can be contained in an array. For each file buffer there is a global array with global locks or timestamps, and for each transaction using the buffer there is a local array with the transaction's local lock states or timestamps. Scalability here refers to the number of concurrent threads that can work on the array, access time refers to the amount of time necessary to retrieve a single cell in the array, and memory consumption refers to the amount of memory it takes to store the global and transaction-local array state.

The scalability and access time for obtaining a record from an array is very good. As individual cells of an array can be accesses concurrently and in constant time, this approach can scale and perform quite well.

One disadvantage of an array is the need to increase its number of cells. The reallocation might move the array to another location in main memory. So it is possible to either use or to reallocate an array, but not both at the same time. This might hinder scalability.

Another disadvantage is the array's dense nature, which results in a large memory consumption for this approach. Assume a file buffer of 1 GiB and a record size of 32 byte. This results in 33,554,432 records. For each of these records there is a global and a transaction-local state variable. Further assume that on average each state variable is 4 byte in size. For a timestamp this might even be to small. The global and local data structures for protecting access to the file buffer would allocate 256 MiB, which is one forth of the file buffer's size. With each additional concurrent transaction, the amount of memory would increase by 128 MiB.

Tree-based implementation

To avoid these problems the framework organizes each file buffer's records in a set of n-ary trees. There is one global tree for each file buffer, and another local tree for each transaction using the buffer. Each node in the tree points to a child node or a leaf of the tree.

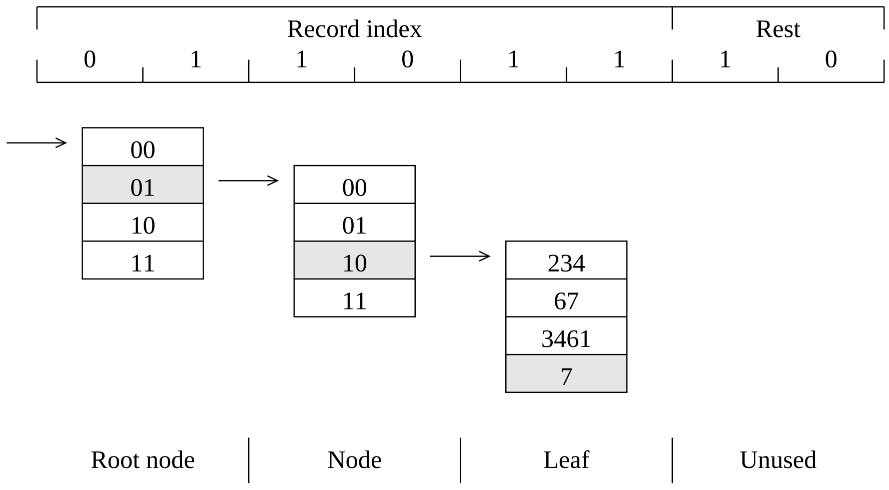


Figure 5.4: Searching in a record tree. The record data structure for the file-buffer offset 0110 1110 is retrieved from the tree. Each record covers 4 bytes of the file buffer. The record index is first divided into smaller chunks. In this example, there are three chunks where each chunk is 2 bit wide. The first chunk 01 is used as index into the root node's child table, the second chunk 10 is used as index for the first child node's child table, and the third chunk is used as index into the leaf's record table. The result is the timestamp of the record; seven in this case. The trailing offset component describes an offset within the record and is not used. For searches of consecutive records the found leaf already holds the respective data structures in many cases.

A tree is build lazily during its traversal. It can happen that the tree spans a smaller range than necessary if no access beyond some offset has taken place. In this case the tree is enlarged to support a larger file size by adding one or more new root nodes. The old root node already spans a range at the file buffer's beginning; so it is used as the new root node's first child.

It can also happen that the tree spans a large enough range, but some child nodes or leafs are not yet in existence because no access in the specific area has taken place before. Such missing child nodes and leafs are allocated and initialized during the traversal.

When looking up a file offset's record data structure from the tree, the offset is first right-shifted by the record size's number of bits to get the record index. The record index is divided into smaller chunks while walking down the tree to search for the leaf containing the record's data structure. Figure 5.4 illustrates a tree's organization and traversal.

When searching for a leaf in a file buffer's global tree, first the transaction's local tree for the buffer is traversed. This returns the local leaf for the specific record. On the first retrieval of the leaf, the global leaf of the record is retrieved from the file buffer's global tree as well. A pointer to the global leaf is stored in the local leaf. For any further searches of records in this leaf only the local traversal has to be done and the global leaf is retrieved automatically. This reduces search effort and contention on the global tree's node data structures.

Searching for a record has logarithmic complexity for trees, instead of constant complexity in the case of arrays. To minimize the time spend on traversal, each leaf can handle a large amount of records. For example, with the benchmark setup in Chapter 6 each leaf holds 512 consecutive records with each record handling an area of 32 bytes.

In most cases when checking a set of records during a read or write operation, few operations on the tree need to be performed. Once the respective leaf has been found for the first record it also holds the data structures of adjacent records.

To further improve access times, the implementation keeps all trees in place once they are established. Global trees are created once and are only destroyed when the process exits. Local trees are reused over transaction boundaries. So if a thread executes several transactions that use the same file buffers, the later transactions mostly reuse trees that have been established by the same thread's previous transactions. Some cleanup operations for each record's data structures might be necessary between successive transactions, but local trees are not destroyed until the thread quits.

The memory consumption of a tree depends on the transactions' usage of its underlying file buffer. If access is distributed equally within the file buffer, the memory consumption is similar to an array-based implementation. If transactions mostly work at the same offsets in the file, the record tree is almost empty and the memory consumption is low.

5.2.6 Actions

The current support of file-descriptor I/O provides the most often used I/O primitives: `accept`, `bind`, `close`, `connect`, `dup`, `dup2`, `fcntl`, `fsync`, `listen`, `lseek`, `open`, `pipe`, `pread`, `pwrite`, `read`, `recv`, `select`, `send`, `shutdown`, `socket`, `write`, and `sync`.

For some functions, such as `open`, it is debatable whether these are really file functions or rather file-system functions. The distribution of functions between this component and the file-system component of Section 5.3 might seem arbitrary. The coverage of the file I/O component is geared to meet the coverage of the file-stream interface in the ISO C standard. File streams only intent to handle file operations, but not file-system operations. The rule of thumb is that if there is an equivalent file-stream function, a function belongs to the file I/O component, otherwise it belongs to the file-system component. For future revisions, it might make sense to merge both components.

The first set of actions is related to the creation and destruction of file descriptors. The concept of their implementation is very similar to that of `malloc` and `free` of Section 5.1.

Open, dup, pipe, and socket. These calls create new file descriptors and possibly open file descriptions in the system. Their execution always proceeds in the same way. In the first step the primitive itself is executed. It returns one or two newly created file descriptors. In the second step the framework-internal data structures for the new resources are created and referenced. Section 5.2.4 discusses this in detail. Due to the inability of distinguishing among open file descriptions, which is also discussed in Section 5.2.4, it is currently not supported to open a file twice.

Calls to `open` respect the transaction local working directory. The execute method of `open` is implemented with `openat`, a new variant of `open` that allows to specify its current working directory. This way each transaction can open files in its own local working directory. This is explained in further detail in Section 5.3.2.

The apply methods of these actions do nothing. The undo methods simply close any created file descriptors. This is possible since no other transaction could have legally accessed them yet. If the file descriptor referred to a file that is owned exclusively by the transaction, the file is removed on undo. Exclusively owned files, such as temporary files and lock files, are detected by checking for the bit mask `O_CREAT|O_EXCL` in the `open` call's flags argument. Files that are not exclusively owned are not removed, which might leave files of aborted transaction in the file system.

The rationale of this decision is that non-exclusively owned files are possibly shared with other, external processes. The framework should not remove files behind the back of those processes. In this case that the file's name is also not chosen randomly, but follows some pattern, such that independent processes can use the correct filename. When the aborted transaction retries, it is likely to use the same filename as before. The possibility of leaving back orphaned files on abort surely is not optimal, but is still better than removing arbitrary files, which also opens the door to security breaches. If the transaction programmer wants to ensure that the transaction does not leave orphaned files in the file system, he or she should switch the transaction to irrevocability before executing the first call to `open`.

If `open` is supplied with the flag `O_TRUNC`, the file buffer has to be erased. Such an action can not be compensated easily. In this case the transaction becomes irrevocable.

Currently unsupported is the flag `O_APPEND` for opening files. This flag makes any write operation take effect at the end of the file buffer instead of the open file description's current offset. Support for `O_APPEND` should be implementable without major problems.

Close. Calls to `close` destroy file descriptors. The file descriptor's open file description is destroyed as well if it is not referred by any other file descriptor. The execute method of a `close` only inserts an entry into the log. The undo method does nothing. To prevent errors in other transactions, the apply method of a close operation does not close the file descriptor directly. Instead, it sets the framework's internal domain for this file descriptor to the state CLOSING and releases its reference. Any other transaction using this file descriptor will abort when it observes CLOSING and release its reference on the file descriptor as well. When the reference counter is zero, the file descriptor is finally closed and its domain's state is set to UNUSED. There is no need for the committing transaction to wait for this to happen. It can continue its commit after it released its own reference. The reference counting on file descriptors is also discussed in Section 5.2.4.

Dup2. Similar to `dup`, `dup2` duplicates file descriptors. It takes two arguments: the source and the target file descriptor.

The function `dup2` is problematic. It makes the target refer to the same open file description as the source. The problem with `dup2` is that it closes the target file descriptor if this already refers to another open file description. The close operation cannot be deferred because the duplication is a constructive operation, which has to be done within the transaction. It can be compensated by duplicating the old target file descriptor, but this strategy conflicts with any deferred write operations. During commit, these writes would use the target's new open file description, which violates consistency and weak isolation from other transactions using the same target file descriptor. For these reasons `dup2` is only provided for irrevocable transactions.

As a workaround it might be possible for the I/O component to defer the `dup2` operation and internally redirect all calls on the old file descriptor to the new file descriptor.

The second set of actions is related to reading and writing of file content. Their execution starts by first referencing the domains of the supplied file descriptor and open file description; then a primitive-specific action follows.

Lseek. This call changes an open file description's current file offset. It does this for revocable transactions by modifying the transaction's local file offset during its execution. When applied, the actual `lseek` call is executed; during undo nothing is done. The initial local offset is retrieved from the domain's global offset during the domain's local setup.

For revocable transactions, it is necessary to detect conflicts on the file offset as well. The concurrency-control strategy for the offset is the same as for the file buffer's records. If NOUNDO has been selected, an `lseek` call is executed immediately.

Pread, read, and recv. These actions read values from a file buffer. The function `pread` is offset-independent, but can only be used on seekable buffers, namely files. Its implementation depends on the selected strategy for concurrency-control, but the general behavior for revocable transactions is outlined here. Calls to `pread` are always executed immediately because the read data is needed within the transactions. In the case of a revocable transaction, the call might need to lock some file records or validate some record's timestamps. It also checks the transaction's local write set whether the transaction has previously written at the read offset, and copies the data from the write set if this is the case. Applying and undoing `pread` does nothing.

Calls to `read` behave similar to `pread`, but operate at the open file description's current file offset and increments its value while being executed. For seekable buffers, it is implemented like `pread` at the local offset with an additional call to `lseek`. For non-seekable buffers, such as FIFOs and sockets, `read` makes the transaction irrevocable.

The function `recv` is a special version of `read` for sockets. It takes an extra argument with socket-specific flags. In any case, `read`'s behavior is provided: the transaction becomes irrevocable.

Reading can block for an unspecified amount of time if data is not available. To prevent this from happening in revocable transactions, the execute method can wait for a certain amount of time and afterwards abort the transaction. This does not really solve the problem of the blocking read, but prevents starvation of unrelated transactions that want to access shared resources that have been locked by the blocked transaction.

Pwrite, write, and send. The function `pwrite` writes at a specific offset within a seekable file buffer. Its implementation depends on the selected concurrency-control strategy. In the case of an irrevocable transaction, calls to `pwrite` are applied immediately.

In the case of a revocable transaction, calls to `pwrite` are deferred into the apply method. Similar to `pread`, `pwrite` might need to acquire some records' locks. For deferred write operations the I/O component applies an optimization: If several successive write operations work at consecutive offsets in the file buffer, they are merged into one single, large write when being applied. This saves the overhead of all but one calls.

The function `write` is an offset-dependent variant of `pwrite` and moves the file offset while being executed. For revocable transactions, the concurrency control for the file offset is the same as for read operations. Calls to `write` operate on any file type. If it is executed on a non-seekable buffer, the action is deferred as for seekable buffers. In the case of an irrevocable transaction, the write is not deferred, but applied immediately.

The function `send` writes to a socket. Like `recv`, it requires an additional flags argument. If no flags are given, it behaves similar to `write` and is handled as such by the framework. If any flags are set, the transaction switches to irrevocability.

Writing can block if the output buffer fills up, like when writing to a FIFO without the other end's process reading. In contrast to reading, write operations are deferred into the commit phase, so the blocked transaction cannot be aborted easily. A possible workaround is to set the respective file descriptor to non-blocking mode during the commit. In that case an affected write operation does not block, but returns the error code `EAGAIN`. This triggers a call to the registered error-handler function, which might be able handle the error by signaling the reader process to deplete the buffer. A more reliable solution is to explicitly support this case within the kernel, which could provide arbitrarily large FIFO buffers.

Command	Purpose	Revocable
F_DUPFD	Duplicate file descriptor and clear O_CLOEXEC	Implemented like dup
F_DUPFD_CLOEXEC	Duplicate file descriptor and set O_CLOEXEC	Implemented like dup
F_GETFD	Get file-descriptor flags	Yes
F_SETFD	Set file-descriptor flags	No
F_GETFL	Get file-status flags	Yes
F_SETFL	Set file-status flags	No
F_GETOWN	Get owner process	Yes
F_SETOWN	Set owner process	No
F_GETLK	Get file-lock information	Yes
F_SETLK	Try to aquire file locks	No
F_SETLKW	Aquire file locks	No

Table 5.2: Overview of `fcntl` commands in the POSIX specification.

The third set of actions consists of functions that can be considered as special purpose and do not strictly fit into resource management or I/O.

Sync and fsync. These calls make the operating system flush its write buffers to the disk, either system wide or for specific file descriptors. `Sync` is executed twice, the first time during the transaction to flush everything that has been written until this point and a second time from within its apply method to flush everything written during the transaction's commit. It is not necessary to flush the transaction's write sets and become irrevocable before calling `sync`: the POSIX specification explicitly permits its implementation to do nothing. `Fsync` is inserted into the log and called from within its apply method.

Select. The function `select` waits for I/O on a set of file descriptors. It is executed immediately within the transaction. The `select` call's execute method references all file descriptors it has to wait upon to prevent their closing by other transactions. `Select` blocks the calling thread, but allows to set a timeout. If no timeout is given the framework's implementation set it to a default value. If no I/O has been detected when the timeout is reached the transaction aborts. This prevents deadlocks and stagnation with `select` calls that would otherwise block forever, and thus violate isolation among transactions. If a transaction is running irrevocably, the call behaves like its non-transactional counterpart. Especially, no automatic timeout is generated by the framework, as the transaction could not abort anyway.

Fcntl. A call to `fcntl` reads or writes the parameters of a file descriptor or open file description. Its transactional behavior depends on the given command parameter. As a general rule: If a read command is given then `fcntl` reads file parameters immediately; if a write command is given it switches to irrevocability. A complete overview of all commands is given in Table 5.2. The same synchronization data structures as for the file offset are used for concurrency control.

Accept, bind, connect, listen, shutdown. These calls are related to connection handling of sockets. The successful use of these calls depends on the communication peer, which is not under control of the transactional memory system. Hence, a transaction becomes irrevocable before executing one of these calls.

5.3 File System

In POSIX systems, the file system is a resource shared by all existing processes. Thus, the transactional memory system does not have complete control of what interaction with other processes is happening while file-system-related code is running.

5.3.1 Problems with file-system-related transactions

Because of the file system's shared nature it is very hard to give any useful guarantees for isolation among different processes. Without support from the operating-system kernel, it is often impossible to guarantee whether conflicts within the file system can be resolved or not. This section gives some examples of what problems to expect.

The first example contains several processes that try to create files with the same name. It is not possible to handle such conflicts from within the transactional memory system. This has to be coordinated by the file system or the transactions itself.

There are also no clear semantics for rolling back updates within the file system. Assume a transaction created a new file. Rolling back this action means removing the file from the file system. This is not safely possible because it is not sure whether the file still is the one created by the transaction or if another process has replaced it. A partial solution to this problem is to compare the inode number of the file to be removed with the inode number of the created file. However, this workaround does not necessarily work if the other process only modified the file, and contains a race condition between the retrieval of the file's ID and the file's removal. Within this time the file could be replaced or modified.

Another problem arises when undoing renames or removals of files. These operations cannot be deferred as the transaction might want to create a new file with the same name. When undoing the operation, a file with the old, removed name might have already been recreated by some other process in the system. The undo method can either restore the old file or keep the new file in place. It is not generally decidable of how to proceed in such a situation.

Finally a transaction might have created a new directory. It is not possible to remove this directory if it is not empty; like in the case that another process created any files there.

5.3.2 POSIX-compatible file-system interaction

The framework builds upon the assumption that for most file-system operations, no special handling is needed to use them from within transactions. This design relies on the observation that

1. it is not possible to give transactions the same isolation guarantees for file-system operations as for process-local operations, and
2. the file system already provides the semantics necessary for single concurrent and atomic operations.

The first point has been illustrated in Section 5.3.1. Whenever the transactional memory system tries to compensate a file-system operation, it might interfere badly with other processes, and the application has to be prepared for this case; just as in the non-transactional case.

Given the second point and the fact that the application needs to be prepared for file-system-related interference with other processes anyway, there is no need to provide guarantees for isolation in most cases. Thus, the framework exposes most of the file-system's actions directly and relies on the application to correctly handle any interference itself.

Two notable exceptions exist: The first exception is related to the transaction's current working directory, the other concerns exclusively owned files. The working directory is a process-local variable, which is fully under control of the transactional memory system. By handling access to this variable, the framework provides each running transaction with its own working directory.

Application programmers on POSIX systems have developed a set of common practices for safely interacting with the file system. One example is the atomic creation of new files. Instead of creating a new file directly and writing to it, the program creates an exclusively owned, temporary file; fills it with content; and renames it to the final name after all write operations have been done. Because the rename operation is atomic, there either is no final file or always exactly one consistent final file; even if multiple writers are present.

The framework makes this strategy available in a transaction-safe fashion. As described above, creating a new file typically starts by opening a temporary file and later renaming it to the final name. It is a reasonable assumption that such a temporary is only used by its creator process, as this is how it is supposed to be actually used. Deleting temporary files during the open's compensation action is therefore possible without interfering with other processes in the system.

The framework detects when an exclusively owned file is being opened: This either happens by a call to `mkstemp`, or if `open` is supplied with the parameters `O_CREAT|O_EXCL`. Any writes to such a file are executed according to the currently selected strategy for concurrency control. The final step of file creation is the rename. A call to `rename` makes the transaction irrevocable. If this succeeds, all pending writes to the temporary file are applied and the file is renamed to its final name. If that, or any previous step, fails; the transaction aborts. During the abort all pending writes are undone and the file is removed. This way, either a new file with consistent content is created, or the temporary file is removed. The discussion of `open` in Section 5.2.6 provides additional details on this behavior.

Note that this does not imply that the final content of the file is created by the last committing transaction. An external process could change or replace the file at any given time. Section 7.2 contains some ideas of how to solve this problem.

5.3.3 Actions

Since most file-system functions are only wrappers around POSIX interfaces, only very few operations are of real interest for the framework's implementation.

Fchdir and chdir. A call to `fchdir` or `chdir` changes the transaction's local working directory. When first called, `fchdir`'s execute method saves the new local working directory as an internal domain. This is now the transaction's current working directory. Further calls to `fchdir` replace the variable's value with the appropriate new location. Within the apply method, the process' working directory is updated with the transaction's local value. The undo method does nothing. The function `chdir` is implemented with `fchdir`.

Mkstemp. The execute method of `mkstemp` creates a new temporary file. Temporary files can be removed safely during an abort. The exclusive nature of the file assures that there can be no legal interference with other processes in the system. See Section 5.3.2 for a more detailed discussion of the topic. The compensation action of `mkstemp` is `unlink`, which is called from within the undo method. `Mkstemp`'s apply method does nothing.

Chmod, fchmod, fstat, getcwd, link, lstat, mkdir, mkfifo, mknod, stat, and unlink. These operations are wrappers around the POSIX functions. The only difference is that they respect the transaction's current working directory.

This is implemented with new system functions, which were added in the 2008 release of the POSIX specification. Thy are variants of the common file-system functions that take an additional first argument specifying the function's working directory.

Rename. Rename atomically changes a file's name and thereby replaces any previous file with the same name. The action makes the calling transaction irrevocable. This forces the apply of the transaction's write operations on the file. If `rename` finishes successfully, a new consistent file has been created. See Section 5.3.2 for details.

5.4 Other Components

In addition to the presented components, the framework's implementation provides a set of various other actions that are handled completely by the transactional memory system.

5.4.1 Errno

A special resource is the variable `errno`: a thread-local value that holds the error code of the last failed POSIX function. As `errno` is very common and used by almost all functions, it is handled by the transactional memory system. At the beginning of a transaction, the transactional memory system saves the thread's current `errno` value. Within the transaction, the thread-local value possibly gets updated by POSIX functions. During an abort, these updates are compensated by restoring the previously saved value. This makes it possible to directly wrap most simple functions, instead of providing an intermediate entry point just for saving and restoring the value of `errno`.

5.4.2 Math and arithmetic functions

The C standard library's math facilities include a set of basic mathematical functions. These consist of some predefined constants; trigonometric functions, such as `sin` and `cos`; exponential and logarithmic functions, such as `exp` and `log`; hyperbolic functions, such as `sinh`; and others. All of these function are protected actions, and as such are handled by the transactional memory system.

The math component also contains some weak pseudo-random-number generators. The framework provides the ISO random-number generator. It includes three functions, which are `srand`, `rand`, and `rand_r`. The function `srand` sets the initial seed value and `rand` generates the next pseudo-random number. Both functions depend on a global state variable. The third function, `rand_r`, is a thread-safe combination of `rand` and `srand`. The framework provides each transaction with a local copy of the state variable and uses `rand_r` to emulate calls to `rand` and `srand`.

