EUROPEAN COMMISSION
SEVENTH FRAMEWORK PROGRAMME
FP7-ICT-2011-C
Grant agreement no.: 308830

Deliverable D1.1

Final report for WP1: Identification of case studies

Project acronym	ADVENT
Project title	Architecture-driven verification of systems software
Funding scheme	FP7 FET Young Explorers
Scientific coordinator	Dr. Alexey Gotsman, IMDEA Software Institute, Alexey.Gotsman@imdea.org, +34 911 01 22 02

Abstract

Work Package 1 in the ADVENT project was concerned with selecting components and interfaces from real-world systems software that would motivate our development of architecture-driven verification methods. To achieve this goal, we have surveyed a number of real-world pieces of systems software and selected the ones that yield the most promising case studies, as per the criteria set in the project proposal. This document summarizes the systems and components we will consider as case studies in a range of domains: concurrent libraries, software transactional memory, operating system kernels, device drivers, eventually consistent replicated stores, weak memory models, compilers and run-time systems.

1 Selection Criteria

In our selection of case studies, we were guided by the following criteria, whose aim was to ensure that the effort of the project is focused upon delivering benefits that can be exploited by real-world applications in a practical software lifecycle structure.

In selecting the target systems, we took into account the following considerations:

- **The role within the existing software infrastructure.** We preferred to investigate widely used mainstream software systems over niche systems. This is motivated by our goal to achieve a maximal impact on the safety and dependability of our society's information infrastructure.
- **Public source code availability.** We focused on open-source implementations over proprietary code.
- **Multiple challenges.** We looked for systems from which we can select multiple case studies coming from different abstraction levels: inter-component interaction mechanisms (addressed in WP2), hardware interfaces (WP3), inter-language interfaces (WP4) and components (WP5).
- **Familiarity**. We considered the existing familiarity with certain application domains within the consortium and its End-User Panel as an asset when selecting the target systems.

In selecting components and interfaces, we employed the following criteria:

- **Core functionality.** The most critical and optimized parts of every system are the ones implementing its core functionality. Therefore, we preferred to consider the parts of the target systems that implement their such functionality.
- **Challenge.** We focused on studying mechanisms and components that are beyond the scope of the existing verification technologies, that the existing technologies handle poorly, or where the programming concepts themselves are poorly understood.
- **Cooperation of system developers.** We preferred to consider components whose developers are open to collaboration with scientific community.

It is likely that we will find more verification challenges during the rest of the project than the ones described in this document. In this case, such challenges considered as part of the work done in the work package to which they are relevant.

2 Concurrent Libraries

Concurrent libraries are collections of useful highly concurrent algorithms that have been packaged together. As with any library, these concurrent libraries try to hide the intricate implementation details that are needed to achieve good performance by exposing abstract interfaces to the programmers. They define concepts such as mutual exclusion locks, memory allocation, container classes, and iterators.

Looking at these libraries is useful for two reasons: (a) they form part of the core infrastructure of concurrent applications, and (b) their implementations contain intricate concurrent algorithms that are worthy of specification and verification. These two properties relate closely to the goals of WP2 (architecture-driven decomposition) and WP5 (architecture-driven verification of complex software components). We will need to formally capture the interfaces provided by such libraries, in terms of notions such as linearizability and abstraction (cf. Task 2.1), and also to find ways of verifying some of the more complex algorithms in these libraries, which lie beyond the current state-of-the-art techniques (cf. Tasks 5.1, 5.3, 5.4). Concurrent libraries will also be useful for WP3 (especially, Task 3.1) as they would be our primary source of low-level synchronization idioms that we will have to reason about in the relaxed memory context. They will also be directly relevant for WP6, as they will be the source of examples for evaluating the proposed verification tools.

We decided to focus on two such concurrent libraries, `java.util.concurrent` and Intel's Thread Building Blocks (TBB), as they are open-source, mature and widely used libraries and, moreover, they present two different ways of implementing concurrent algorithms. On the one hand, `java.util.concurrent` is written for a Java, a managed language with built-in memory management (garbage collection), whereas on the other hand, Intel's TBB targets C/C++ programmers which have to manage memory manually. In terms of algorithms implemented, `java.util.concurrent` is somewhat more advanced, but then TBB also has to deal with the thorny memory management issue.