Another facility is the rounding-mode setup for floating-point computations. This is not a problem with transactions because the setup is thread-local and works on processor registers only. The framework relies on TinySTM to govern these registers.

5.4.3 String and memory functions

The string and memory functions copy, compare, and modify strings and areas in main memory. All of these functions are protected actions that receive pointer arguments. To handle them, the frameworks announces the pointer's targets to the transactional memory system so that the functions can be used safely.

5.4.4 Process, thread, and system functions

Functions for process handling include `fork`, `exec`, or `wait`. Most of these functions alter some system-wide state and can neither be deferred nor compensated. The transaction has to become irrevocable before executing one of them. An exception are functions that return some constant value; such as `getpid`, which returns the process ID. Those can be called without special handling.

The same is true for thread-related functions, such as `pthread_create` and `pthread_join`. Most of these functions force irrevocability on the transaction. Transactional support for locking data structures can be provided when correctly integrated with the transactional memory system [VGS08].

6 Evaluation

This chapter evaluates the presented framework's design and implementation regarding the specified requirements. The first sections talk about correctness, standards compliance and extensibility; the final section focuses on performance.

6.1 Correctness

This sections presents an argument on the correctness of the transactional execution and afterwards describes the test cases for the framework's implementation.

6.1.1 Argument

Correctness has been defined in terms of atomicity, consistency and isolation. The criteria for isolation of concurrent transaction is conflict serializability. The execution of a set of concurrent transactions T_i is modeled by a history *H:* $H = \{T_1, T_2, ..., T_n\}$. H is serializable if it is conflict equivalent to a serial history $H^{'}$ that contains the same transactions.

The framework constructs H by atomically committing consistent T_i. To show that the framework provides serializability for its transactions it is necessary to show the equivalence of all histories H to a serial history $H^{'}$. Therefore, first the serializability of single T_i is examined and afterwards the serializability of the union *H.*

Each T_i describes one single, sequential execution. Conflicting operations in T_i are ordered by the relation $<_\mathrm{i}$. T_i is therefore a conflict-serial history by itself.

To show the correctness of the union H of transactions it is necessary to show how conflict equivalence is ensured, such that all T_i are ordered according to their observed conflicts, and how the commit is implemented atomically, so that two transactions do not interfere during commit.

The trivial case is an irrevocable transaction. The framework's implementation makes such a transaction run exclusively and not overlap with other transactions, which is exactly the definition of *serial.*

For the non-trivial case of revocable transactions, conflict equivalence is ensured by keeping each domain in a consistent state: a state that can be observed after a commit in a serial execution. Two cases have to be examined: the consistency of single, individual domains and the consistency of the composition of all domains used by a transaction.

For individual domains, each domain isolates its resource from the rest of the system. It provides the transaction with a local state of the resource: the domain's global state when the transaction first accessed the resource plus the local updates. A domain also provides some form of concurrency control, like two-phase locking or timestamp validation, to ensure the consistency of the local state.

The actual validation is specific to the domain's implementation. With pessimistic concurrency control, the consistency of the local state is guaranteed once it has been validated. With optimistic domains, the consistency is only guaranteed up to the point of validation. To validate a read operation, the value is first retrieved and afterwards the consistency of the domain is validated. Because the validation happened after the retrieval, the retrieved value is consistent if the validation succeeded.

The composition of a set of domains is consistent if all its domains are consistent. By validating all domains individually the validity of the composition can be determined.[1] For pessimistic domains, this again happens by definition; for optimistic domains the validation is performed after the retrieval of a (every) data value.

To ensure the commit of consistent transactions only, there has to be an atomic step of validating the composition of a transaction's domains and applying the transaction's actions.

For domains with pessimistic concurrency control this is again ensured by definition. All once validated data items of such a domain are guaranteed to stay valid until the end of the transaction, which is the end of the commit. For domains with optimistic concurrency control the framework implements a two-phase locking protocol for locking each domain's affected data items during the commit. This allows optimistic domains to perform an atomic step of validation and update.

The commit of a transaction T_i

1. locks all optimistic domains' data items that are used by T_i,
2. validates their consistency,
3. updates all data items, and
4. unlocks all domains.

To summarize: the framework allows for the implementation of two-phase locking for all domains. Pessimistic domains have this semantics by default, optimistic domains can implement two-phase locking during the commit. After all optimistic domains have been locked their consistency is validated. If this succeeds the consistency until this point is guaranteed. In conjunction with two-phase locking, the consistency until the end of the transaction is guaranteed.

Two-phase locking ensures atomicity of the transaction, as it excludes concurrent transactions during the commit phase. The validation of the optimistic domains after the locks have been acquired ensures consistency: no inconsistent transactions can be committed. Two-phase locking also guarantees that the constructed history H is equivalent to a serial history $H^{'}$. A formal proof of the correctness of two-phase locking can be found in [BHG87].

For the currently implemented components, the only exception is the support for file-system operations. The file system is shared among all processes in the system. For a transactional memory system in user mode, it is not possible to guarantee isolation among them without having support from the kernel. It is at most possible to isolate file-system operations of individual transactions in the local process. The operations of external processes cannot be influenced, thus a transaction still has to handle them correctly.

Correctness cannot be guaranteed in the presence of failures or certain use cases. If an error happens during the commit of a transaction, it is possible that the transaction is only committed partially, thus violating atomicity and exposing an inconsistent state. Section 7.3 presents some ideas on how to improve commit-time error handling. Section 5.2.3 discusses several use cases where inter-process communication leads to deadlocks, and thus violating isolation among transactions. A problem for isolation with non-transactional code is the inability to reliably distinguish among open file descriptions, which is described in Section 5.2.4.

[1]Very informal argument: According to Sections 3.2.1 and 3.2.2, each domain represents a set of page-level data items; each action can be expressed as a series of page-level reads and writes. The composition of a set of domains represents the composition of the domains' page-level data items. Thus, the consistency of a set of domains can be validated by checking for the non-existence of conflicting page-level operations. The respective algorithms are well-known [WV01].

6.1.2 Tests

To test for correctness, a set of test cases has been implemented during the framework's development. The correct results of these tests indicate the soundness of the design and the absence of major bugs in the implementation.

The tests of memory allocation include allocating and freeing memory from within a transaction, freeing memory in a transaction that has been allocated outside of the transaction, and allocating memory in a transaction that is freed outside of the transaction.

The file-descriptor I/O tests include concurrent I/O on one or more file buffers where the files are opened and closed inside or outside of the transactions; piping byte streams from non-transactional code into transactions and vice versa; and copying content from one file descriptor to another file descriptor, like reading from a file and writing to a pipe or the terminal.

For the file system the correct handling of the local working directory was tested.

The tests of error-handling checked for the correct call of registered error-handler functions during the commit.

6.2 Standards Compliance

Standards compliance describes the framework's requirement of making POSIX interfaces available to transaction programmers without changing the interfaces' semantics. This section first discusses how this is achieved and afterwards summarizes the exceptions.

The POSIX specification lists 1191 interfaces, of which 232 are currently supported by the framework. This gives a coverage of approximately 20 percent. The supported functions are mostly related to memory management, file-descriptor I/O, file-system interaction, string and memory handling, and math.

The case of transactions is completely handled by the framework. If a transaction is running irrevocably, the non-transactional implementations are used directly, so the semantics are the same. The only exception is memory allocation, due to its interference with transactional memory operations. Allocation is performed immediately. Freeing memory still needs to be deferred. This is not a problem: According to the POSIX standard, using heap memory that has been supplied to a call of `free` results in undefined behavior. It is therefore compliant to keep the memory allocated until the end of the transaction.

The other case are revocable transactions. For these to work, all examined interfaces where classified into one of the three categories for external actions: protected, unprotected, and real actions. Protected actions execute on memory only. They are supported by wrappers around the actual non-transactional implementation and handled by the transactional memory system. Real actions, which can neither be deferred nor compensated, render the transaction irrevocable. While this happens any previously deferred actions are applied.

The interesting case is the use of unprotected actions, which are either deferred or compensated. By definition, all compensated actions are executed immediately. The support of deferred actions is provided by the domain concept. As each domain provides information hiding for its associated resource, so the results of deferred actions can be simulated to give transactional code the impression that the action took effect.

Such dependencies among deferred or compensated actions are represented by the transaction's local history, which contains events of actions that have been executed within the transaction. Each event in the history refers to exactly one invoked action. The events are ordered by the conflicts among their actions. This guarantees that a transaction's deferred actions are applied and compensated in the correct order.

The history spans over all domains that are used by a transaction, so it is possible to apply or compensate actions that depend on the state of more than one domain in the correct order. An example of this is a call to `sync`, which flushes the system's write buffers. Another example is a domain that represents a global function pointer. The pointer's function is called by several deferred actions for allocating memory. These actions depend on the pointer's value and some other unrelated domain that represents the actual resource they work on. If the function pointer's value is modified (by some setter action) during the transaction, any successive, deferred actions have to use its latter value. Such inter-domain/inter-action dependencies can be represented.

The the correct interaction of transactional and non-transactional code via weak isolation has to be assured by the application programmer. The provided components supports this by leaving all resources in a state that is equivalent to the state of a non-transactional, sequential execution after the last concurrent transaction has completed. Especially no framework-internal state is kept that might interfere with non-transactional code, such as internal write buffers or file offsets.

The issue of supporting common practices is hard to argue about, as these practices depend on the system, the programmer, and the local conventions. All these factors change over time. The reasoning here is that common practices result from formal compliance with the relevant standards. If a specific technique is possible with the non-concurrent implementation of an interface, it is possible with any implementation that exposes the same semantics.

The presented framework achieves standards compliance for most operations, but fails for at least one important interface and some use cases. The former concerns the inability to distinguish among open file descriptions if they share the same underlying buffer. This limitation prevents an application from opening a file multiple times and using it safely within transactional code. The problem is mostly a result of the shortcomings of the POSIX specification. Additional information about the open file description, such as a unique identifier, would solve this. The problem is discussed in Section 5.2.4.

The other problems relate to the inability to perform certain use cases. When doing inter-process communication via file buffers, FIFOs, or sockets; some scenarios are impossible to perform or result in deadlocks. These problems are caused by irrevocability of transactions, deferral of write operations, or some combination of both. See Section 5.2.3 for more information.

6.3 Extensibility

Extensibility describes the capability of connecting additional, unrelated components to the framework. The motivation for this requirement is to allow future extensions to be added without the need of rewriting the framework's core.

Extensibility is provided by the framework's component architecture. Each component encapsulates a set of domains and actions, and is responsible for maintaining their global and transaction-local state.

A transaction's local history only interacts with the components, but not with any domains or actions directly. Each component provides the history with the necessary interfaces; such as to validate domains, apply events, or undo them. As all action-specific information is kept inside the respective component, the history can hold events of arbitrary actions.

New components can be supported by registering the component's public interfaces with the history. When connecting a component, it is provided with functions for interacting with the rest of the framework, such as adding events to the transaction's history.

The current implementation already provides several distinct components. Additional components can be build by adapting the underlying design principles. The steps to implement a new component from non-transactional code are roughly to

1. identify non-transactional resources, such as data structures, which form new domains;
2. identify non-transactional functions, which form new actions;
3. identify consistency constraints for new domains;
4. implement new domains and actions; and
5. provide some component-internal infrastructure to connect domains and actions with the framework.

For making actions available there exists a code generator that creates most of the necessary wrapper code. Many cases of adding protected or real actions, which only execute on main memory or need irrevocability, can be handled completely by the code generator.

Besides the code generator, the implementation contains several data structures that can be reused. This includes locks, timestamps, and basic memory allocation. The local and global tree's of record data structures are build upon generic tree structures. These can be used with different tree-leaf implementations to provide new types of trees. Most of this code is currently located in the I/O component, but can be moved to a shared library easily.

The component's internals mainly consist of code and data structures for translating among resources, as seen by the transaction programmer, and domain data structures; and for translating among events and actions. The necessary information can be attached to the memory transaction via a pointer, or stored in thread-local storage.

Given that the architecture and feature set of POSIX systems do not change radically with future revisions, the requirement of extensibility has been achieved.

6.4 Performance

The framework's performance is strongly influenced by the implementation of its components. This section presents benchmark results for memory allocation and I/O on file descriptors. The other components mostly consist of protected or real (irrevocable) actions. Benchmarking these components would not give much information about the framework.

The tests measure the throughput t in transactions per second. A transaction's average response time r is $r = 1/t$. The framework does not provide freeness from starvation, so an upper bound on the response time can not be guaranteed.

All tests are micro-benchmarks. It is not yet possible to come up with some real applications or workloads because software transactional memory and external actions are still subject to intensive research, and not used in production systems. Several free database systems, such as MySQL or PostgreSQL, were examined for conversion to memory transactions with external actions. However, none was found to be easily convertible. The design of the database systems is not really compatible with the framework: Databases do their own internal transaction management, which cannot be replaced by the framework without restructuring large parts of the database application.

All of the performance measurements were done on a system with 4 AMD Opteron 8346 HE quad-core processors and 16 GiB of main memory. The system was running Fedora 10 in 64-bit mode. The benchmark application was a 32-bit binary.

6.4.1 Memory allocation

This section presents some measurements on the performance of transaction-safe memory allocation.

Methodology

The performance tests are micro-benchmarks that measure the number of transactions per second. Each test consists of a set of threads or transactions; each first allocating 32 bytes of heap memory using `malloc`, then writing its respective thread ID to it, and afterwards deallocating the memory using `free`. This is repeated 10, 100, and 1000 times per transaction. The memory allocator is the standard memory allocator of glibc 2.9, which comes with the Fedora distribution.

The small number of allocated bytes was chosen to minimize the influence of book-keeping and page mapping. Each test ran for 60 seconds to minimize the influence of other processes in the system. The number of bytes covered by TinySTM's internal locks was set to 4, which is the size of a machine word. Care was taken to ensure that no sharing of memory words among transactions happened.

In the transaction-safe setup, the `malloc-free` pair is implemented by the framework. The setup denoted `TinySTM` uses transactional memory, but is not build upon the framework. Instead, it uses special support of `malloc` and `free` that comes with TinySTM. In the non-transactional scenario each thread was executing the plain system functions.

Results

Figure 6.1 shows the throughput with transactional and non-transactional code. In the non-transactional case, the throughput roughly scales linearly with the number of concurrent threads. The minor fluctuations in the curve's slope, can be explained with the non-deterministic performance of computer systems. Both transactional cases, specially depicted in Figure 6.2, do not scale well. For a small number of threads, roughly up to seven, the throughput climbs linearly with a slope of one. For higher numbers of threads the throughput either stays at this level or falls back to lower levels.

When comparing the curves of the transactional cases only, their shapes are similar, but TinySTM's implementation has a lower overhead than the framework. This comes of no surprise as it is a special purpose implementation for handling heap allocations. It does not maintain a complete, generic history for the transaction.

No conflicts were observed in any of these cases, but the overhead of the transaction is considerable. For example, in the case of 10 iterations, the non-transactional, single-threaded case has a throughput of 799,419 calls per second, the framework only achieves 167,602 transactions per second, which is a ratio of 5:1. With 253,429 transactions per second, the implementation that comes with TinySTM performs better, roughly at a ratio of 3:1.

6.4.2 File-descriptor I/O

Some performance measurements for I/O on regular files are presented in this section. Each measurement is a micro-benchmark that simulates a specific use case. For legibility all figures with benchmarks have been moved to the end of the section. The number of aborts for each test is listed in Table 6.3.

Methodology

Each test consists of a set of threads, all doing `pread` and `pwrite` operations on the same file buffer. Each operation moves 24 bytes from a location in the file buffer to an application buffer or vice versa. All application buffers are transaction-local. The buffer size of 24 bytes is intended to reflect the size of an average I/O operation, such as accessing binary values or a single line in a file.

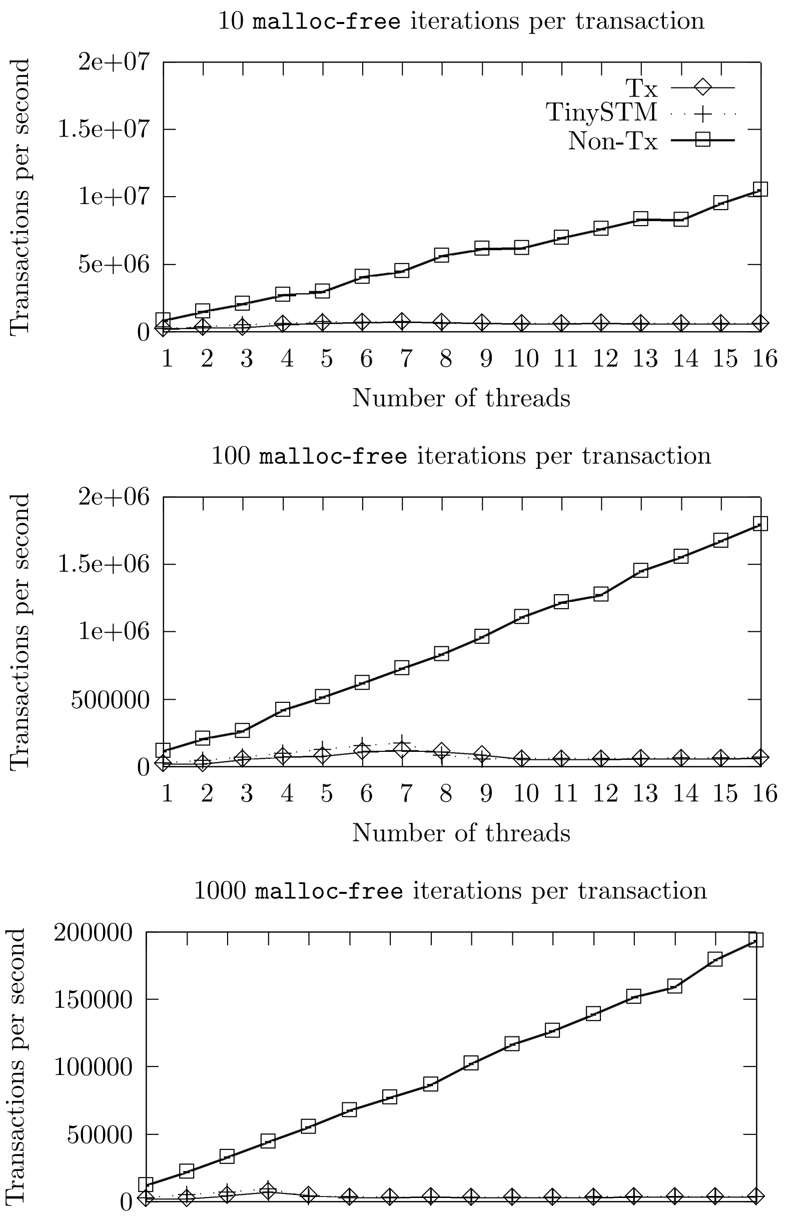


Figure 6.1: The throughput of `malloc` and `free`. Each transaction repeatedly allocated and freed memory on the heap. This facility is often used and can become necessary while converting an action's arguments to transaction-safe values.

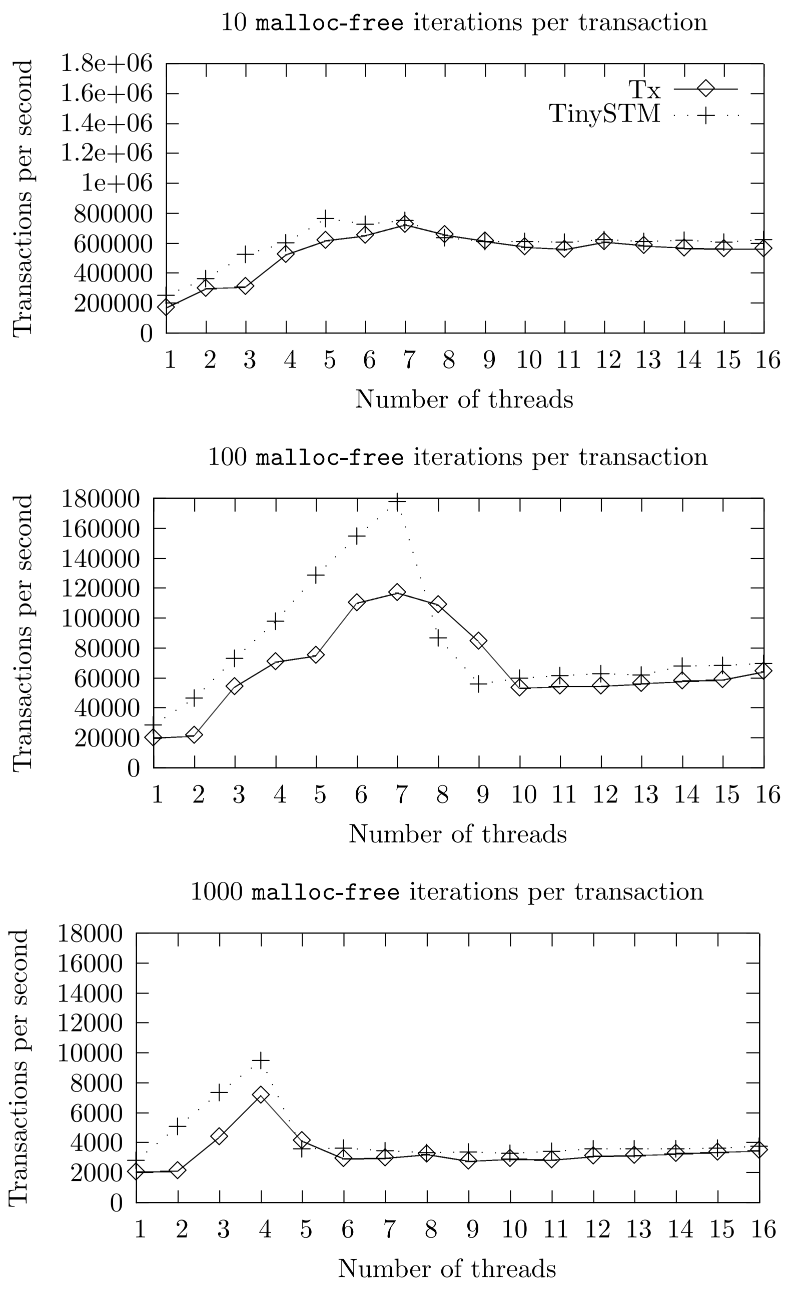


Figure 6.2: The throughput of transactional `malloc` and `free`. This is the same data as in Figure 6.1, but only the transactional case are shown.

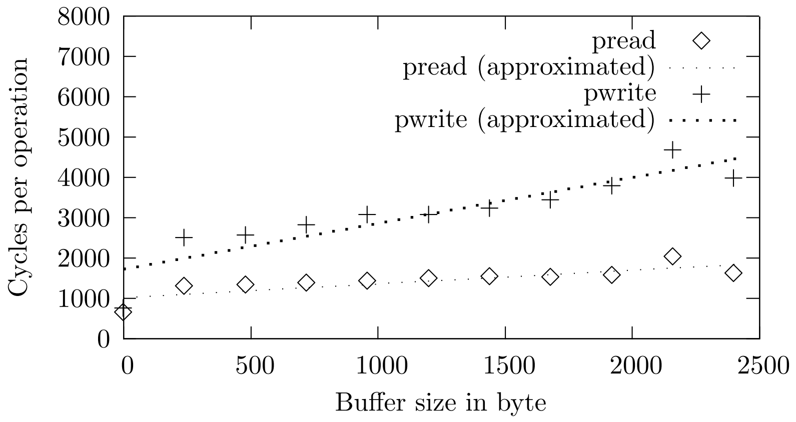


Figure 6.3: The necessary processor cycles per I/O operation. The discrete points depict the measured overhead of `pread` and `pwrite` for specific buffer lengths. The lines are approximations of these values.

Each offset is pseudo-randomly computed by a call to `rand_r`. The overhead of `rand_r` has been measured to be roughly two hundred fifty cycles. Each thread has its own seed value with an initial value greater or equal to 1, which represents the thread's logical ID.

In addition to the number of threads, it is possible to set the number of `pread-pwrite` iterations within a transaction. Increasing the number of iterations decreases the influence of each transaction's initial setup, but increases the influence of the transaction's history and the probability of conflicts among transactions.

Two different file sizes are considered. Access to very small (database) files is simulated by a file of 1 MiB size; access to larger files is simulated by a file of 100 MiB size. The size of the file might affect the conflict probability among transactions and the performance of the framework's internal data structures.

For concurrency control, each file buffer is divided into records of a fixed size. With many records of small size, the probability of conflicts among transactions is low; but the costs of record management, like memory consumption or tree-search overhead, is high. With larger record sizes the costs of management decrease, but the conflict probability increases. Several record sizes have been tested. For the presented benchmarks a size of 32 byte per record performed best on average. This results in 32,768 records for a file of 1 MiB size and 3,276,800 records for a file of 100 MiB size.

Four different strategies for concurrency control are tested. The strategies NOUNDO, 2PL, and TS are provideed by the framework. NOUNDO switches the transaction to irrevocability, 2PL implements pessimistic concurrency control by the use of two-phase locking, and TS implements optimistic concurrency control based on timestamps. The strategy CS does not use the transactional memory system, but serializes by the use of a global lock around the calls to `pread` and `pwrite`. This represents a coarse-grainly locked, non-transactional critical section.

Concurrent I/O on the same file buffer does not scale on current Linux systems. When two or more threads concurrently access the same file buffer, all but one block. This has been observed on different file systems, such as tmpfs and XFS.

For this reason, no actual system calls are performed. The framework has been modified to busy-wait[2] for an operation-specific amount of time, which simulates a file system that scales linearly with the number of processors. The number of cycles to perform an `pread` or `pwrite` operation depends on the size of the supplied buffer. To minimize the error in the benchmark, the simulated overhead is kept close to the overhead of an actual system call. Figure 6.3 shows the overhead of single `pread` and `pwrite` operations for various buffer sizes on a tmpfs file system. The results are approximated using the method of least squares to simulate the system-call overhead. In the special case of a buffer size of zero, the actually measured value is used.

Not performing actual system calls is problematic in terms of realism of the tests, but allows to make conclusions about the performance of the presented framework. Otherwise, all measurements were hampered by the file system's shortcomings. A solution to this problem is the implementation of a new file system that allows concurrent I/O on the same file buffer.

Transaction phases

Each transaction conists of several *phases*, which are performed during the transaction's execution. The first benchmark examines overhead within these transaction phases. Table 6.1 shows a detailed measurement of various transaction phases for the single-threaded case of `pread` and `pwrite` operations. The ratio among them is 1:1. For this test the framework uses the actual system calls.

The columns *Strategy* and *Iterations* give the transaction's concurrency-control strategy and the number of iterations. The framework's provided strategies are tested, either for `1`, `10`, or `100` iterations per transaction.

The column *complete* gives the average of processor cycles for a complete transaction. The strategies NOUNDO or 2PL show the smallest overhead. The TS strategy performs worst because of its high amount of bookkeeping, which is explained in later columns.

The column *pread* shows the average duration of `pread`'s execute function, ranging from two thousand to twenty two thousand cycles. For all strategies, `pread` takes a significant amount of cycles. In the case of NOUNDO this is the result of switching to irrevocability. The switch imposes an overhead on the operation: validation, commit, and acquisition of a reader-writer lock. For the concurrent cases the overhead comes from an initial setup of the framework's internal data structures. This at least involves some checking whether these data structures are already allocated or not. Later invocations of `pread`'s execute method do not have these overheads, so the average duration of the call decreases with the amount of iterations. Another observation is that the execute method has a significantly higher overhead for the TS strategy than for any other strategy. This is a result of the additional validation of the file buffer's records, which has to be done to prevent the read of inconsistent data.

The average duration of `pwrite`'s execute method is shown in the column named *pwrite*. In the case of the NOUNDO strategy, the write is done immediately. Thus, NOUNDO has a significantly higher overhead than the other strategies: between three thousand and sixteen thousand two hundred cycles for NOUNDO, but only a few hundred cycles for the others. In the latter cases, only internal data structures are updated, but the actual write operation is deferred.

The *lock* and *unlock* phases, shown in the respectively named columns, are necessary to guarantee atomic write operations for the optimistic TS strategy. All records that were accessed from within a transaction are locked at the beginning of the commit and unlocked at its end. This imposes a significant overhead for long transactions, such as around ninety thousand cycles in the case of 100 iterations.

[2]Each thread runs on its own core, so busy waiting should not have an influence among threads.

Transaction		Phase								
Strategy	Iterations	complete	`pread`	`pwrite`	lock	unlock	validate	apply	updatecc	finish
	1	40,274	8,107	16,190	702	394	492	82	207	444
NOUNDO	10	99,920	3,073	4,855	746	397	490	81	207	453
	100	676,205	2,050	3,544	750	403	525	81	207	433
	1	13,814	6,240	785	652	392	549	2,156	445	429
2PL	10	66,182	2,244	792	656	408	562	19,965	2,463	459
	100	655,028	2,643	714	685	452	582	197,592	23,459	478
	1	30,363	21,859	649	1,188	625	705	2,281	319	427
TS	10	109,628	5,593	634	7,784	2,602	2,010	21,348	1,241	440
	100	903,269	3,992	574	88,037	21,113	15,519	205,090	10,416	499

Table 6.1: The number of processor cycles of a transaction's various phases. Each test performs `pread` and `pwrite` operations with a ratio of 1:1 at random offsets in the file buffer. The buffer size is 1 MiB.

The other strategies do not need the locking and unlocking phases, and therefore consume much less cycles. The minor overhead shown here is a result of the implementation, which needs to call these functions even for non-optimistic domains.

The column denoted *validate* shows the duration of the transaction's final validation. This phase is again only necessary for the strategy TS; the other strategies use some form of locking for mutual exclusion, which makes validation unnecessary. As with locking any overhead in their cases is related to implementation details. For TS, the validation overhead increases linearly with the number of read operations. In the case of 1 iteration it is 705 cycles, of which roughly five hundred cycles can be attributes to the framework's core. In the case of 100 iterations validation accounts of 15,519 cycles.

The table's column *apply* shows the overhead of applying a transaction's updates. NOUNDO does not defer its writes, so in this case it is almost zero. The other strategies both execute the same algorithm: They walk through the transaction's local history and call the apply method of each event's action. The overhead increases almost linearly with the length of the history; from around two thousand two hundred cycles with 1 iteration to roughly two hundred thousand cycles with 100 iterations.

The *updatecc* phase, shown in the respectively named column, updates the framework's data structures after a transaction has applied all its actions. In the case of NOUNDO it does nothing. The small overhead is implementation related. For the TS strategy this phase increases the global timestamps of the written records; for the 2PL strategy it releases all acquired record locks. For both strategies, the overhead increases linearly with the number of write operations in the transaction. The TS strategy benefits from the locking and unlocking that is done during the lock and unlock phases. It can access the records' data structures normally and therefore performs much better than the 2PL strategy. With 2PL each record is instead released with the help of an atomic operation, which takes significantly more time.

The last column shows the overhead of the *finish* phase, which is executed before a transaction finally quits. The finish phase is common to all strategies and only involves some minor cleanups. With around four hundred fifty cycles it is almost negligible.

In general, the strategies NOUNDO and 2PL perform equal most of the time, whereas TS has a significant overhead. This can be attributed to the extra validation, locking, and unlocking during the commit; which is necessary for optimistic concurrency control.

Random reads

This benchmark, shown in Figures 6.4 and 6.5, tests the performance of read-only transactions that access random locations. It simulates use cases with extensive read operations, like searching a database for specific information. This and all successive benchmarks approximate system calls.

The strategy CS does not scale at all, as only one thread is running at any point in time. When going from the single-threaded case to multi-threading, there is a measurable drop in performance. This can be attributed to the increased overhead of maintaining multiple threads, such as thread switching and additional cache misses. For short transactions, the throughput drops from 763,071 to 203,971 transactions per second. The more iterations a test contains, the less impact the additional overhead has.

With the NOUNDO strategy the framework prevents the execution of all transactions, but the irrevocable one. Thus, the NOUNDO strategy scales equally bad as CS, and does in addition suffer from transaction-related overhead.

In the single-threaded case, the strategy 2PL performs between twenty five and sixty percent of CS. The transaction's overhead is relevant here. In the multi-threaded cases, the overhead is hidden by the concurrency.

This gives 2PL an up to two times higher performance for transaction of short and medium length. The throughput peaks with 13 threads, at 407,916 short-length transactions per second; respectively with 3 threads and 104,298 medium-length transactions per second. It stays roughly at these levels when the number of threads increases towards 16. For long transactions, the performance of 2PL is not that good. With 4 threads, its throughput peaks at 15,665 transactions per second, which is one and a half times better than the CS. Afterwards 2PL decreases to CS's level around ten thousand transactions. There are no conflicts if values are only read and no aborts happen. Still the throughput does not scale linearly with the number of threads. There is always some setup involved when starting a transaction. At least the transaction has to find the file's global data structures. The search involves some internal locking. The framework also maintains an internal lock for each record. Acquiring and releasing these locks affects other transaction, which use the same memory bus.

For short transactions, the optimistic strategy TS performs similar to 2PL. It reaches its peak at 435,999 transactions per second with 9 concurrent threads. It stays at this level when more threads are added. With the increase in transaction length, the performance of TS decreases significantly. For medium-length transactions, it performs below fifty thousand transactions per second; a level close to CS. For long transactions it performs around four thousand transactions per second, which is near NOUNDO. The decrease in throughput can be attributed to the increased amount of necessary validations. Also, during the commit of a transaction, the transaction's records are locked to guarantee the commit's atomicity. As described in Section 5.2.5 all records are stored in the leafs of a tree. Two adjacent records are likely to end up in the same leaf structure. The locking of records is implemented by locking the complete leaf structures. This reduces the time and memory overhead; but temporarily blocks concurrent, but unrelated, transactions that want to commit to a nearby record's location in the file buffer. Locking is not strictly necessary for read-only transactions. A future optimization could explicitly check for for this case and leave the locking out.

When performing these tests on a file of 100 MiB size, similar results are retrieved. The only difference is the performance of TS. For short and medium-length transactions, the throughput is miserable compared to the test on a small file. For long transactions the throughput is better than for small files. This again might be attributed to the internal handling of records. The framework tries to reuse internal data structures once they are initialized. For example, a record's timestamp is kept around even though the actual file might have been closed, and reused for the next file. Refer to Section 5.2 for a more detailed discussion. With larger file sizes there are more records and it is less likely that a transaction hits previously allocated data structures; thus the framework's overhead increases. The effect can be seen when comparing the single-threaded cases, where no distorting contention happens. For TS the throughput of long transactions per second is 4168 with small files, whereas it is only 822 with large files. For transactions of short and medium length it is less possible to reuse internal data structures. For long transactions, less reuse also happens, but results in less contention on component-internal locks than for short files. This in turn yields better performance.

Sequential reads

Figures 6.6 and 6.7 show the performance of sequential reads. This benchmark simulates use cases, such as paging in parts of a database, where a transaction reads large amounts of consecutive data from a file.

For short transactions on small files, no differences to reads at random offsets can be observed: a sequential read at a single offset is the same as a random read.

For transactions of medium and long length, the strategy TS yields a much better throughput for reading at consecutive offsets than for reads at random offsets. This is a result of the much shorter locking phase and the less interference with other transactions. Again this is a side effect of the the leaf-based locking. Assume 10 read operations in a transaction. In the case of reading at 10 random offsets, each record is likely to be located in a different tree leaf. This results in the locking of 10 leafs and possibly blocks 10 concurrent, but otherwise independent transactions. In the case of 10 reads at consecutive offsets, only one or two leafs have to be locked, since each leaf contains several records at once. This minimized interference with other transactions.

The other strategies do not show any significant difference to the case of reading at random offsets. For these strategies the layout of the record tree is not that much important: 2PL locks individual records, whereas NOUNDO and CS do not use the record tree at all.

For large files, the benchmark shows similar results as for small files. The only significant difference between random and consecutive reads is again in the behavior of TS, which now scales much better for long transactions. This is caused by the same implementation details that have been described above.

Random writes

This benchmark simulates use cases with extensive write operations at random offsets, such as synchronizing updates to a database with a file. Figures 6.8 and 6.10 show the detailed results.

The strategies CS and NOUNDO do not scale at all. The strategies TS and 2PL do scale for some setups. For transactions of short and medium length, 2PL scales up to 7 threads; where it peaks at 432,157 transactions per second, respective 105,827 transactions per second. It stays roughly on that level for short transactions, but decreases for medium-length transactions, due to conflicts. Similar behavior can be observed for long transactions, where 2PL peaks with 3 threads at 8012 transactions per second; and afterwards decreases against zero. With TS, the main difference to reading transactions is the lack of necessary validation. Therefore, no conflicts are observed and the strategy should scale very well when compared to the pessimistic 2PL. Even though no conflicts happen, TS does not scale well. For short transactions it behaves similar to 2PL. For medium and long transactions, TS behaves like CS or worse.

The mediocre throughput of the TS strategy can be explained with the framework's internal setup. In the original adjustment each element in the record tree contains 2^9 child elements: each node contains 512 nodes or leafs and each leaf contains 512 records. Figure 6.9 shows the same benchmark with a different adjustment of the internal tree of record. For this test the number of child elements has been reduced to $2^5 = 32$. This leads to deeper trees with longer search time; but also reduces contention within the global tree and on individual leaf locks, as each lock now protects a smaller amount of records.

The purpose of this change is to illustrate the influence of the framework's internal parameters on the performance. For short transactions, the modified version has a clearly smaller throughput. It reaches at most two hundred thousand transactions per second: only half on the original setup. This can be explained with the higher overhead of searching through the record tree. The tree's depth has increased, so more elements need to be traversed. For transactions of medium size the situation is reversed: The original setup performs only half as good as the modified setup, which peaks around ninety thousand transactions per second at 11 threads. The less amount of contention on the leafs' internal locks allows for scaling up to this level and afterwards keeping the throughput there. No differences can be observed for long transactions. This case seems to be influenced by other factors than the record tree's setup.

Back to the original setup, when working on large files, no major difference in the general behavior of CS, NOUNDO, and 2PL are observable.

Due to less conflicts, the latter performs better for long transactions. Its peak throughput increases from 8012 transactions per second to 12,316 transactions per second. The TS strategy performs similar as for sequential reads. The throughput is again hampered by the absent reuse of internal data structures. Only for long transactions the throughput is higher than with small files, since the reduced internal contention enables better performance.

Sequential writes

This file-I/O benchmark is shown in Figures 6.11 and 6.12. It simulates use cases, such as paging out parts of a database, where a transaction writes large amounts of consecutive data to a file.

For short transactions on small files sequential writes behave similar to random writes, as they are both the same. For medium-length and long transactions, 2PL also behaves as with random writes, but with 15,078 transactions per second its peak throughput is twice as high. This can be attributed to the better locality of operations at consecutive offsets. Also, when writing at adjacent offsets, the framework's I/O component merges consecutive write actions into one single operation. This save the overhead of all but one system call. The strategy TS performs much better than with writes at random offsets. As with reads this can be attributed to the less contentious internal locking during the transaction's commit. The merging of write operations is applied for TS as well.

For large files, no difference can be seen for CS, NOUNDO, and 2PL. TS does not perform as well as for small files. Except for long transactions it shows the same performs as with random offsets. Its throughput increases to 11,234 with 10 threads, where it is roughly twice as high as CS and 2PL. Afterwards the throughput decreases again.

Random reads and writes, ratio 1:1

Figure 6.13 shows the performance of a benchmark with a read-write ratio of 1:1 for a file size of a 1 MiB. This simulates the update of independent entries of a database, like multiplying each by a specific value.

As in the tests before, the strategies CS and NOUNDO do not scale. NOUNDO again suffers from the common transaction overhead. Compared to CS, 2PL achieves an up to four times higher throughput for transactions of short and medium length. It peaks at 5 threads, with 355,334 short-length transactions per second and 60,345 medium-length transactions per second, then decreases, and stays roughly constant for more than 8 threads. For long transactions, the performance of 2PL is not that good. Around 3 threads its throughput peaks, with 4497 transactions per second, which is slightly better than CS. Afterwards decreases to 1292 transactions per second, which is only one third of CS.

For short transactions, TS scales with the number of threads, but performs worse than 2PL. With 8 threads, it reaches its peak at 292,738 transactions per second, which is similar throughput as 2PL. It stays at this level when more processors are added.

With the increase in transaction length, the performance of TS decreases significantly. For medium-length transactions, it performs somewhere between twenty and thirty thousand transactions per second; a level close to CS. For long transactions it performs between one thousand and two thousand transactions per second, which is similar to the other transactional strategies. The decrease in throughput can again be attributed to the increased amount of conflicts, the higher amount of necessary validations, and the more contentious internal locking.

The same tests have been performed for a file size of 100 MiB. The results are shown in Figure 6.14. In general the results are the same as for the other tests: For the strategies of CS, NOUNDO, and 2PL no significant differences can be observed, whereas TS performs much worse for transactions of short and medium length, but better for long transactions.

Random reads and writes, ratio n:1

Benchmarks with different read-write ratios are shown in Figures 6.15, 6.16, 6.17, 6.18, 6.19, 6.20. In these cases the number of writes is 33, 20, and 11 percent. This simulates the update of constraint database entries, such as recomputing one entry from several others.

The trends of all graphs are similar to the case of 50 percent writes. The major difference among these tests is the decrease in throughput that comes with the increase in reads per transactions. This can be attributed to the additional processing in general and the increased number of locking, validation, and conflicts that arise from these additional reads.

Summary

With the given setup, the performance measurements often show that the framework scales up to a certain amount of concurrent transactions, but then the throughput stalls or decreases again. This can typically be explained by the increasing amount of conflicts among transactions and internal contention. The framework can outperform non-transactional code for short and medium-length transactions, but rarely does with long transactions.

The strategy 2PL shows some strange behavior with reading transactions of medium and long size. Its throughput first scales with the number of concurrent transactions. When reaching around seven transactions it drops to values near or below the single-threaded case. For higher numbers of concurrent transactions the throughput increases again. The effect is particularly visible in Figures 6.4, 6.5, 6.14, and 6.20. Its cause is currently unknown. The effect is independent of the file size. It is also not related to the number of aborts, as the random-read tests do not generate aborts at all. The file offsets are generated pseudo-randomly via `rand_r`. To eliminate certain sequences of random numbers that yield a high amount of interference among threads, some tests were tried with different initial seed values. The results show the same anomaly, so the effect seems unrelated to the file-offset generation and framework-internal interference. Possibly this phenomenon is caused by Linux' thread scheduling.

At the moment the strategy TS seems useless for buffer I/O. It rarely outperforms 2PL for the tested setup, notably in Figures 6.4 and 6.12. Otherwise, it performs worse. Maybe a scenario with heavy write contention, and thus many false conflicts with the strategy 2PL, could benefit by using TS. A possible use case might be FIFO writes, where validation is not necessary and 2PL generates many false conflicts.

6.5 General considerations

Having presented an evaluation of the framework with regard to the specified design goals, this section discusses Taglibc with respect to the time complexity of its underlying algorithms and the number of code lines. These metrics do not provide objective results, but give an estimate of the software's performance, design, and bug count.

Time complexity

Time complexity describes the increase in computation steps when the input n increases linearly. In the case of the framework, n is the number of actions that are contained in a transaction's local history.

Before executing its first action, a component has to register itself with the framework. This has to be done only once, so its complexity is $O(1)$. While the action is executed, it can append events to the transaction's history. The log is reallocated and the event is copied to its end.

Component	Lines of code
Core (log and error handling)	1,241
Allocator	334
File-descriptor I/O	10,099
File system	717
Others	266
Entry-point declarations	232
Σ	12,889

Table 6.2: The number of lines of C code and entry-point declarations in various components.

The framework's internal allocation function allocates with respect to powers of two, so the reallocation has a complexity of $O(\log n)$.[3] The copy operation is done for every action, so its has a complexity of $O(n)$. The execution of actions itself has an overhead of $O(n)$.

However, when using optimistic domains, it is necessary to re-validate their state after each executed (read) operation. The overhead of the validation step can hardly be estimated. Its input m is not the number of actions, but the number of data items to validate. This depends on the domains used by a transaction and the actions executed on them. For example, validating the record timestamps of a file buffer is likely to have a higher complexity than testing a single value. The worst case is a large amount of optimistic read operations on a large number of domains. The validation has to be done after each read operation, so the execution complexity is $O(n \times m)$. Assuming that each action yields one necessary validation, $m = n,$ a complete execution has a complexity of $O(n^2)$.