2.1 java.util.concurrent

First, `java.util.concurrent` provides direct access to low-level atomic primitives of the JVM, by defining classes (`java.util.concurrent.atomic.*`) that implement atomic booleans, integers, long integers, references, arrays, etc. An interesting structure among these is `AtomicMarkableReference`, which represents a pair of a Boolean flag and a reference and provides atomic read and update operations on it.

Further, it implements a variety of locks and condition variables (`java.util.concurrent.locks.*`), as well as a collection of other simple synchronization primitives. These include count-down latches (`CountDownLatch`), barriers (`CyclicBarrier`) exchangers (`Exchanger`) counting semaphores (`Semaphore`), and phasers (`Phaser`). Of particular interest are *exchangers*, as they can be used as building blocks for improving the efficiency of other concurrent algorithms. For example, one can use them to turn Treiber's stack into the elimination-based HSY stack.

The library also contains a number of advanced concurrent implementations of standard container data types. These are listed below:

- Blocking single-ended FIFO queues:
  - `ArrayBlockingQueue`: a bounded array implementation,
  - `LinkedBlockingQueue`: a unbounded linked-list implementation,
  - `SynchronousQueue`: a single-element queue similar to a barrier or a rendezvous;
- Non-blocking single-ended FIFO queues:

  - `ConcurrentLinkedQueue`: an unbounded linked-list implementation,
  - `LinkedTransferQueue`: another unbounded linked-list implementation;
- Double-ended queues (useful for work stealing parallel task scheduling),
  - `ConcurrentLinkedDeque`: a non-blocking unbounded linked-list implementation,
  - `LinkedBlockingDeque`: a blocking linked list implementation;
- `PriorityBlockingQueue`: a priority queue, where items are attached priorities and the higher priority item is removed first;
- `DelayQueue`: a kind of priority queue, where each enqueued item is given a possibly different number of milliseconds to delay its visibility: items can be removed only after their delay period has expired;
- `ConcurrentSkipListSet`: a skip-list implementation of a set;
- `ConcurrentSkipListMap`: a skip-list implementation of a finite map; and
- `ConcurrentHashMap`: a hash-table implementation of a finite map.

The more complex algorithms among these are the skip lists and the hashtables, and the non-blocking double-ended queue. A particularly interesting aspect of the skip list implementations is their treatment of the deleted nodes: in order to mark them as logically deleted before unlinking them from the data structure, one inserts a dummy node right after them. Thus, to verify this algorithm, one cannot take the set of values reachable from the head of the list as the representation invariant, but one has to define a direct recursive definition that 'looks ahead' in order to discard logically deleted nodes. The concurrent hashtable uses external chaining, and is somewhat simpler than the skip lists as it uses locking for updating the external linked lists; what is challenging from a verification perspective is the resizing, and in particular to come up with a sufficient model of arrays and to verify the resizing of the array as done by the algorithm. Next, the non-blocking deque (`ConcurrentLinkedDeque`) combines a number of ideas from earlier concurrent linked-list algorithms: it would be useful to attempt to see whether the verification of this algorithm can be decomposed in a way that each of these ideas can be separately specified and verified. In contrast, the single-ended linked-list queues are adaptations of the two standard Michael and Scott queue algorithms [59], and are somewhat simpler, which makes them more suitable for small verification case studies. Since the Michael and Scott queues have already been proved linearizable in the sequential consistency setting [27, 76], the challenge would be (a) to extend these proofs to a relaxed memory setting and also (b) to devise alternative proof techniques that would render the construction of such linearizability proofs more automatic.

2.2 Intel's Thread Building Blocks

A first notable aspect of the TBB library is that it defines a variety of mutual exclusion and reader-writer locks (`mutex`, `spin_mutex`, `queuing_mutex`, `spin_rw_mutex`, `queuing_rw_mutex`, `recursive_mutex`, `reader_writer_locks`) with various different properties and implementation costs. Generally speaking all mutexes are supposed to provide mutual exclusion, but some (namely, the queuing variants) also provide fairness, in the sense that every lock acquisition command will eventually succeed provided that all critical regions have a finite duration. The challenge here would be to verify that the mutex implementations achieve mutual exclusion as well as fairness where applicable.