When committing a transaction, all optimistic domains have to be locked to guarantee correctness. To prevent deadlocks, the locking has to follow a canonical order. If each executed action works on an individual value, the number of values is n. With a good sorting algorithm, the complexity of sorting the values to a canonical order is $O(n \log n)$.

Afterwards all domains are locked, each is validated, the transaction's history is either applied or undone, and each domain is unlocked again. Each of these steps has an inherent complexity of $O(n)$.

To summarize, the complexity strongly depends on the validation step. If validation is not necessary, the framework's implementation imposes a complexity of $O(n)$. If validation is necessary, the complexity increases up to $O(n^2)$.

Size

Size is measured in *lines of code*. The number of code lines of Taglibc and its components is shown in Table 6.2.[4]

The core framework, which consists of the event log and the error-handling infrastructure, contains only 1,241 lines of code. By far the largest part is the component for file-descriptor I/O with 10,099 lines of code. This comes of no surprise as it provides the most actions and incorporates interfaces for socket support as well. It also contains quite a lot of internal data structures. The size of other components, such as the memory allocator and the file system, is almost negligible with only a few hundred code lines each.

[3]The framework's allocator uses the system's `malloc` implementation. Although Figure 6.1 indicates otherwise, this might have an influence here.

[4]These statistics have been generated using David A. Wheeler's 'SLOCCount'.

Most of Taglibc's entry points are generated automatically. Their declarations account of another 232 lines of code. The complete package with entry-point declarations has been measured with 12,889 lines of code.

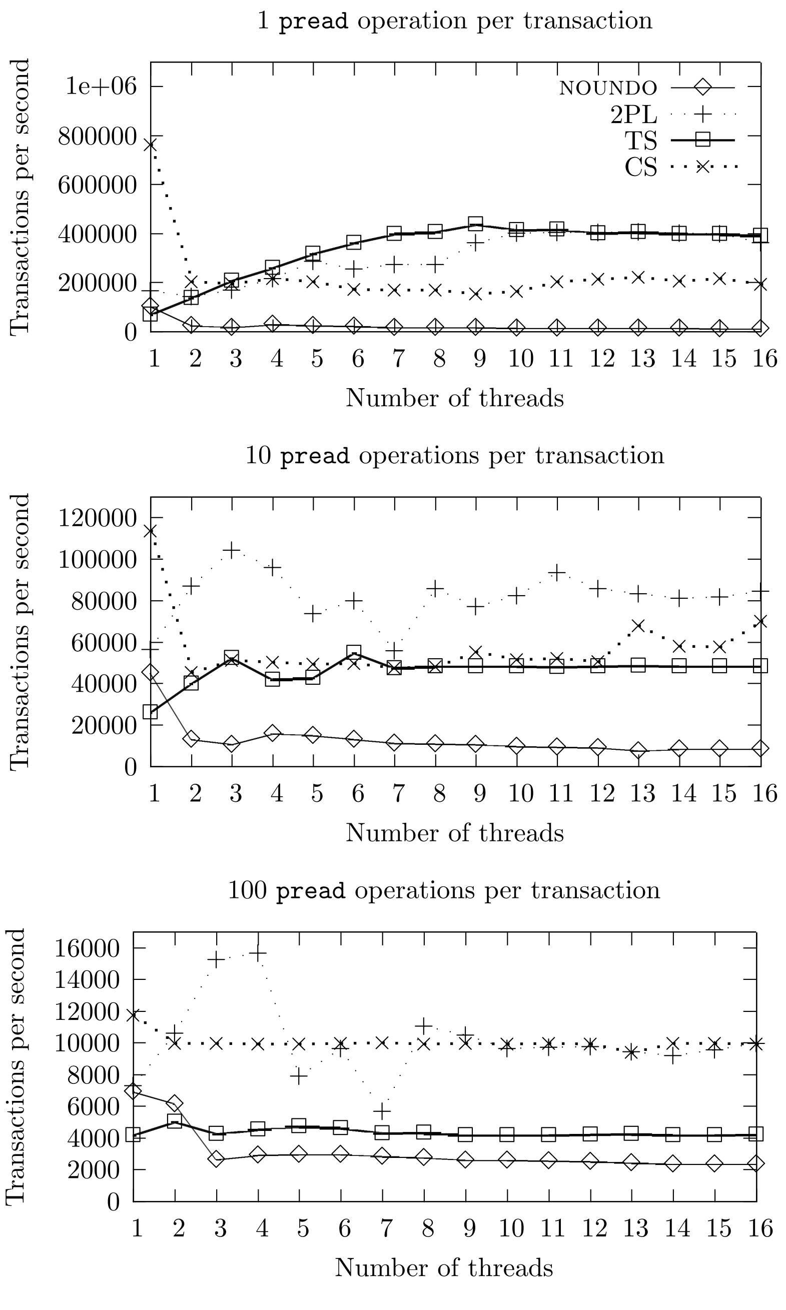
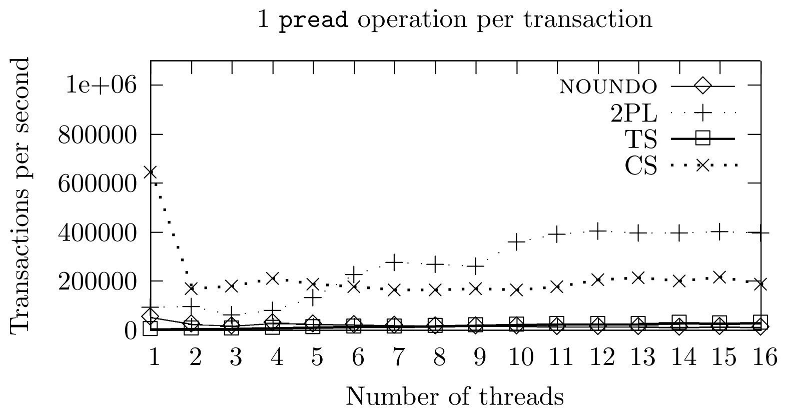
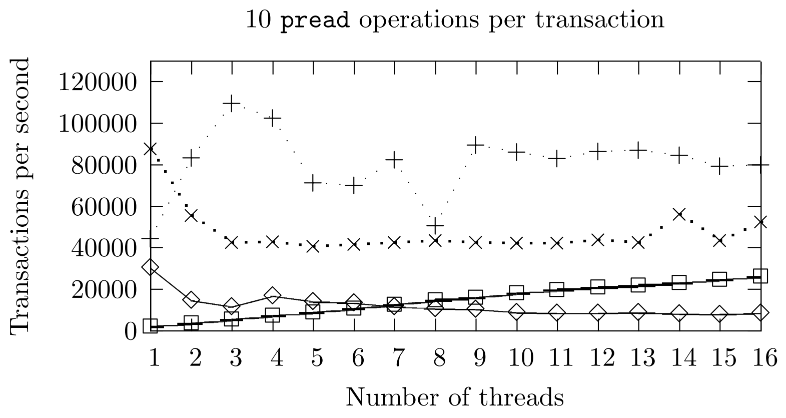


Figure 6.4: Reads at random offsets in a file of 1 MiB size. Each transaction consists of a number of `pread` operations, which operate at random offsets within the file. This simulates reads of sole data items, such as extracting specific entries from a database file.





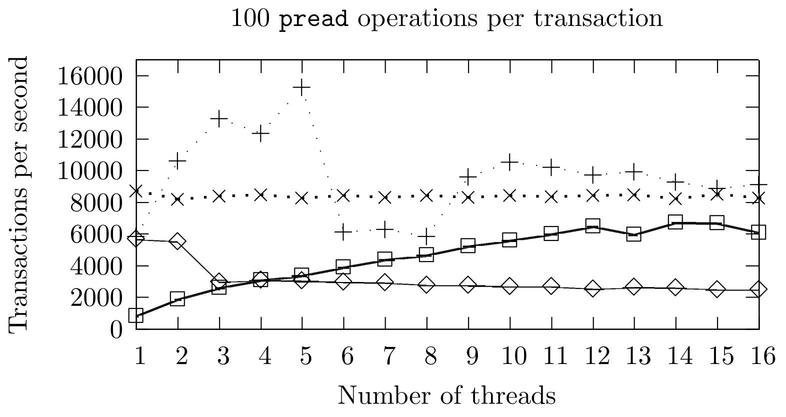


Figure 6.5: Reads at random offsets in a file of 100 MiB size. Each transaction consists of a number of `pread` operations, which operate at random offsets within the file. This simulates reads of sole data items, such as extracting specific entries from a database file.

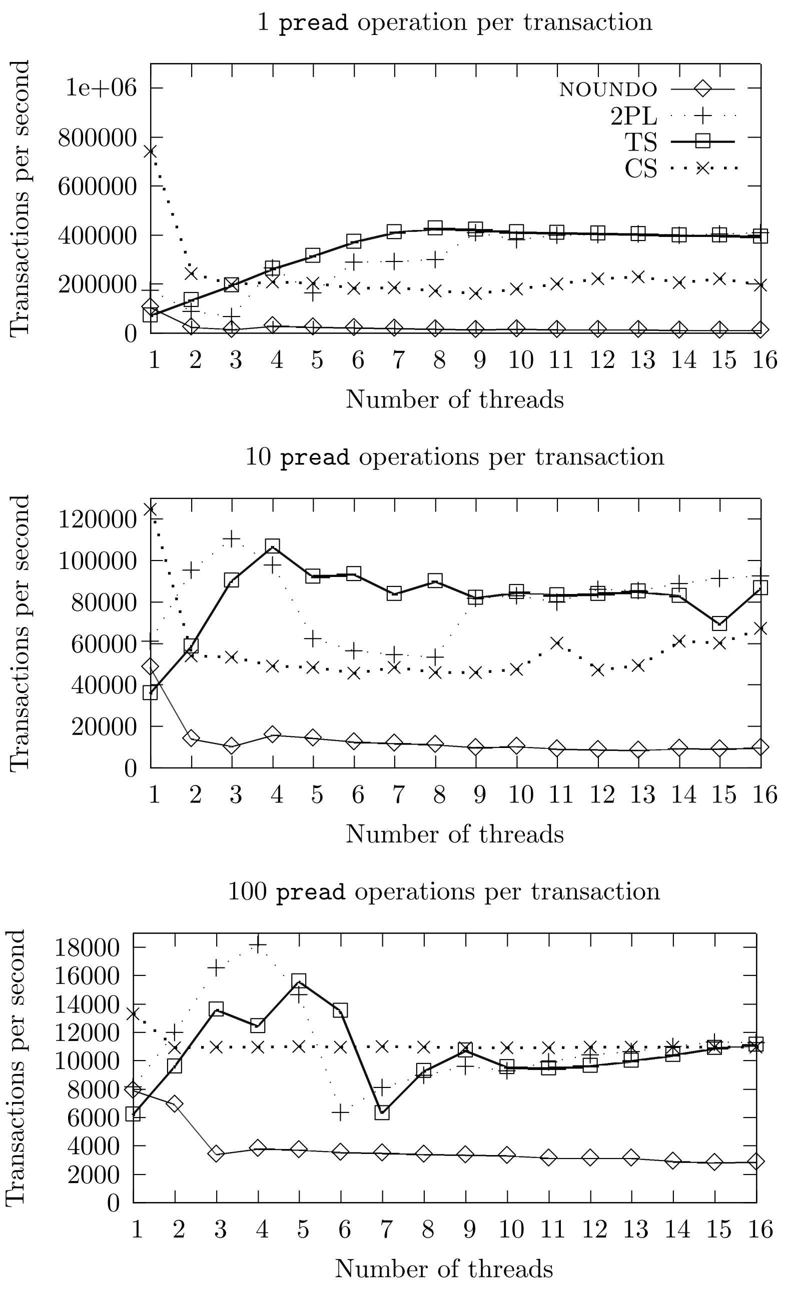


Figure 6.6: Reads at sequential offsets in a file of 1 MiB file size. Each transaction consists of a number of `pread` operations, which operate at sequential offsets from a random initial value. This simulates reads of constrained data items, such as reconstructing memory content from a database dump.

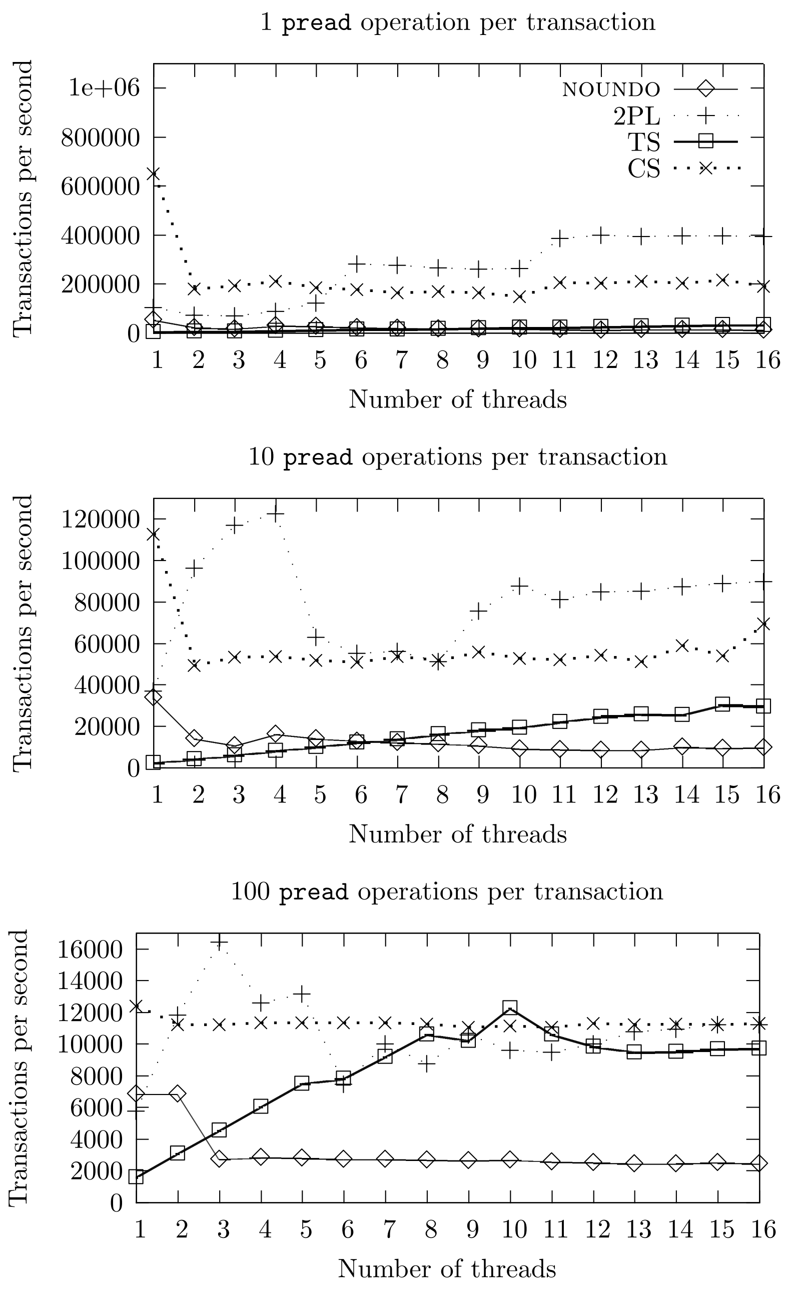


Figure 6.7: Reads at sequential offsets in a file of 100 MiB file size. Each transaction consists of a number of `pread` operations, which operate at sequential offsets from a random initial value. This simulates reads of constrained data items, such as reconstructing memory content from a database dump.

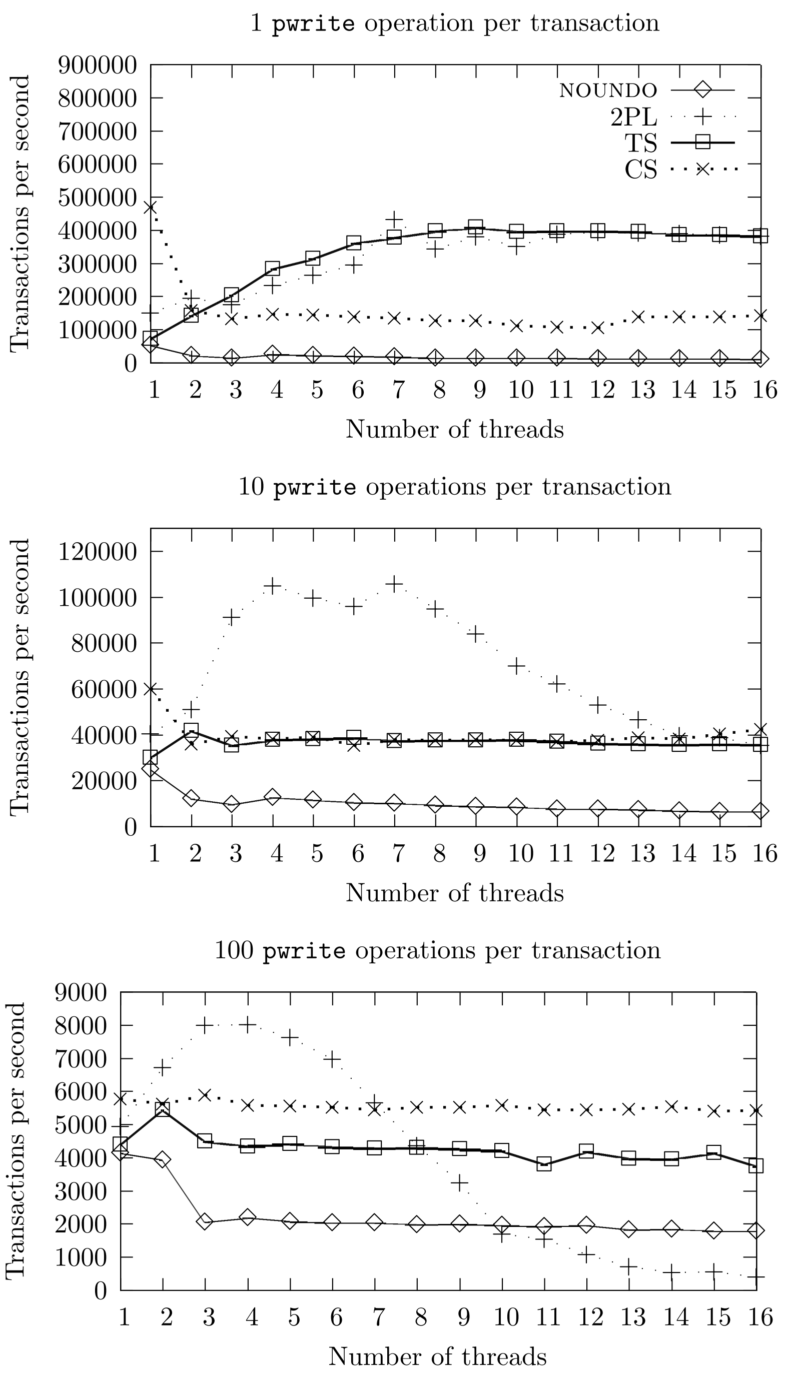
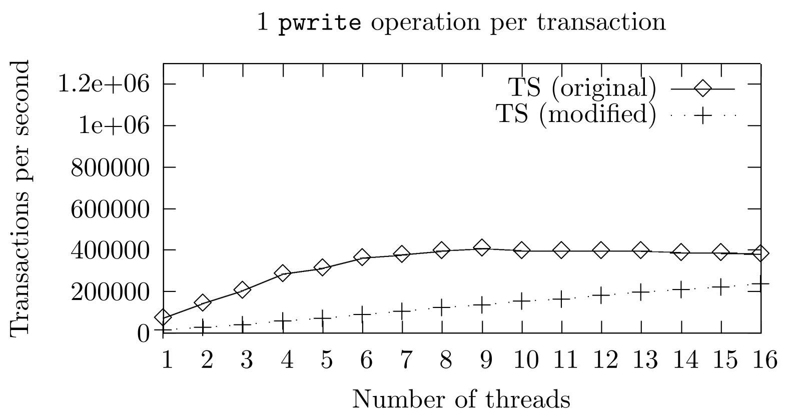
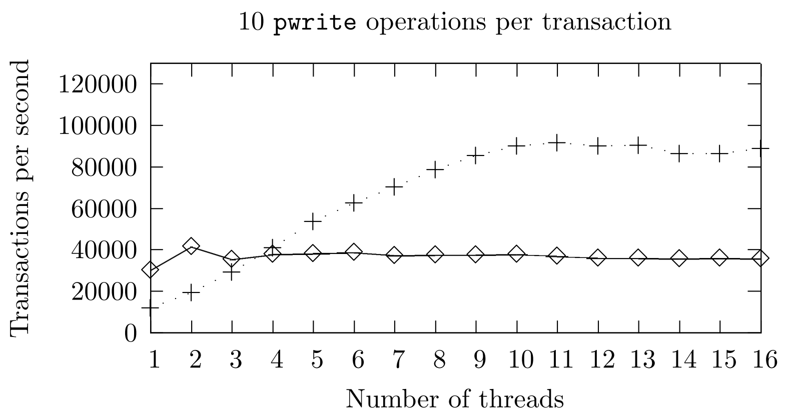


Figure 6.8: Writes at random offset in a file of 1 MiB file size. Each transaction consists of a number of `pwrite` operations, which operate at random offsets within the file. This simulates writes of changed data items, such as synchronizing updates to a database with a file.





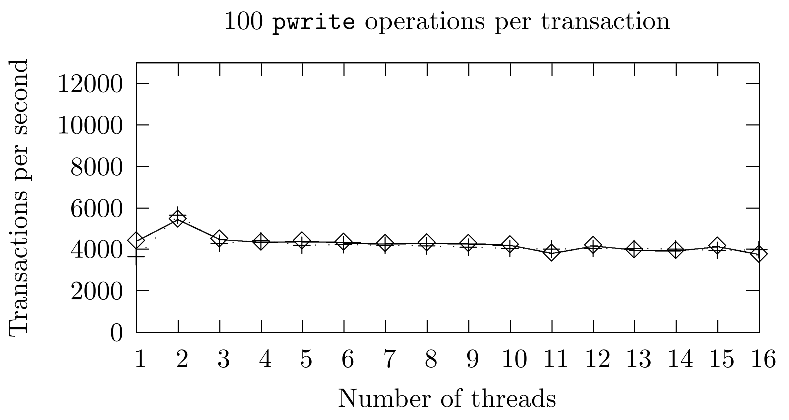


Figure 6.9: Writes at random offset in a file of 1 MiB file size. Both transactions use the optimistic strategy `TS`, but each has a different internal setup of the record tree.

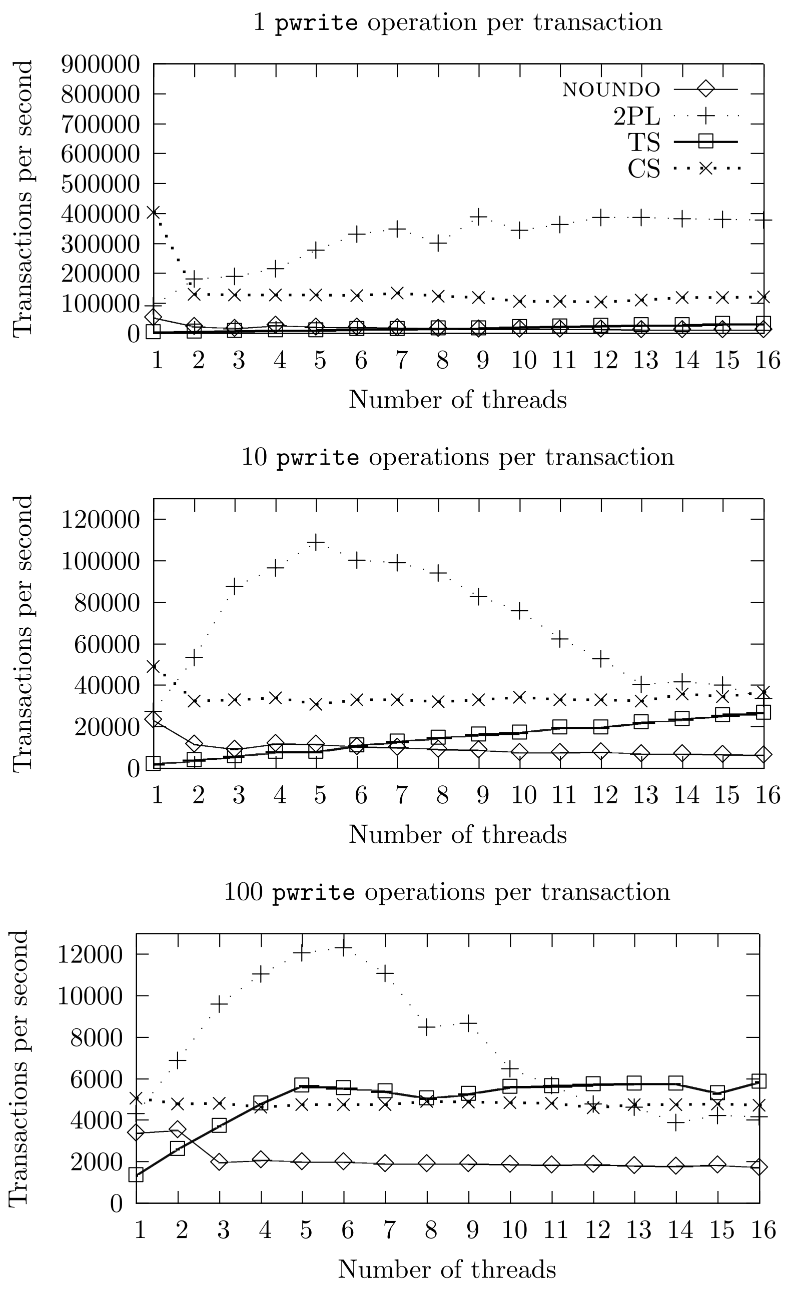


Figure 6.10: Writes at random offset in a file of 100 MiB file size. Each transaction consists of a number of `pwrite` operations, which operate at random offsets within the file. This simulates writes of changed data items, such as synchronizing updates to a database with a file.

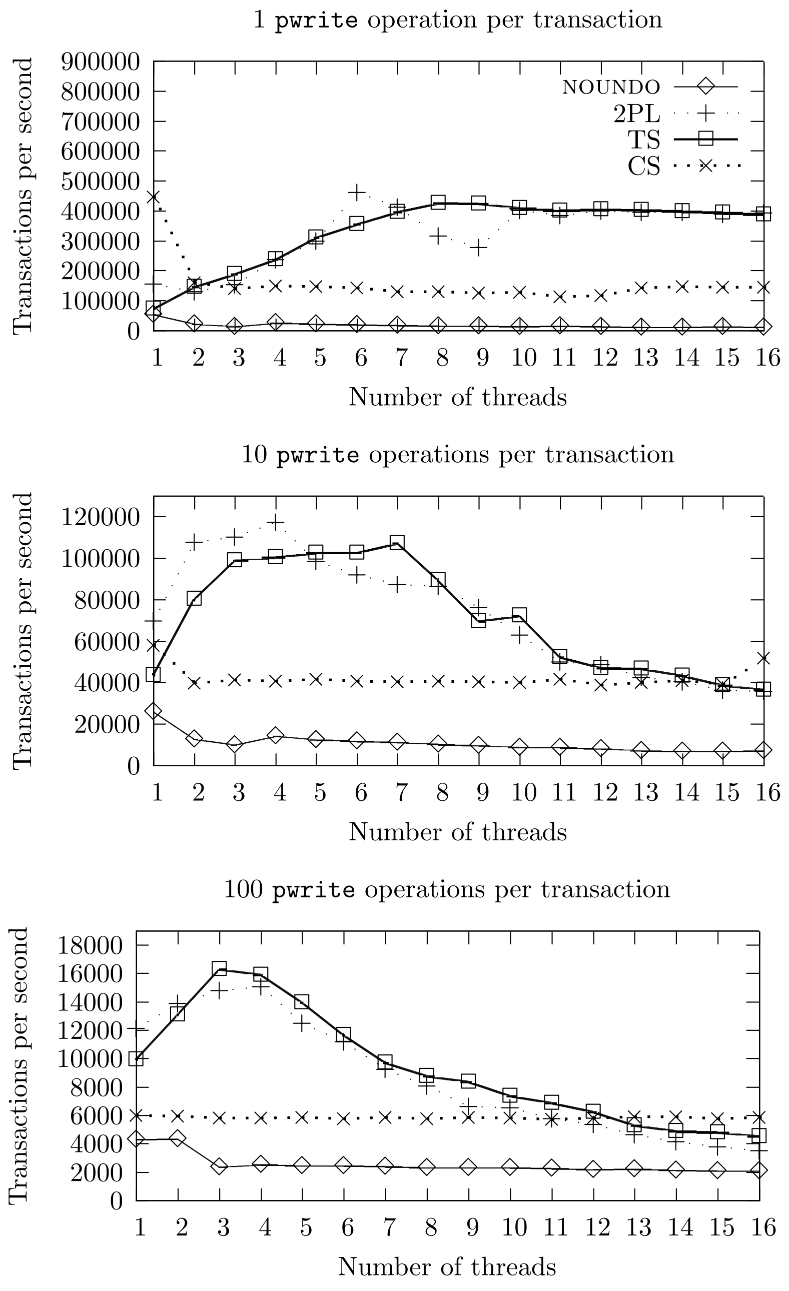


Figure 6.11: Writes at sequential offsets in a file of 1 MiB size. Each transaction consists of a number of `pwrite` operations, which operate at sequential offsets from a random initial value. This simulates writes of constrained data items, such as dumping the content of memory to a file.

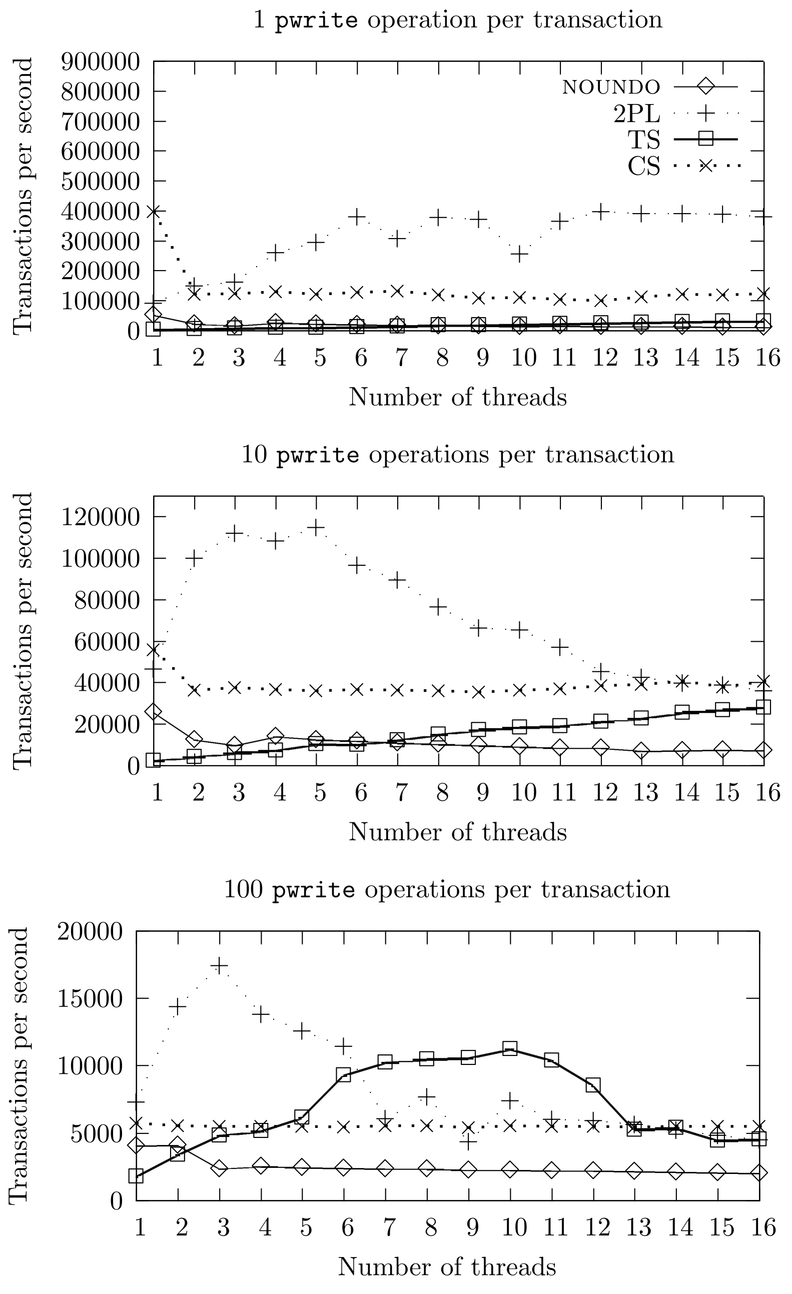


Figure 6.12: Writes at sequential offsets in a file of 100 MiB size. Each transaction consists of a number of `pwrite` operations, which operate at sequential offsets from a random initial value. This simulates writes of constrained data items, such as dumping the content of memory to a file.

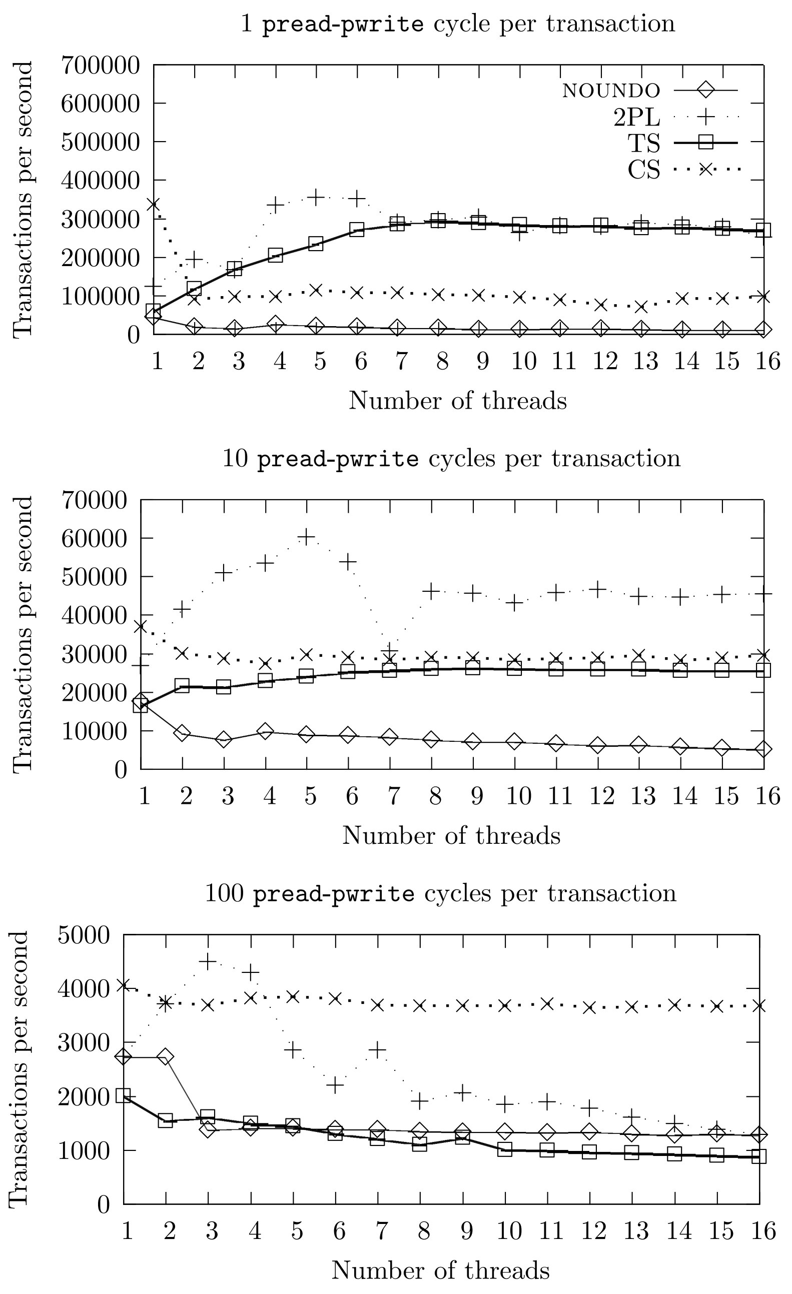


Figure 6.13: Reads and writes at random offsets in a file of 1 MiB size. Each transaction consists of a number of read-write cycles. Each cycle consists of one `pread` operation at a random offset within the file, and one `pwrite` operation at the same offset. This simulates updates of independent data items, such as multiplying a set of database entries by some fixed value.

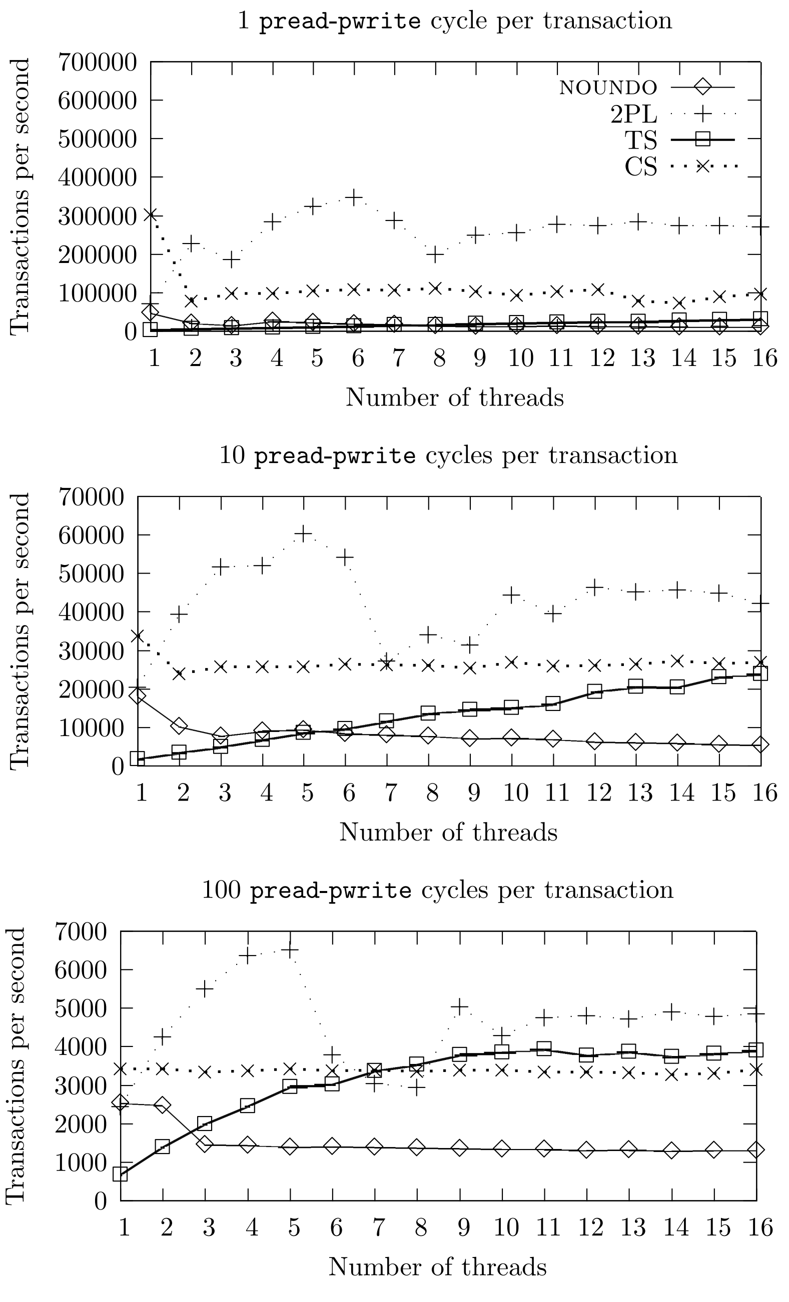


Figure 6.14: Reads and writes at random offsets in a file of 100 MiB size. Each transaction consists of a number of read-write cycles. Each cycle consists of one `pread` operation at a random offset within the file, and one `pwrite` operation at the same offset. This simulates updates of independent data items, such as multiplying a set of database entries by some fixed value.

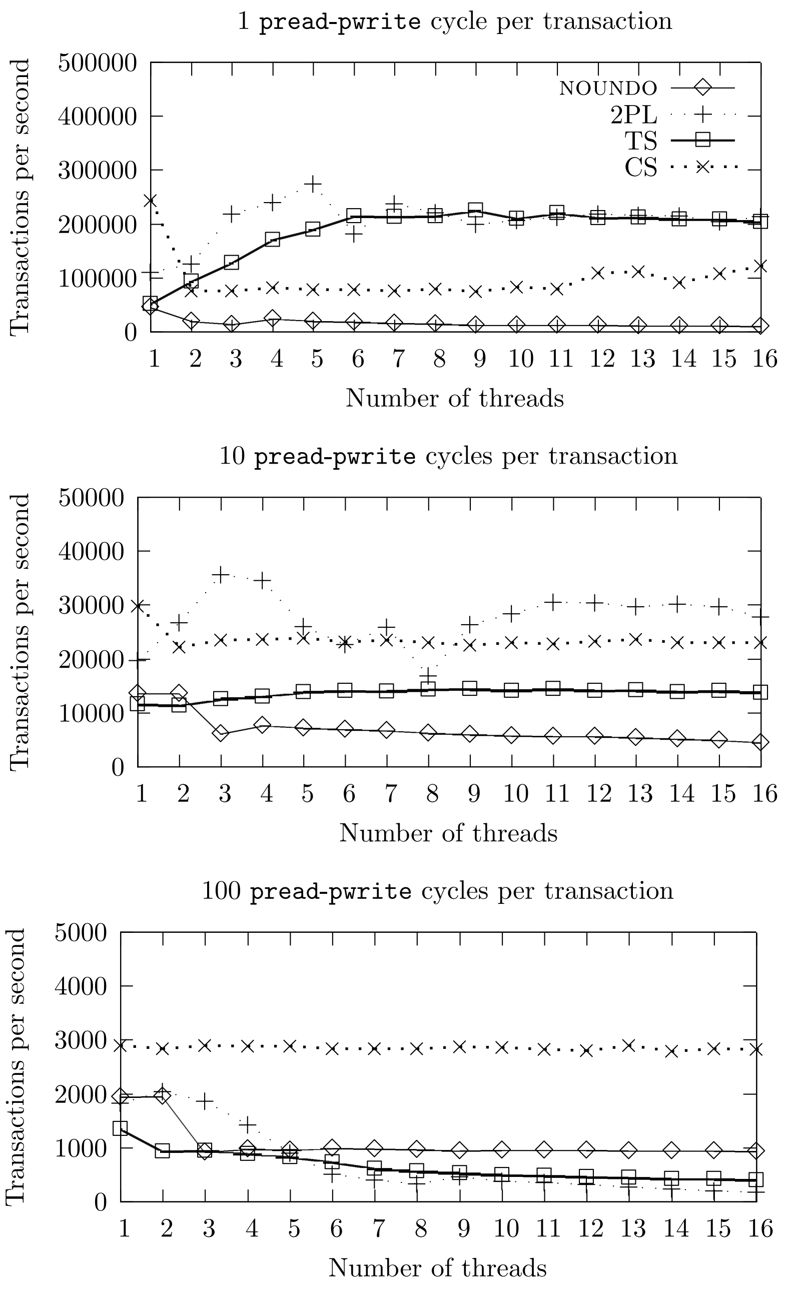
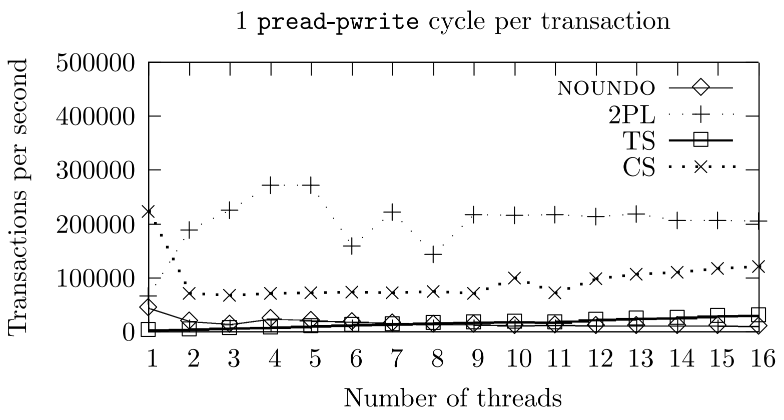
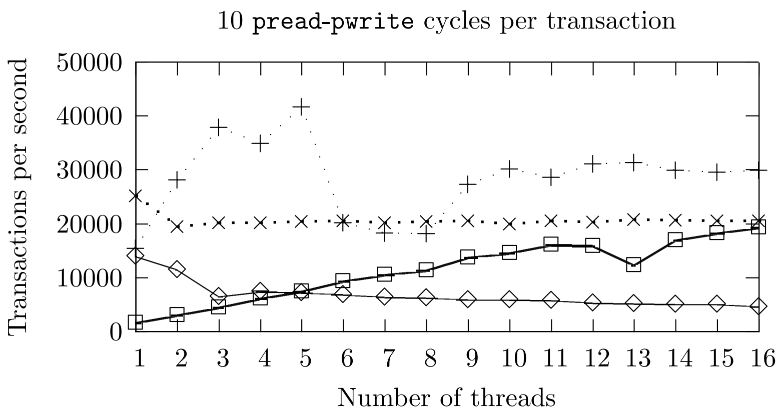


Figure 6.15: Reads and writes at random offsets in a file of 1 MiB size. Each transaction consists of a number of read-write cycles. Each cycle consists of two `pread` operations at random offsets within the file, and one `pwrite` operation at the last offset. This simulates updates of constrained data items, like recomputing a set of database entries from others.