Next, TBB provides scalable memory allocators that try to avoid synchronization between threads as much as possible (`scalable_allocator`, `cache_aligned_allocator`). This is typically done by having threads manage their own allocation pool and only when this thread-local

pool is depleted, do they synchronize with the other threads to request a larger allocation pool. Another important property of a good concurrent allocator is to avoid creating contention because of 'false sharing.' That is, it should avoid allocating multiple data structures on the same cache line because then threads logically accessing disjoint memory cells will be considered by the hardware as effectively operating on the same data, yielding poor performance.

Intel's TBB also provides highly concurrent implementations of container data structures that permit multiple threads to invoke a method simultaneously on the same container. Currently, it provides various concurrent queues (`concurrent_queue`, `concurrent_bounded_queue`), a concurrent resizable array (`concurrent_vector`), and a concurrent hash map (`concurrent_hash_map`). The main verification question arising from these algorithms is to prove that these implementations are linearizable, specifically with respect to the C11 or the TSO memory model. The implementations themselves are either fine-grained locking or lock-free, so to prove their correctness one should develop concurrent program logics that are sound with respect to the TSO and/or C11 memory models. Another question arising from these container libraries is what kind of ownership transfer happens when inserting an element to the data structure or removing one element from it. In the latter case, who is responsible for deallocating the element?

Table 2.2 details the particular tasks pertaining to case studies described below.

3 Software Transactional Memory

Software Transactional memory (STM) runtime systems [71] ease the task of writing concurrent applications by letting the programmer designate certain code blocks as *atomic*. The intended goal of STMs is to allow designing programs and reason about their correctness as if each atomic block executes as a *transaction*—in one step and without interleaving with other atomic blocks.

3.1 STM Implementations

STMs are developed using different implementations strategies (see discussion below). Verifying STM implementations is a challenging verification problem as sophisticated algorithms are used to efficiently maintain the atomic block abstraction. In particular, inspecting STM implementations is interesting as it allows to expose the transactional protocols and idioms they use. Formalizing the guarantees these protocols provide will allow to develop modular reasoning techniques that can (i) separately verify different components of an STM and show that they implement the intended transactional protocols correctly and (ii) combine the separate proofs leveraging the guarantees provided by the implemented protocols. To illustrate the variety of approaches for implementing STMs, we briefly discuss some of these implementations below.[1]

DSTM [38] is an object based STM that maintains two versions of each object, a current (working) version and an old (stable) version. A transaction marks an object that it is writing. All read operations in a transaction are validated with every new read.

LSA [66] is also an object based STM. However, it has multiple versions for each object unlike DSTM that has only two. The validation is done based on *validity intervals for snapshots*. There is no revalidation of previous reads, instead a global counter is incremented when an update transaction commits.

TL2 [25] uses a global version clock. It maintains a lock for every memory location, and augments it by version number. An update transaction acquire locks on the locations it writes to, increment the global version clock and try to commit by validating read set.

[1] ScalaSTM [15] is discussed in Section 3.3.

Case study	WP2	WP3	WP5	WP6
2.1 java.util.concurrent				
AtomicMarkableReference	T2.3	T3.1	T5.1, T5.2	T6.1, T6.2, T6.3, T6.4
locks.*	T2.3	T3.1		
CountDownLatch	T2.3	T3.1		
CyclicBarrier	T2.3	T3.1	T5.1, T5.2	
Exchanger	T2.3	T3.1	T5.1, T5.2	T6.1, T6.2, T6.3, T6.4
Semaphore	T2.3	T3.1		
Phaser	T2.3	T3.1	T5.1, T5.2	
HSY-stack	T2.1		T5.1, T5.2	T6.1, T6.2, T6.3, T6.4
ArrayBlockingQueue	T2.1		T5.1, T5.2	T6.1, T6.2, T6.3, T6.4
LinkedBlockingQueue	T2.1		T5.1, T5.2	T6.1, T6.2, T6.3, T6.4
SynchronousQueue	T2.1		T5.1, T5.2	T6.1, T6.2, T6.3, T6.4
ConcurrentLinkedQueue	T2.1		T5.1, T5.2	T6.1, T6.2, T6.3, T6.4
LinkedTransferQueue	T2.1		T5.1, T5.2	T6.1, T6.2, T6.3, T6.4
ConcurrentLinkedDeque	T2.1		T5.1, T5.2	T6.1, T6.2, T6.3, T6.4
LinkedBlockingDeque	T2.1		T5.1, T5.2	T6.1, T6.2, T6.3, T6.4
PriorityBlockingQueue	T2.1		T5.1, T5.2	T6.1, T6.2, T6.3, T6.4
DelayQueue	T2.1		T5.1, T5.2	T6.1, T6.2, T6.3, T6.4
ConcurrentSkipListSet	T2.1		T5.1, T5.2	T6.1, T6.2, T6.3, T6.4
ConcurrentSkipListMap	T2.1		T5.1, T5.2	T6.1, T6.2, T6.3, T6.4
ConcurrentHashMap	T2.1		T5.1, T5.2	T6.1, T6.2, T6.3, T6.4
2.2 Thread Building Blocks				
mutex		T3.1		
spin_mutex		T3.1		
queuing_mutex		T3.1		
spin_rw_mutex		T3.1		
queuing_rw_mutex		T3.1		
recursive_mutex		T3.1		
reader_writer_locks		T3.1		
scalable allocator	T2.1	T3.1	T5.1, T5.2	T6.1,T6.2,T6.4
cache aligned allocator	T2.1	T3.1	T5.1, T5.2	T6.1,T6.2,T6.4
concurrent queue	T2.1	T3.1	T5.1, T5.2	T6.1,T6.2,T6.4
concurrent bounded queue	T2.1	T3.1	T5.1, T5.2	T6.1,T6.2,T6.4
concurrent vector	T2.1	T3.1	T5.1, T5.2	T6.1,T6.2,T6.4
concurrent hash map	T2.1	T3.1	T5.1, T5.2	T6.1,T6.2,T6.4