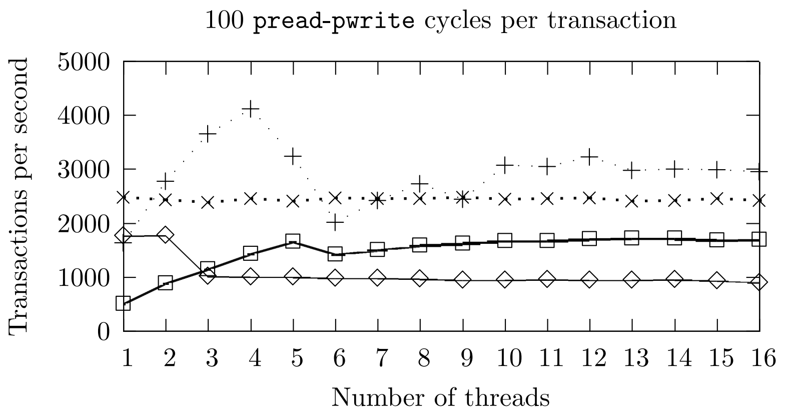


Figure 6.16: Reads and writes at random offsets in a file of 100 MiB size. Each transaction consists of a number of read-write cycles. Each cycle consists of two `pread` operations at random offsets within the file, and one `pwrite` operation at the last offset. This simulates updates of constrained data items, such as recomputing a set of database entries from others.

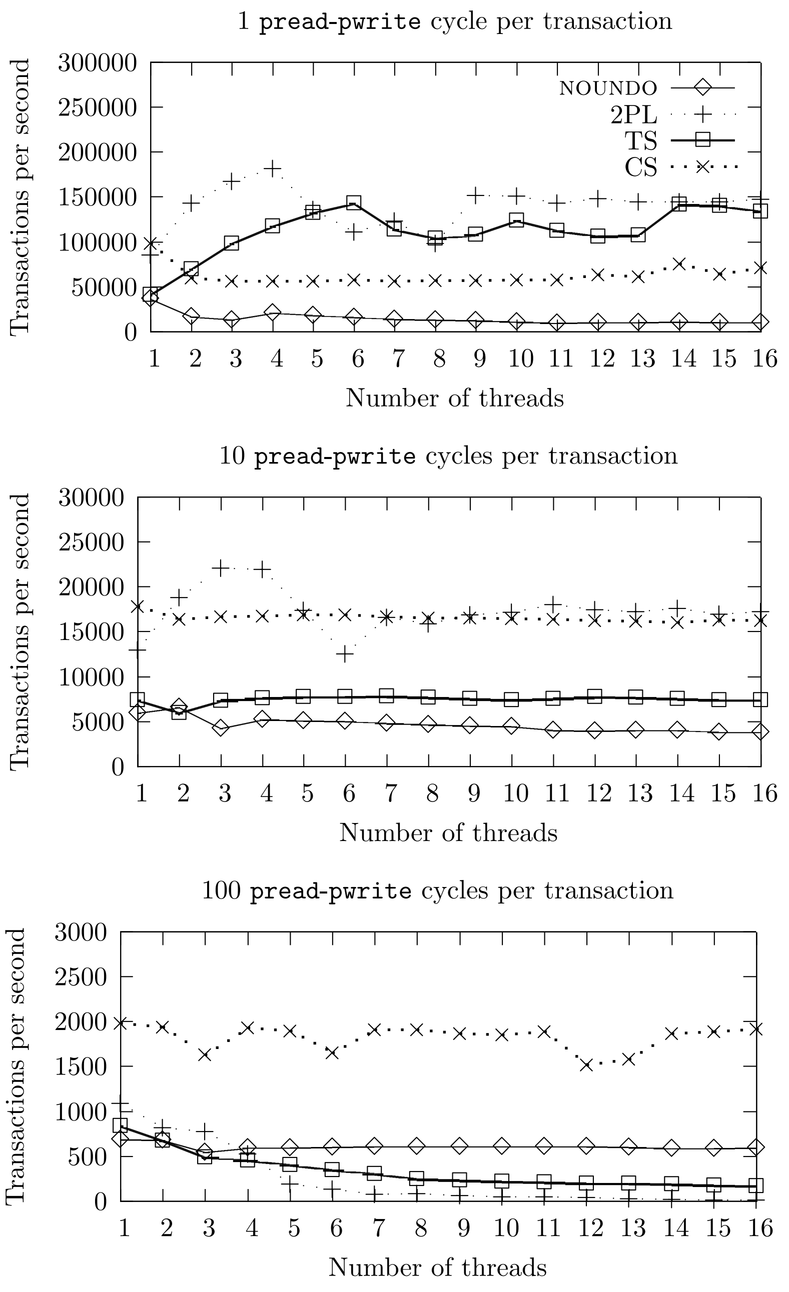


Figure 6.17: Reads and writes at random offsets in a file of 1 MiB size. Each transaction consists of a number of read-write cycles. Each cycle consists of four `pread` operations at random offsets within the file, and one `pwrite` operation at the last offset. This simulates updates of constrained data items, such as recomputing a set of database entries from others.

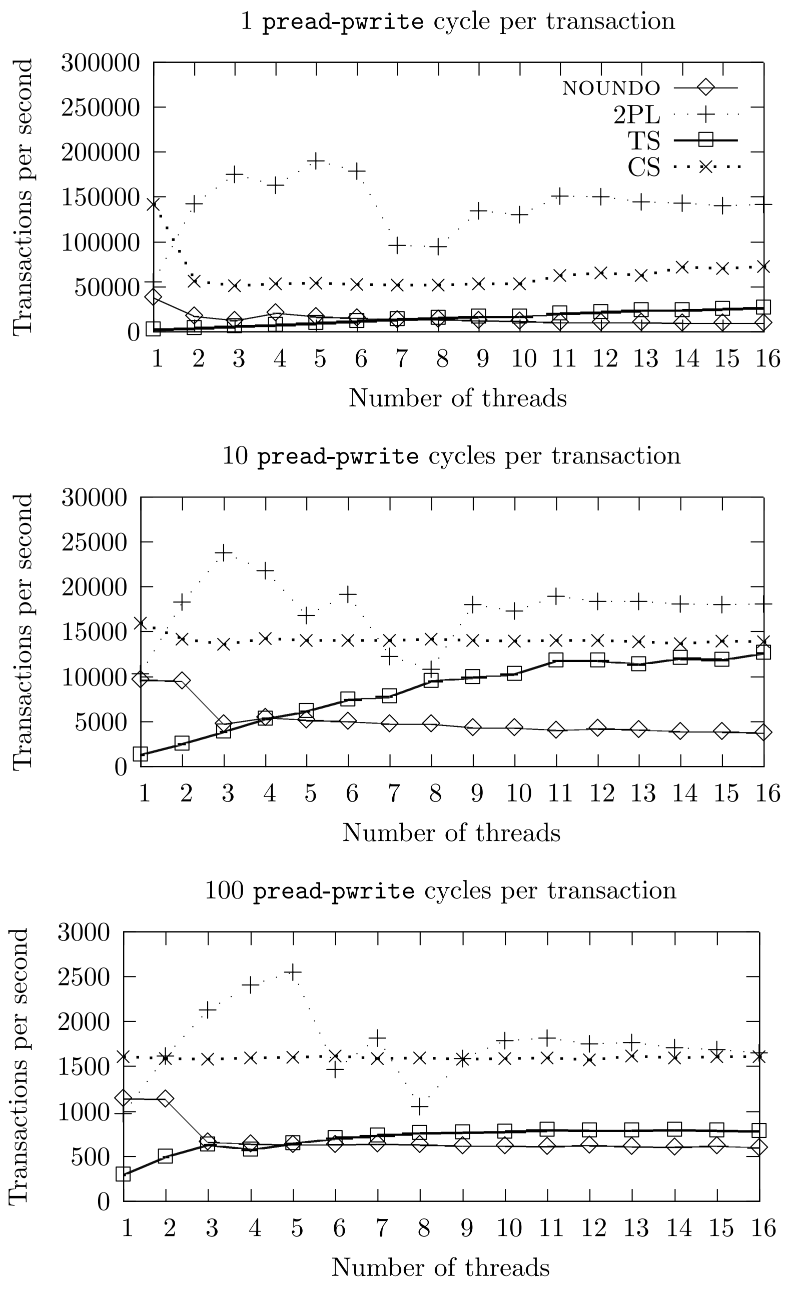
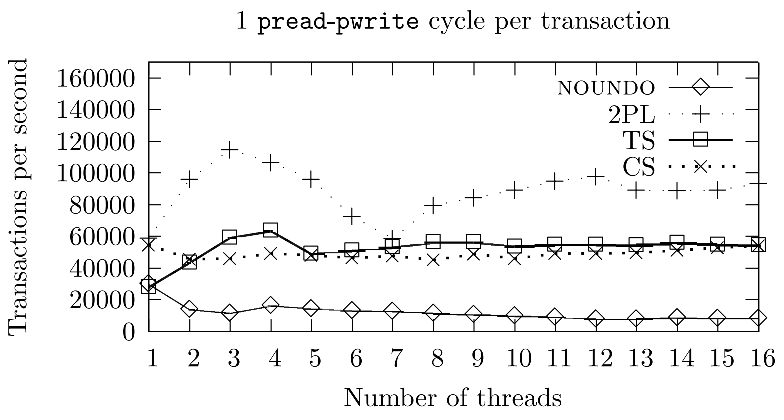
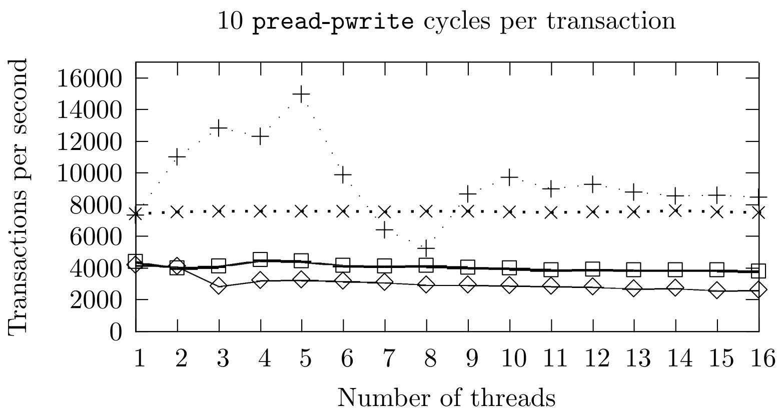


Figure 6.18: Reads and writes at random offsets in a file of 100 MiB size. Each transaction consists of a number of read-write cycles. Each cycle consists of four `pread` operations at random offsets within the file, and one `pwrite` operation at the last offset. This simulates updates of constrained data items, such as recomputing a set of database entries from others.





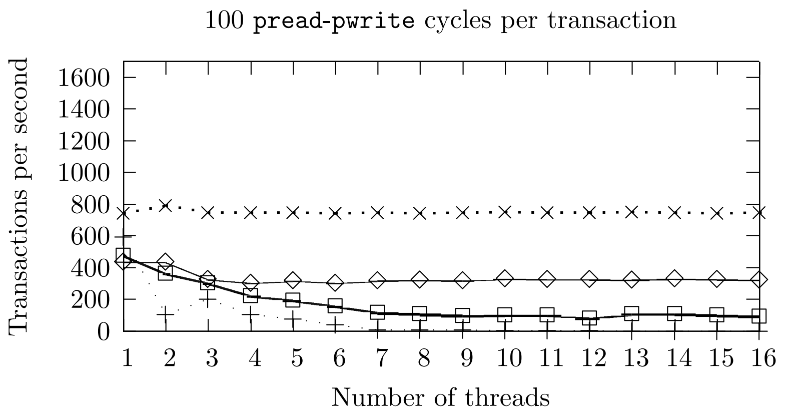


Figure 6.19: Reads and writes at random offsets in a file of 1 MiB size. Each transaction consists of a number of read-write cycles. Each cycle consists of eight `pread` operations at random offsets within the file, and one `pwrite` operation at the last offset. This simulates updates of constrained data items, such as recomputing a set of database entries from others.

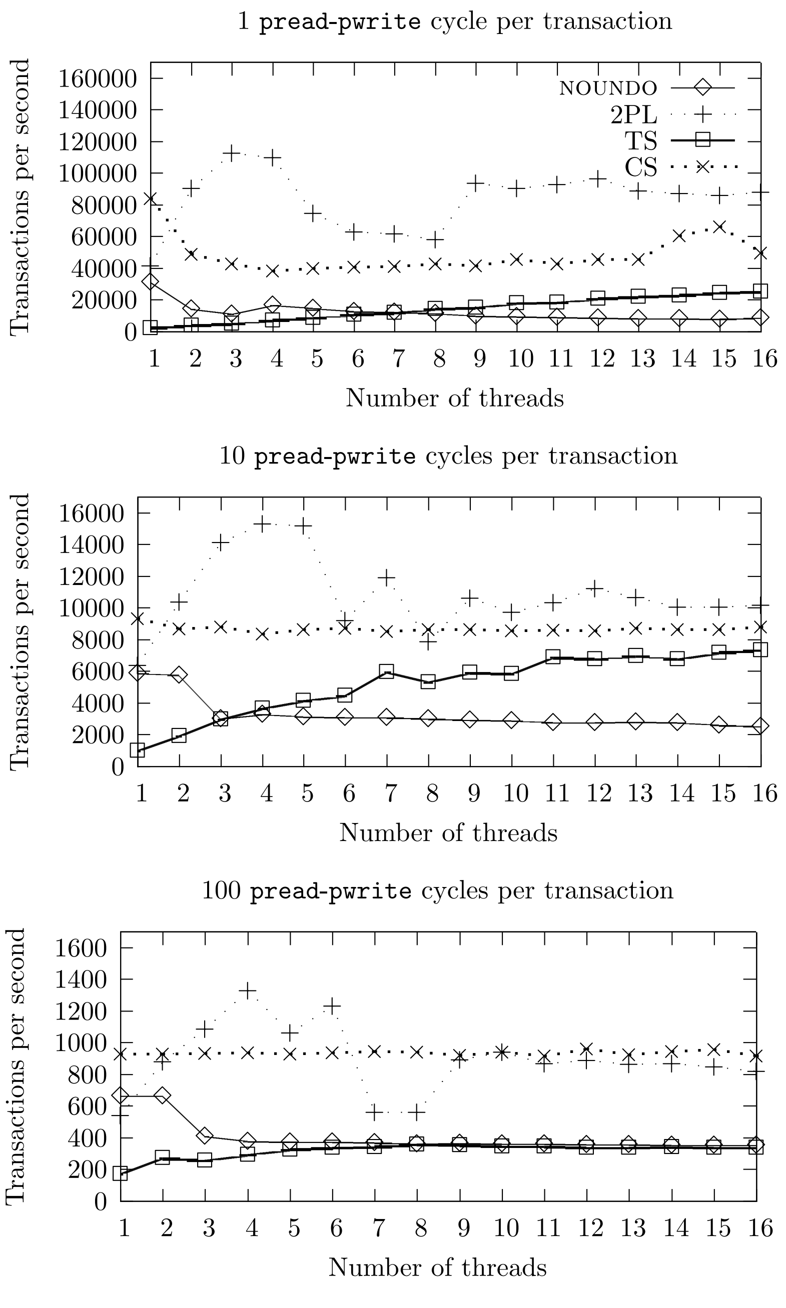


Figure 6.20: Reads and writes at random offsets in a file of 100 MiB size. Each transaction consists of a number of read-write cycles. Each cycle consists of eight `pread` operations at random offsets within the file, and one `pwrite` operation at the last offset. This simulates updates of constrained data items, such as recomputing a set of database entries from others.

Benchmark	Iterations	Threads	Strategy					
			1 MiB			100 MiB		
			NOUNDO	2PL	TS	NOUNDO	2PL	TS
Random reads	1	1	0	0	0	0	0	0
		2	1,368,863	0	0	1,405,091	0	0
		3	1,888,243	0	0	1,689,038	0	0
		4	165,331	0	0	129,920	0	0
		5	1,400,280	0	0	1,524,632	0	0
		6	2,479,083	0	0	2,500,926	0	0
		7	3,177,490	0	0	3,295,052	0	0
		8	3,829,704	0	0	3,666,054	0	0
		9	4,429,858	0	0	4,212,558	0	0
		10	4,421,658	0	0	4,568,289	0	0
		11	4,751,481	0	0	4,660,336	0	0
		12	5,034,354	0	0	5,170,851	0	0
		13	5,267,059	0	0	5,724,244	0	0
		14	5,593,215	0	0	5,436,942	0	0
		15	5,510,875	0	0	5,614,483	0	0
		16	5,798,994	43,273,203	0	5,827,566	0	0
	10	1	0	0	0	0	0	0
		2	775,129	0	0	866,083	0	0
		3	1,205,655	0	0	1,287,990	0	0
		4	183,264	0	0	131,242	0	0
		5	926,659	0	0	878,361	0	0
		6	1,628,806	0	0	1,550,021	0	0
		7	2,107,922	0	0	2,143,091	0	0
		8	2,691,158	0	0	2,635,803	0	0
		9	3,225,610	0	0	3,075,167	0	0
		10	3,455,679	0	0	3,397,746	0	0
		11	3,922,657	0	0	3,864,190	0	0
		12	4,436,169	0	0	3,878,818	0	0
		13	4,364,779	0	0	4,593,733	0	0
		14	4,291,865	0	0	4,528,308	0	0
		15	4,333,283	0	0	4,429,694	0	0
		16	4,493,143	0	0	4,353,989	0	0
	100	1	0	0	0	0	0	0
		2	13,057	0	0	17,252	0	0
		3	270,926	0	0	282,858	0	0
		4	45,602	0	0	25,464	0	0
		5	184,704	0	0	187,245	0	0
		6	363,796	0	0	358,456	0	0
		7	541,082	0	0	515,041	0	0
		8	661,257	0	0	685,194	0	0
		9	790,218	0	0	805,488	0	0
		10	870,580	0	0	942,928	0	0
		11	1,079,957	0	0	1,156,694	0	0
		12	1,036,805	0	0	1,398,276	0	0
		13	1,317,591	0	0	1,290,121	0	0
		14	1,594,134	0	0	1,538,163	0	0
		15	1,659,902	0	60,201,997	1,681,053	0	0
		16	1,526,326	0	0	1,800,116	0	0

Sequential reads	1	1	0	0	0	0	0	0
		2	1,354,532	0	0	1,327,062	0	0
		3	1,636,651	0	0	1,648,954	0	0
		4	95,485	0	0	106,620	0	0
		5	1,523,577	0	0	1,555,571	0	0
		6	2,461,918	0	0	2,456,449	0	0
		7	3,284,288	0	0	3,300,451	0	0
		8	3,697,960	0	0	3,742,585	0	0
		9	4,207,175	0	0	4,030,179	0	0
		10	4,859,563	0	0	4,594,921	0	0
		11	4,831,609	0	0	4,808,384	0	0
		12	5,235,798	0	0	5,086,031	0	0
		13	5,284,014	0	0	5,392,329	0	0
		14	5,358,105	0	0	5,536,072	0	0
		15	5,569,366	0	0	5,742,928	0	0
		16	5,639,888	0	0	5,809,279	0	0
	10	1	0	0	0	0	0	0
		2	822,340	0	0	825,906	0	0
		3	1,156,787	0	0	1,164,380	0	0
		4	71,289	0	0	80,804	0	0
		5	874,972	0	0	951,545	0	0
		6	1,493,383	0	0	1,550,742	0	0
		7	2,126,416	0	0	2,199,627	0	0
		8	2,660,015	0	0	2,746,637	0	0
		9	2,964,155	0	0	3,116,196	0	0
		10	3,632,380	0	0	3,574,378	0	0
		11	4,023,344	0	0	3,954,892	0	0
		12	4,227,609	0	0	4,058,288	0	0
		13	4,252,239	0	0	4,380,428	0	0
		14	4,218,888	0	0	4,273,656	0	0
		15	4,341,184	0	0	4,374,943	0	0
		16	4,338,484	0	0	4,341,712	0	0
	100	1	0	0	0	0	0	0
		2	5,685	0	0	13,740	0	0
		3	339,439	0	0	263,215	0	0
		4	46,688	0	0	121,131	0	0
		5	269,679	0	0	177,093	0	0
		6	429,881	0	0	335,227	0	0
		7	649,603	0	0	489,579	0	0
		8	820,695	0	0	651,173	0	0
		9	996,206	0	0	814,498	0	0
		10	1,123,269	0	0	918,717	0	0
		11	1,411,402	0	0	1,041,298	0	0
		12	1,540,334	0	0	1,258,166	0	0
		13	1,438,615	0	0	1,366,741	0	0
		14	1,918,645	0	0	1,572,023	0	0
		15	2,026,377	0	0	1,639,859	0	0
		16	2,050,457	0	0	1,560,311	0	0
	1	1	0	0	0	0	0	0
		2	1,297,688	139	0	1,267,875	1	0
		3	1,635,911	192	0	1,658,331	1	0
		4	211,372	487	0	272,279	7	0
		5	1,302,774	523	0	1,235,675	8	0
		6	2,300,843	747	0	2,256,382	8	0
		7	3,153,365	1,911	0	3,155,879	8	0
		8	3,688,506	3,086	0	3,820,377	5	0
		9	4,196,763	2,560	0	4,328,151	14	0
		10	4,577,331	1,721	0	4,929,505	9	0
		11	4,637,316	2,423	0	4,946,625	8	0
		12	5,111,689	1,978	0	5,330,427	12	0
		13	5,330,238	3,373	0	5,188,429	62	0
		14	5,616,083	1,521	0	5,453,466	8	0
		15	5,452,041	1,500	0	5,783,122	18	0
		16	6,151,253	2,026	0	5,986,577	8	0

		1	0	0	0	0	0	0
		2	718,282	16,589	0	678,011	138	0
		3	1,045,866	45,290	0	1,046,739	294	0
		4	223,613	69,002	0	263,837	590	0
		5	786,799	79,733	0	808,561	661	0
		6	1,283,523	87,621	0	1,281,100	888	0
		7	1,856,604	109,808	0	1,806,870	1,024	0
Random writes	10	8	2,285,947	109,985	0	2,203,997	1,007	0
		9	2,793,096	112,417	0	2,792,321	1,085	0
		10	2,968,317	115,537	0	3,096,343	1,042	0
		11	3,406,303	139,908	0	3,465,656	1,011	0
		12	3,968,713	196,810	0	3,411,472	19,383	0
		13	3,974,968	230,227	0	4,016,217	27,330	0
		14	4,455,098	177,119	0	4,213,841	9,694	0
		15	4,464,066	319,324	0	4,354,117	39,070	0
		16	4,522,599	367,616	0	4,553,869	3,657	0
		1	0	0	0	0	0	0
		2	9,784	315,442	0	6,870	2,070	0
		3	196,670	775,580	0	198,755	7,110	0
		4	29,554	1,066,570	0	74,272	12,497	0
		5	186,734	1,252,146	0	124,423	17,499	0
		6	249,933	1,326,765	0	242,082	20,510	0
		7	385,970	1,175,560	0	358,435	18,741	0
	100	8	499,406	1,009,857	0	500,545	20,279	0
		9	627,754	973,959	0	643,509	15,089	0
		10	738,307	815,034	0	728,197	22,700	0
		11	890,951	1,013,173	0	987,373	21,002	0
		12	842,527	1,011,237	0	835,068	51,320	0
		13	1,192,006	1,125,549	0	684,354	37,672	0
		14	1,185,632	1,049,067	0	1,188,343	162,706	0
		15	1,237,940	943,982	0	1,147,691	98,019	0
		16	1,496,284	959,438	0	1,153,450	93,164	0
		1	0	0	0	0	0	0
		2	1,301,581	71	0	1,262,470	2	0
		3	1,578,663	203	0	1,631,199	2	0
		4	260,562	464	0	232,137	9	0
		5	1,384,180	1,511	0	1,375,489	7	0
		6	2,287,836	2,615	0	2,360,793	8	0
		7	3,167,712	1,796	0	3,183,283	10	0
	1	8	3,791,915	2,138	0	3,788,499	13	0
		9	4,377,496	1,029	0	4,137,109	15	0
		10	4,596,318	2,528	0	4,157,795	4	0
		11	4,753,568	3,003	0	4,861,491	18	0
		12	4,987,534	2,175	0	4,908,018	11	0
		13	5,163,576	2,488	0	5,227,493	14	0
		14	5,403,482	3,632	0	5,330,612	15	0
		15	5,452,499	2,488	0	5,455,554	5	0
		16	5,844,467	2,318	0	5,543,060	9	0
		1	0	0	0	0	0	0
		2	760,052	3,566	0	739,018	19	0
		3	1,093,101	7,224	0	1,063,675	44	0
		4	106,342	11,684	0	155,626	68	0
		5	809,694	15,318	0	769,205	170	0
		6	1,450,282	17,556	0	1,429,460	130	0
		7	2,040,076	21,787	0	2,003,567	165	0
Sequential writes	10	8	2,560,720	25,564	0	2,439,702	195	0
		9	2,847,879	34,076	0	2,800,605	278	0
		10	3,321,462	44,960	0	3,167,860	357	0
		11	3,257,326	47,184	0	3,708,773	491	0
		12	3,896,505	72,208	0	4,010,059	499	0
		13	4,095,354	75,569	0	4,142,839	515	0
		14	4,303,702	83,929	0	4,368,953	533	0
		15	4,431,367	76,791	0	4,216,048	1,293	0
		16	4,305,791	115,325	0	4,231,973	1,109	0

		1	0	0	0	0	0	0
		2	10,486	13,694	0	9,276	217	0
		3	226,506	41,648	0	215,925	428	0
		4	30,579	72,038	0	22,921	712	0
		5	152,941	87,625	0	155,879	893	0
		6	298,085	92,745	0	292,340	789	0
		7	424,228	110,625	0	433,054	899	0
	100	8	580,164	152,609	0	561,530	1,316	0
		9	700,993	250,689	0	677,116	1,220	0
		10	772,338	393,164	0	758,865	2,788	0
		11	920,734	443,800	0	912,110	3,277	0
		12	1,146,131	640,228	0	1,028,514	4,948	0
		13	823,586	727,952	0	870,656	6,551	0
		14	1,293,867	720,587	0	1,223,505	8,592	0
		15	1,476,060	670,550	0	1,435,192	8,001	0
		16	1,453,444	618,926	0	1,611,690	7,081	0
		1	0	0	0	0	0	0
		2	1,141,970	224	18	1,193,432	2	0
		3	1,554,494	466	130	1,564,804	2	0
		4	137,831	1,809	251	148,079	6	0
		5	1,287,360	2,372	461	1,318,034	24	0
		6	2,200,018	2,743	648	2,163,919	29	0
		7	2,904,627	2,717	707	3,041,322	26	0
	1	8	3,590,092	3,186	1,075	3,572,417	25	0
		9	3,915,724	3,469	1,092	3,978,713	27	0
		10	4,290,581	3,523	1,362	4,305,103	37	0
		11	4,657,306	4,191	1,463	4,544,872	28	0
		12	4,889,613	4,802	1,720	4,676,063	37	0
		13	4,913,999	5,100	1,724	4,673,940	36	0
		14	4,926,720	37,048	2,056	5,060,792	41	0
		15	5,150,946	6,000	2,070	5,282,921	70	0
		16	5,380,476	56,366,125	2,368	5,580,536	80	0
		1	0	0	0	0	0	0
		2	494,316	16,495	1,571	498,764	117	1
		3	848,140	26,790	3,177	851,541	258	2
		4	139,539	37,747	5,177	324,062	294	4
		5	506,458	52,278	7,586	647,269	557	5
		6	1,057,519	57,925	10,429	1,028,398	538	2
		7	1,413,232	36,642	12,759	1,415,451	309	6
Reads-writes, 1:1	10	8	1,846,076	64,627	15,311	1,856,413	431	9
		9	2,235,933	74,050	18,174	2,183,579	446	21
		10	2,356,377	74,938	20,861	2,327,950	731	8
		11	2,894,605	87,864	22,918	2,546,487	86,386,150	19
		12	3,150,811	96,937	25,670	3,308,247	880	35
		13	3,321,340	104,387	27,843	3,337,866	1,102	28
		14	3,542,068	111,673	30,288	3,238,058	1,035	42
		15	4,039,059	124,404	32,642	3,786,180	1,144	36
		16	4,072,205	132,186	35,348	3,970,927	90,755,868	55
		1	0	0	0	0	0	0
		2	4,635	191,277	17,650	4,417	2,140	81
		3	36,012	408,700	32,336	34,037	4,853	188
		4	8,939	541,126	45,129	24,567	8,108	400
		5	86,611	445,513	56,285	85,483	8,389	632
		6	166,649	401,778	60,943	172,449	5,256	714
		7	253,490	673,402	66,066	254,966	4,840	994
	100	8	330,608	517,132	68,203	340,776	5,414	1,214
		9	413,582	671,651	64,568,606	431,853	10,018	1,533
		10	472,086	693,287	76,705	484,358	9,911	1,923
		11	554,669	811,423	82,741	517,400	11,985	2,060
		12	495,162	877,934	86,282	658,021	13,741	2,436
		13	726,835	907,001	91,059	732,701	13,885	2,496
		14	827,757	972,041	94,787	679,256	16,601	2,729
		15	698,791	1,055,213	98,396	765,159	17,155	3,214
		16	835,568	1,118,883	101,292	901,439	18,298	3,496

		1	0	0	0	0	0	0
		2	1,072,104	332	59	1,125,968	4	0
		3	1,480,159	1,308	226	1,525,444	1,008	0
		4	107,882	2,036	457	123,212	21	0
		5	1,209,490	3,182	820	1,239,872	21	0
		6	2,156,045	2,423	1,112	2,186,030	24	0
		7	2,844,787	4,231	1,440	2,811,662	31	0
	1	8	3,453,401	44,008,400	1,678	3,260,134	26	0
		9	3,719,433	4,628	2,087	3,891,614	30	0
		10	4,149,597	4,704	84,138,916	4,114,232	49	0
		11	4,236,925	6,078	2,664	4,329,414	99	0
		12	4,797,115	6,803	3,063	4,526,456	56	0
		13	4,760,991	6,986	3,300	4,627,566	71	0
		14	4,868,593	7,022	3,445	4,864,668	52	0
		15	5,012,035	65,493,604	3,739	5,062,356	53	0
		16	5,643,145	10,827	4,143	5,233,394	74	1
		1	0	0	0	0	0	0
		2	94,324	19,229	1,959	196,363	185	1
		3	627,562	46,107	4,328	671,225	383	2
		4	53,321	61,111	7,030	73,165	561	5
		5	456,043	58,320	10,021	445,405	923	14
		6	843,340	62,133	12,789	833,500	527	17
		7	1,207,245	84,353	15,776	1,171,026	495	29
Reads-writes, 2:1	10	8	1,434,600	61,094	18,811	1,560,044	583	23
		9	1,803,746	111,295	22,488	1,758,513	1,011	48
		10	2,040,950	132,286	25,221	1,968,687	1,307	68
		11	2,178,019	155,061	29,286	1,991,989	1,474	73
		12	2,555,733	172,010	31,943	2,618,062	1,509	90
		13	2,924,076	183,390	35,229	2,754,254	1,697	75,587,770
		14	2,767,402	209,074	37,878	2,869,926	1,963	99
		15	3,222,874	230,981	40,931	2,947,181	2,365	107
		16	3,553,923	76,094,616	43,921	3,260,537	2,341	144
		1	0	0	0	0	0	0
		2	3,505	290,114	17,662	2,812	3,058	133
		3	17,593	543,198	36,221	112,653	9,090	290
		4	4,838	638,420	49,361	19,812	14,038	501
		5	103,757	565,752	60,059	61,438	11,599	832
		6	119,770	441,789	64,958	121,237	8,593	921
		7	172,552	449,690	63,296	178,772	12,368	1,157
		8	239,339	481,840	67,110	238,132	14,352	1,509
	100	9	290,620	865,749	70,873	302,119	14,610	1,749
		10	358,802	879,304	71,100	353,623	20,215	2,073
		11	366,251	949,060	76,001	332,818	22,683	74,610,165
		12	468,589	1,061,951	78,638	477,181	24,926	2,750
		13	510,885	1,120,637	82,043	489,945	90,165,552	3,026
		14	601,938	1,173,677	83,699	476,728	55,277	3,287
		15	582,573	1,207,046	86,548	574,289	30,838	3,559
		16	767,207	1,243,436	88,128	642,998	34,839	3,880
		1	0	0	0	0	0	0
		2	991,631	781	117	1,006,129	9	0
		3	1,548,699	1,836	426	1,377,042	18	0
		4	134,902	3,001	839	152,237	36	0
		5	1,116,576	2,800	1,376	1,013,564	30	0
		6	1,877,219	2,967	1,789	1,847,694	41	1
		7	2,547,937	3,870	1,837	2,507,082	30	0
		8	3,153,324	3,498	1,991	3,271,212	58	0
	1	9	3,585,995	5,662	2,469	3,526,555	42	1
		10	3,873,651	11,793	3,175	3,987,872	64	0
		11	4,365,726	6,521	3,272	4,147,710	53	0
		12	4,263,009	12,901	3,434	4,426,116	69	1
		13	44,81,066	8,318	3,825	4,364,179	50	2
		14	4,831,710	9,102	5,465	4,421,809	67	0
		15	4,509,130	11,082	5,883	5,743,770	85	0
		16	4,880,687	12,481	57,360,977	4,914,192	120	0

		1	0	0	0	0	0	0
		2	20,278	30,506	2,433	33,540	186	4
		3	495,989	61,564	6,002	490,576	663	11
		4	52,116	82,611	9,244	133,110	697	14
		5	311,945	86,376	12,413	379,712	768	24
		6	603,665	77,466	15,463	593,171	1,019	39
		7	867,547	121,110	18,781	853,661	879	64
Reads-writes, 4:1	10	8	1,048,296	132,573	21,524	1,038,843	809	84
		9	1,339,772	171,436	24,719	1,372,081	1,832	116
		10	1,553,177	181,303	27,317	1,573,214	1,691	119
		11	1,861,970	205,858	31,450	1,814,199	2,126	148
		12	2,091,376	220,778	35,386	1,921,108	2,344	174
		13	2,211,813	247,902	39,058	1,962,985	7,917	210
		14	2,437,334	285,992	41,909	2,090,061	2,787	245
		15	2,498,094	312,268	44,510	2,172,748	3,155	239
		16	2,497,470	326,432	47,879	2,563,169	3,148	313
		1	0	0	0	0	0	0
		2	1,012	306,540	24,716	1,522	3,398	170
		3	30,541	641,681	31,884	3,144	12,325	338
		4	5,612	802,210	43,259	3,924	16,231	520
		5	36,947	466,368	49,883	3,9146	23,220	803
		6	73,381	485,848	52,669	76,446	14,367	1,111
		7	111,608	428,747	54,475	115,571	19,448	1,421
	100	8	152,477	646,615	51,291	150,945	13,181	1,800
		9	185,423	756,044	53,291	205,889	22,161	1,968
		10	228,027	768,633	56,245	231,334	30,845	2,272
		11	246,155	886,960	59,178	272,884	32,050	2,608
		12	307,383	934,826	60,071	300,768	37,186	2,930
		13	322,243	1,003,733	64,072	292,474	36,015	3,035
		14	369,113	1,013,108	66,202	359,984	39,848	3,527
		15	347,137	1,102,995	65,293	404,955	43,640	3,678
		16	418,867	1,158,639	67,509	406,499	47,950	4,025
		1	0	0	0	0	0	0
		2	763,424	949	231	845,909	7	0
		3	1,357,146	2,377	740	1,208,277	25	0
		4	95,619	3,080	1,217	126,322	37	1
		5	947,901	3,639	1,148	900,177	26	0
		6	1,593,778	3,398	1,590	1,545,514	31	0
		7	2,172,170	3,176	1,912	2,222,696	35	1
	1	8	2,711,946	5,143	2,410	2,705,954	34	1
		9	3,117,777	5,823	3,053	2,930,588	68	1
		10	3,522,296	15,717	3,219	3380738	72	0
		11	3,735,396	9,566	3,762	3,763,342	65	2
		12	4,152,194	8,939	4,126	4,271,909	86	2
		13	4,334,197	12,481	4,510	4,278,053	119	5
		14	4,243,699	10,715	45,284,599	4,222,797	133	4
		15	4,409,051	13,864	5,175	4,489,961	114	1
		16	4,733,603	12,069	5,761	4,196,276	114	4
		1	0	0	0	0	0	0
		2	11,497	35,014	3,710	30,549	228	4
		3	300,438	72,518	7,395	314,285	650	21
		4	39,108	99,168	12,294	72,558	1,146	44
		5	205,055	159,919	16,099	210,522	1,464	88
		6	384,469	129,869	18,529	373,537	1,125	71
		7	521,877	105,189	21,939	560,399	1,761	132
Reads-writes, 8:1	10	8	672,190	93,843	25,524	728,010	1,685	130
		9	903,911	182,365	28,305	873,575	2,018	202
		10	968,262	222,241	31,085	1,054,995	2,200	202
		11	1,122,084	225,294	33,637	1,219,399	2,307	281
		12	1,325,120	263,973	37,327	1,352,856	2,753	279
		13	1,505,480	295,584	39,939	1,214,906	2,959	335
		14	1,659,500	362,594	43,539	1,058,749	3,690	282
		15	1,804,523	368,546	46,111	1,602,766	3,814	329
		16	1,465,097	381,548	49,092	1,857,269	4,045	402

	1	0	0	0	0	0	0
	2	809	206,134	18,497	705	6,506	194
	3	8,526	468,305	30,940	10,425	11,218	338
	4	4,435	495,287	31,957	3,080	21,015	626
	5	19,692	721,504	36,798	23,215	18,863	926
	6	37,243	611,243	37,511	46,430	28,783	1,190
	7	59,520	320,852	32,342	68,693	14,482	1,386
	8	83,359	341,355	33,809	93,100	15,908	1,728
100	9	100,857	687,560	33,604	117,363	28,642	1,978
	10	204,083	680,063	38,981	136,900	35,322	2,179
	11	155,733	578,688	44,615	160,573	33,480	2,369
	12	164,909	812,030	38,590	179,731	41,361	2,669
	13	194,939	941,114	56,260	192,363	84,560,515	2,802
	14	227,543	959,357	60,129	257,009	47,300	3,055
	15	215,458	1,254,096	59,685	279,151	51,969	3,212
	16	245,468	950,033	57,642	274,801	56,002	3,377

Table 6.3: The numbers of aborts that have been observed during the evaluation. Each number refers to a complete test with a run time of one minute. The allocator tests do not generate conflicts; so they are left out here. The test of Figure 6.9, where a modified tree setup was used, only contains non-conflicting writes, so it is left out as well. With several tens of millions some of the values are extremely high. These aborts can be attributed to the implementation of TinySTM: The way in which memory locations are mapped to locks can result in false sharing of locks among independent transactions: If stack memory of two transactions is mapped to the same internal locks, the transactions observe conflicts, even though their read and write sets do not overlap.

7 Future Work

This chapter presents some ideas on how to improve the design and implementation of the presented work.

7.1 Performance and Scalability

There are some possible changes to design and implementation that might result in a higher overall performance and scalability.

7.1.1 Offset-independent scatter-gather I/O

When instructing a component to apply or undo an event, the framework's current implementation passes successive entries in the history that were inserted by the same component in one call. This results in better cache locality, less function-call overhead, and allows the I/O component to merge successive write operations at consecutive offsets to one single, large write.

The calls `readv` and `writev`, known as scatter-gather I/O, allow for the reading and writing at various file offsets with one single call. This is a companion feature to the `read` and `write` calls. Instead of supplying a single buffer and length argument, a vector of these values is passed. The calls themselves work as if each entry in the vector had been executed individually.

Linux kernels since version 2.6.30 and some BSD systems support offset-independent variants of scatter-gather I/O, called `preadv` and `pwritev`. These interfaces work similar to `readv` and `writev`, but each vector entry has an extra field, which holds the file offset for the operation.

This offset-independent scatter-gather I/O can be used to merge the apply phase of write operations, even if the writes are not performed at consecutive offsets. The transaction then only calls `preadv` once, instead of `pread` for each individual write operation in the history.

There is indication that this could result in significant performance improvements. A single write operation of 24 byte was measured with approximately two thousand five hundred cycles. For 10 calls to apply, this results in an overhead of around twenty five thousand cycles. If the framework is able to perform write merging, the 10 writes only take eight thousand five hundred cycles to apply; an improvement of factor three. This can also be observed by comparing Figures 6.8 and 6.11. The throughput for sequential writes is two to three times higher than for random writes. If the use of offset-independent scatter-gather I/O leads to similar results in the general case, it might be possible to hide some of the framework's overhead behind it.

7.1.2 Irrevocability

A discussion about mixing revocable and irrevocable transactions can be found in [SMS08]. Therein Spear, Michael, and Scott present different approaches for parallel execution of revocable transactions while an irrevocable transaction is present.

Similar approaches could be applied to the framework. Currently its implementation switches to serial mode whenever a transaction becomes irrevocable. This is unnecessary in some situations. Often it is preferable to run one transaction irrevocable, but allow for other, unrelated transactions to execute concurrently.

An improvement to irrevocability would be to first switch to a less restrictive irrevocable-but-concurrent mode, where concurrent transactions are allowed. The transaction running irrevocably can then still be switched to serial mode if necessary.

The framework's implementation is not yet prepared to handle this. The most important problem to solve is the abort of transaction without their assistance. At the moment, when a transaction detects a conflict, it frees its resources and aborts itself. This works well because each transaction is aware of its resources and can release them. In the case of on an irrevocable-but-concurrent mode, when the irrevocable transaction detects a conflict it does not abort itself, but the transaction it conflicts with. This can lead to resource leaks if the aborted transaction is not prepared; like when it is in the process of acquiring a resource, but not yet finished. Care has to be taken to prevent this from happening.

7.1.3 Contention management

Having many concurrent transactions likely results in a large number of aborts if the transactions conflict with each other. If a transaction detects a conflict it aborts itself, even though it might be better to abort the other transaction. In bad cases the resulting transactional application has a lower throughput than a coarse-grainly locked one, even if the framework's internals scale well.

To minimize aborts and optimize towards particular workloads during runtime, a *contention manager* could be provided with the framework. This component is called whenever two transactions conflict. It then decides which of the transactions wins the conflict. The contention manager can take additional parameters into account; such as transaction length, size of the read and write sets, or the number of previous aborts. As an example, the contention manager might prefer a transaction with many file reads over a transaction that mostly executed deferred file writes.

The framework does not yet contain a contention manager. What is missing, besides the contention manager itself, is again a way to abort a transaction without its assistance, or a way to signal the transaction to abort itself. In a simple implementation, a flag variable could be used to mark a transactions as aborted. It is set by the contention manager and checked frequently by the transactions. As soon as the losing transaction aborted itself, the winning transaction can continue.

7.1.4 Transaction scheduling

Currently, transaction scheduling relies completely on the operating system's thread scheduler, which schedules *aggressively*: a thread is scheduled if it is ready. This strategy contradicts an observation made in Chapter 6: certain workloads scale up to a number of transactions, but then their throughput stalls or decreases due to conflicts. As a solution, the previously mentioned contention manager, once in place, could be extended and be part of a larger transaction scheduling system. This scheduler could use a *conservative* approach: it starts transactions until an optimum throughput is reached, but not more. In the case of conflicts it could stop transactions instead of aborting them completely. For example, in the case that a transaction holds a reader lock, a possible writer does not have to be aborted, but could just wait until the reader committed.

With the current implementation it is not possible to have transactions waiting for each other, as this might result in deadlocks. With the transaction scheduler, this feature is well imaginable. The irrevocable-but-concurrent mode, described in the previous section, is just a special case of this scenario and trivially supportable by the scheduler.

There are two possible implementations of transaction scheduling: a kernel-based approach and an application-based approach. For the kernel-based approach the transactional memory system has to make information abort active transactions available to the operating system. Some pages of shared memory between application and scheduler should be sufficient. The transaction scheduler would extend or replace the standard kernel thread scheduler to take the additional information into account when scheduling threads with active transactions. The approach is not portable, but enables good scheduling results.

The application-based transaction scheduler does not need modifications in the kernel. The kernels thread scheduler only has to be informed which threads to schedule. A locking protocol between transaction and transaction scheduler allows this. Whenever the transaction is blocked on a lock, it is not scheduled by the thread scheduler. When the transactional memory system decides to schedule a transaction it releases the respective lock, which unblocks the transaction. The application-based approach is portable among different systems but likely yields worse results than the kernel-based approach; especially if the strategies of the transaction scheduler and the underlying thread scheduler are not compatible.

7.2 Transactional File-System Support

The current support of file-system operations works well enough if application developers can live with the semantics provided by POSIX. However, it might surprise some programmers that operations on the file system are not executing in isolation. For example, without support from the operating-system kernel a file can be removed by an external process while it is in use by a transaction.

A likely assumption of the application programmer might instead be that the state of the file system, as seen by a transaction, is consistent with the transaction's operations. The transactional memory system could therefore try to provide each transaction with its own view on the file system and resolve conflicts before committing any updates. This section provides some ideas of how to achieve this.

7.2.1 Snapshots

Snapshot support is hardly achievable from within the transactional memory system, but several possible solutions are imaginable. For example, the file system itself could provide such a facility. Modern file systems, such as ZFS and Btrfs, provide means to create snapshots of the file system's state. It might be possible to use or extend these facilities to provide a transaction-local view on the file system.

The transactional memory system retrieves a snapshot of the file system before the first file-related external action is executed. All further file operations are invoked on this snapshot. During validation the transactional memory system instructs the file system to search for conflicts when reintegrating the altered snapshot. Depending on the result, it then decides on how to proceed. Irrevocable transactions can be supported by reintegrating the snapshot before becoming irrevocable and afterwards working on the file system's master copy directly.

7.2.2 Version control systems

A similar solution could use a version control system as file system. It is possible to mount repository's of most popular version control systems like normal file systems. The mount directory then contains the local view on the repository's data.

A transactional memory system would present each transaction with an individual copy or a branch of the repository. On commit the local copy is reintegrated into the repository's master branch and any conflicts are detected by the version control system. In the case of a conflict, the transactional memory system would roll back the transaction and checkout a new copy of the master branch.

This solution might work even better than the snapshot mechanism presented above: conflict detection and resolution is the core function of every version control system and not just an extra feature. Also, this completely operates in user space, which makes it more portable than a real file system.

7.2.3 Union mounts

A third solution could be the use of union mounts. Normally, when mounting several file systems at the same path only the last mounted file system is visible. Union mounts provide a means to make them all visible at the same time. The content of earlier mounted file system is visible as long as it is not overlayed by content of a later mounted file system.

This allows the transactional memory system to provide each transaction with its own view on the file system. For each transaction it mounts a transaction-local, write-able file system over a read-only global file system such that all the transaction's operations execute within the local file system. For committing a transaction's updates, some central service is required to coordinate among the participating transactions and the rest of the system.

7.2.4 Problems

While the presented ideas could certainly be implemented in most cases, it is questionable whether the results are always useful to the application programmer. Providing local views on the file-system state might differ from POSIX semantics in subtle ways and therefore confuse the programmer or break the application. For example, the file system is used for various forms of inter-process communication. Providing each transaction with a local view might be completely against the programmer's intention. Assume a transaction creates a FIFO and waits for someone to connect. As the FIFO only exists in the transaction's local file-system view no process has the chance of connecting. The transaction would either be blocked in a deadlock by waiting forever; or a livelock by timing out, and retrying again and again.

7.3 Error Handling

Commit-time error handling can be confusing to the application developer: There is a separation of a call's invocation, which happens from within the transaction, and the occurrence of the error, which happens at commit time.

In most situations it might be better to prevent commit-time errors from happening in the first place. An enhancement to the framework could automatically predict a commit's chance to be successful before the commit even starts. For example, the framework could pre-allocate all necessary disk space before applying any write actions. If one of the pre-allocations fails, it is possible to revoke previously done pre-allocations without problems. The transaction can afterwards abort and provide an error code on the first write operation that is executed during its retry. For this to work in the general case, one has to examine all error codes and their underlying cause, whether the error can be predicted.

A more complicated approach is to completely trace the execution of a transaction. When an error occurs during the commit's pre-allocation step, the state is rolled back to the point where the error would have occurred in non-transactional code and an error is reported to the transaction.

The obvious problem is the performance of this approach. Tracing an application during its execution makes the application noticeably slower. A possible solution is to make a single snapshot at the transaction's start and let it first execute without tracing. If an error occurs during the commit phase, the transaction is set back to its snapshot state and executed a second time with tracing enabled.

Another problem is the relevance of the error during the retry. When retrying the transaction and injecting an error, it is not guaranteed whether the error would occur in the non-transactional execution. Such false positives have to be checked for as well.

The major problem with all variants of rollback is that once an action has been applied, it is very hard to revoke its effects. Think of a write to a FIFO: once it is applied, it cannot be undone anymore. However, if these problems could be solved in a useful way, this would provide *failure atomicity* for the transaction's actions. Then a transaction is guaranteed to leave the system in a consistent state, even in the case of an error.

8 Conclusion

A framework for executing external actions within memory transactions has been presented. It strives to fulfill the requirements of correctness, standards compliance, extensibility, and performance.

The design is based on domains and actions, which abstract the system's resources and function calls. Each domain provides information hiding, and independent consistency constraints for its resource. Each action allows its underlying function to either be deferred until the commit happens, compensated during the transactions aborts, or make the transaction irrevocable. To handle commit-time errors a simple mechanism is provided that allows the application programmer to specify error handlers for individual actions.

Domains and actions are grouped into components. Components provide a generic public interface, such that new domains and actions can be added without changing the framework's internals.

The framework's implementation provides transactional versions of the most important parts of the POSIX programming interface. This includes memory allocation, file-descriptor I/O, socket I/O, basic file-system support, and many simple functions.

The framework has been evaluated with respect to the specified requirements. Correctness; namely atomicity, consistency, and isolation; can be achieved for process-local resources, which are under the control of the transactional memory system. The state of external resources, such as the file system, cannot be isolated completely and therefore cannot be handled reliably by the transactional memory system. Standards compliance has been mostly achieved. Some incompatibilities with non-transactional POSIX systems remain. These are related to limitations of the platform or the semantics of some function calls. Extensibility has been achieved by the use of components. The requirement of performance has been achieved partially. The throughput of memory allocation does not scale with the number of processors, but this is unrelated to the framework. The file-descriptor I/O throughput varies with the parameters, such as file size, transaction length, or read-write ratio. It is hampered by serious overhead in the single-threaded case, but can outperform non-transactional code in some multi-threaded scenarios. For pessimistic concurrency control, file-descriptor I/O throughput typically scales up to approximately four to six concurrent transactions, and afterwards stalls or decreases. Optimistic concurrency control currently does not seem to provide any benefits. It rarely outperforms the pessimistic strategy. Also, shortcomings in the implementation of Linux' virtual file-system switch currently prevent good scalability for real applications.

Several ways of improving the framework have been identified. Performance improvements could possibly be achieved by the amount of low-level commit operations, and using an optimized transaction scheduling. The problem with isolation in the file system might be solved with file-system implementations that support isolation natively, such as file-system snapshots, version control systems, or union mounts. A simplified error handling that more resembles the non-transactional case might be possible with an extensive tracing of transactions.

Currently unknown are real-world workloads of production systems. As a near-term goal, Linux' file-system implementation should be changed to allow scalable, concurrent access to file buffers; and some real applications should be converted to use transactional memory, which would allow to gather information about realistic workloads.

A Transactional Functions of the POSIX Standard

This chapter lists some important functions of the POSIX specification, together with a short note on how they are supported by Taglibc.

A.1 assert.h

Interface	Class	Remarks
`assert`	stateless	This macro is typically resolved to a conditional `abort` or some internal interface.

Table A.1: Interfaces in `assert.h`.

A.2 complex.h

Interface	Class	Interface	Class	Interface	Class
`cacosf`	stateless	`ccosf`	stateless	`csinhf`	stateless
`cacoshf`	stateless	`ccoshf`	stateless	`csinhl`	stateless
`cacoshl`	stateless	`ccoshl`	stateless	`csinh`	stateless
`cacosh`	stateless	`ccosh`	stateless	`csinh`	stateless
`cacosl`	stateless	`ccosl`	stateless	`csinl`	stateless
`cacos`	stateless	`ccos`	stateless	`csin`	stateless
`casinhf`	stateless	`cexpf`	stateless	`csqrtf`	stateless
`casinhl`	stateless	`cexpl`	stateless	`csqrtl`	stateless
`casinl`	stateless	`cexp`	stateless	`csqrt`	stateless
`casin`	stateless	`clogf`	stateless	`cssinf`	stateless
`catanf`	stateless	`clogl`	stateless	`ctanf`	stateless
`catanhf`	stateless	`clog`	stateless	`ctanhf`	stateless
`catanhl`	stateless	`cpowf`	stateless	`ctanhl`	stateless
`catanh`	stateless	`cpowl`	stateless	`ctanh`	stateless
`catanl`	stateless	`cpow`	stateless	`ctanl`	stateless
`catan`	stateless	`csinf`	stateless	`ctan`	stateless

Table A.2: Interfaces in `complex.h`.

A.3 errno.h

Interface	Class	Remarks
`errno`	stateful	In multi-threaded programs, `errno` is typically implemented as a macro around an internal interface. The use of `errno` is so common that is makes sense to handle its state in the transactional memory system.

Table A.3: Interfaces in `errno.h`.

A.4 fcntl.h

Interface	Class	Remarks
`creat`	unprotected	
`fcntl`	unprotected or real	Reading a file property is an unprotected action. Changing a property makes it a real action.
`open`	unprotected	POSIX lacks the necessary capabilities to fully implement `open` in a transaction-safe fashion.

Table A.4: Interfaces in `fcntl.h`.

A.5 math.h

Interface	Class	Interface	Class	Interface	Class
`acosf`	stateless	`erff`	stateless	`logbf`	stateless
`acoshf`	stateless	`erfl`	stateless	`logbf`	stateless
`acoshl`	stateless	`erf`	stateless	`logbl`	stateless
`acosh`	stateless	`exp2f`	stateless	`logbl`	stateless
`acosl`	stateless	`exp2l`	stateless	`logb`	stateless
`acos`	stateless	`exp2`	stateless	`logb`	stateless
`asinf`	stateless	`expf`	stateless	`logf`	stateless
`asinhf`	stateless	`expl`	stateless	`logl`	stateless
`asinhl`	stateless	`expm1f`	stateless	`log`	stateless
`asinh`	stateless	`expm1l`	stateless	`powf`	stateless
`asinl`	stateless	`expm1`	stateless	`powl`	stateless
`asin`	stateless	`exp`	stateless	`pow`	stateless
`atan2f`	stateless	`hypotf`	stateless	`sinf`	stateless
`atan2l`	stateless	`hypotl`	stateless	`sinhf`	stateless
`atan2`	stateless	`hypot`	stateless	`sinhl`	stateless
`atanf`	stateless	`ilogbf`	stateless	`sinh`	stateless
`atanhf`	stateless	`ilogbl`	stateless	`sinl`	stateless
`atanhl`	stateless	`ilogb`	stateless	`sin`	stateless
`atanh`	stateless	`j0`	stateless	`sqrtf`	stateless
`atanl`	stateless	`j1`	stateless	`sqrtl`	stateless
`atan`	stateless	`jn`	stateless	`sqrt`	stateless
`cbrtf`	stateless	`lgammaf`	stateful	`tanf`	stateless
`cbrtl`	stateless	`lgammal`	stateful	`tanhf`	stateless
`cbrt`	stateless	`lgamma`	stateful	`tanhl`	stateless
`cosf`	stateless	`log10f`	stateless	`tanh`	stateless
`coshf`	stateless	`log10l`	stateless	`tanl`	stateless
`coshl`	stateless	`log10`	stateless	`tan`	stateless
`cosh`	stateless	`log1pf`	stateless	`tgammaf`	stateless
`cosl`	stateless	`log1pl`	stateless	`tgammal`	stateless
`cos`	stateless	`log1p`	stateless	`tgamma`	stateless
`erfcf`	stateless	`log2f`	stateless	`y0`	stateless
`erfcl`	stateless	`log2l`	stateless	`y1`	stateless
`erfc`	stateless	`log2`	stateless	`yn`	stateless

Table A.5: Interfaces in `math.h`.

A.6 pthread.h

Interface	Class	Interface	Class
`pthread_attr_init`	real	`pthread_create`	real
`pthread_attr_destroy`	real	`pthread_detach`	real
`pthread_attr_getdetachstate`	real	`pthread_equal`	stateless
`pthread_attr_getschedpolicy`	real	`pthread_exit`	real
`pthread_attr_getschedparam`	real	`pthread_join`	real
`pthread_attr_getinheritsched`	real	`pthread_once`	real
`pthread_attr_getscope`	real	`pthread_self`	stateless
`pthread_attr_setdetachstate`	real		
`pthread_attr_setschedpolicy`	real		
`pthread_attr_setschedparam`	real		
`pthread_attr_setinheritsched`	real		
`pthread_attr_setscope`	real		

Table A.6: Interfaces in `pthread.h`.

A.7 sched.h

Interface	Class
`sched_yield`	stateless

Table A.7: Interfaces in `sched.h`.

A.8 stdio.h

Interface	Class	Remarks
`rename`	real	A call to `rename` has to commit all previous writes to the file to behave like its non-transactional counterpart.

Table A.8: Interfaces in `fcntl.h`.

A.9 stdlib.h

Interface	Class	Interface	Class
`abort`	pure	`qsort`	stateful
`calloc`	unprotected	`rand_r`	stateful
`free`	unprotected	`rand`	unprotected
`malloc`	unprotected	`realloc`	unprotected
`mkdtemp`	stateful	`srand`	unprotected
`mkstemp`	unprotected	`system`	real
`posix_memalign`	unprotected		

Table A.9: Interfaces in `stdlib.h`.

A.10 string.h

Interface	Class	Interface	Class	Interface	Class
`memccpy`	stateful	`memcpy`	stateful	`strcase`	stateful
`memchr`	stateful	`memmove`	stateful	`strcpy`	stateful
`memcmp`	stateful	`memset`	stateful	`strlen`	stateful

Table A.10: Interfaces in `string.h`.

A.11 sys/select.h

Interface	Class
`select`	stateful

Table A.11: Interfaces in `sys/select.h`.

A.12 sys/socket.h

Interface	Class	Remarks
`accept`	real	
`bind`	real	
`connect`	real	
`listen`	real	
`send`	unprotected or real	Similar to `write`
`shutdown`	unprotected or real	Similar to `close`
`socket`	unprotected or real	Similar to `open`
`recv`	unprotected or real	Similar to `read`

Table A.12: Interfaces in `sys/socket.h`.

A.13 sys/stat.h

Interface	Class	Interface	Class	Interface	Class
`chmod`	stateful	`lstat`	stateful	`mknod`	stateful
`fchmod`	stateless	`mkdir`	stateful	`stat`	stateful
`fstat`	stateful	`mkfifo`	stateful	`umask`	real

Table A.13: Interfaces in `sys/stat.h`.

A.14 unistd.h

Interface	Class	Remarks
`chdir`	unprotected	
`close`	unprotected	
`dup`	unprotected	
`dup2`	real	This call creates and closes a file descriptor at the same time.
`fchdir`	unprotected	
`fsync`	unprotected	
`getcwd`	unprotected	
`link`	unprotected	
`lseek`	unprotected	
`pipe`	unprotected	Similar to `open`.
`pread`	unprotected	
`pwrite`	unprotected	
`read`	unprotected	
`sleep`	stateless	
`sync`	unprotected	
`unlink`	unprotected	
`write`	unprotected	

Table A.14: Interfaces in `unistd.h`.

Bibliography

[BH03] Ray Bryant and John Hawkes. Linux scalability for large NUMA systems. In *Proceedings of the Linux Symposium*, 2003. http://oss.sgi.com/projects/numa/Linux_Scalability_for_Large_NUMA_OLS2003.pdf.

[BHG87] Philip Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems.* Addison Wesley Publishing Company, 1987.

[BLM05] Colin Blundell, E. Lewis, and Milo Martin. Deconstructing transactional semantics: the subtleties of atomicity. 2005. http://www.cis.upenn.edu/acg/papers/wddd05_atomic_semantics.pdf.

[BLM06a] Colin Blundell, E. Lewis, and Milo Martin. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5(2):17, 2006. http://www.cis.upenn.edu/acg/papers/cal06_atomic_semantics.pdf.

[BLM06b] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Unrestricted transactional memory: supporting I/O and system calls within transactions. Technical report, University of Pennsylvania, May 2006. http://www.cis.upenn.edu/~milom/papers/tr06_unrestricted_transactions.pdf.

[BZ07] Lee Baugh and Craig Zilles. An analysis of I/O and syscalls in critical sections and their implications for transactional memory, 2007. http://www.cs.rochester.edu/meetings/TRANSACT07/papers/baugh.pdf.

[Dre09] Ulrich Drepper. Defensive programming for Red Hat Enterprise Linux (and what to do if something goes wrong), 2009. http://people.redhat.com/drepper/defprogramming.pdf.

[FFM$^+$07] Pascal Felber, Christof Fetzer, Ulrich Müller, Torvald Riegel, Martin Süßkraut, and Heiko Sturzrehm. Transactifying applications using an open compiler framework, 2007. http://wwwse.inf.tu-dresden.de/papers/preprint-felber2007tanger.pdf.

[FFR08] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008. http://wwwse.inf.tu-dresden.de/papers/preprint-felber2008tinystm.pdf.

[GNU] GNU C Library. http://www.gnu.org/software/libc/.

[GR93] Jim Gray and Andreas Reuter. *Transaction processing: concepts and techniques.* Morgen Kaufmann, 1993.

[Gra81] Jim Gray. The transaction concept: virtues and limitations. In *Proceedings of the 7th International Conference on Very Large Databases.*, 1981. http://research.microsoft.com/~gray/papers/theTransactionConcept.pdf.

[Har04] Tim Harris. Exceptions and side-effects in atomic blocks, 2004. http://research.microsoft.com/en-us/um/people/tharris/papers/2004-csjp.pdf.

[HK08] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects, 2008. http://www.cs.brown.edu/~ejk/papers/boosting-ppopp08.pdf.

[HR83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. In *ACM Computing Surveys 15*, 1983. http://portal.acm.org/ft_gateway.cfm?id=291&type=pdf&coll=GUIDE&dl=GUIDE&CFID=18545439&CFTOKEN=99113095.

[HS08] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[ISO04] ISO/IEC. *ISO/IEC 9899 - Programming languages - C*, TR2 edition, 2004. http://www.open-std.org/jtc1/sc22/wg14/www/standards.

[LK08] James Larus and Christos Kozyrakis. Transactional memory. *Commun. ACM*, 51(7):80–88, 2008.

[LLV] The LLVM Compiler Infrastructure. http://llvm.org.

[Ope08] The Open Group. *The Open Group base specifications, issue 7*, 2008. http://www.opengroup.org/onlinepubs/9699919799/toc.htm.

[PHR+09] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *22nd ACM Symposium on Operating Systems Principles (SIGOPS)*, 2009. http://www.sigops.org/sosp/sosp09/papers/porter-sosp09.pdf.

[RWZ09] Torvald Riegel, Jons-Tobias Wamhoff, and Thomas Zimmermann. Supporting transactional execution of library code. Unpublished, 2009.

[SMS08] Michael Spear, Maged Michael, and Michael Scott. Inevitability mechanisms for software transactional memory, 2008. http://www.cs.rochester.edu/u/scott/papers/2008_TRANSACT_inevitability.pdf.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, 1995.

[Sut05] Herb Sutter. The free lunch is over: a fundamental turn toward concurrency in software, 2005. http://www.gotw.ca/publications/concurrency-ddj.htm.

[VGS08] Haris Volos, Neelam Goyal, and Michael Swift. Pathological interaction of locks with transactional memory, 2008. http://www.cs.wisc.edu/multifacet/papers/transact08_txlock.pdf.

[VTG+09] Haris Volos, Andres Jaan Tack, Neelam Goyal, Michael Swift, and Adam Welc. xCalls: safe I/O in memory transactions. In *Proceedings of EuroSys 2009*, 2009. http://www.cs.wisc.edu/multifacet/papers/eurosys09_xCalls.pdf.

[Whe03] David A. Wheeler. Secure programming for Linux and Unix HOWTO, 2003. http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO.pdf.

[WV01] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery.* Morgen Kaufmann, 2001.

[Yan84] Mihalis Yannakakis. Serializability by locking, 1984. http://portal.acm.org/ft_gateway.cfm?id=322425&type=pdf&coll=GUIDE&dl=GUIDE&CFID=50220664&CFTOKEN=40951514.

[ZGU$^+$09] Ferad Zyulkyarov, Vladimir Gajinov, Osman S. Unsal, Adrián Cristal, Eduard Ayguadé, Tim Harris, and Mateo Valero. Atomic quake: using transactional memory in an interactive multiplayer game server. *SIGPLAN Not.*, 44(4):25–34, 2009. http://research.microsoft.com/en-us/um/people/tharris/papers/2009-ppopp-quake.pdf.

Index