Table 1: Case Studies: Concurrent libraries

NOrec [23] has a global sequence clock for update transactions. Every transaction maintains a local write buffer for write operations and writes to the memory only after validating. Note that transactions do not hold locks while committing.

ASTM [55] is an adaptive STM that switches between two such STMs based on the current workload. (Some STMs work better for write-dominated workloads, while others work better for read-dominated workload.)

The challenges provided by verifying STMs' implementations corresponds closely to WP2 (architecture-driven decomposition), Task 5.2 (Reasoning about transactional idioms).

3.2 Consistency Conditions

Today, the behavior of an STM is defined through a *consistency condition* that restricts its possible executions. Several such conditions have been proposed, including *opacity* [33, 35], *virtual world consistency* [43], *TMS* [50, 29] and *DU-opacity* [10]. Another consistency condition [28] for STMs was defined using an I/O automata. Opacity [34] seems to be the most widely accepted for STMs and most of the known STMs are opaque. Roughly speaking, it means that all transactions, even aborted ones, have the same serialized view. *Virtual world consistency* [43] is a weaker condition and states that the aborted transactions can have different serialized views of the execution, but committed transactions have the same view.

Consistency conditions provide a description of an STM behavior from the point of view of their implementation. Unfortunately, These descriptions are not connected to the semantics of the programming language in which their client programs are written. In fact, it is not clear which of the aforementioned conditions provide the programmer with behaviors that correspond to the intuitive notion of atomic blocks, and which of them puts the minimal restrictions on STM implementations needed to achieve this.

The disconnect between STMs and programming languages thwarts modular reasoning because it prevents the formal treatment of transactions as atomic blocks, thus going against the very purpose of using STMs to develop reliable concurrent software! An interesting problem, thus, is to connect the two views. Specifically, formalizing the intuitive expectations of a programmer from the transactional system by means of *observational refinement* [37, 36], and determining which of the consistency conditions provide the expected notion of atomic blocks for different choices of programming languages and under different notions of observations are interesting and important challenges.

Our initial results show that, in certain respects, an opaque STM observationally refines the notion of atomic blocks in a particular, idealized, programming language [9]. This result suggests that a programming language-based approach for evaluating and comparing TM consistency conditions is a viable one. Obtaining additional results of this kind for other consistency conditions and richer programming languages will allow to reduce the effort of proving that a TM implements its programming language interface correctly (see below), by only requiring its developer to show that it satisfies the corresponding consistency condition.

This challenge corresponds closely to Task 2.1. (Decomposition on library boundaries) and Task 5.4. (Integration with decomposition techniques and state-of-the-art) methods.

3.3 Programming Language Interfaces for Transactional Memory

There are several interfaces connecting programming languages with STMs. Below, we list two interesting examples coming from two different ends of the spectrum of programming languages community: An interface proposed by the industry (*C++* [73]) and an interfaced proposed by the research community (*Scala* [60]).

C++. The Draft Specification of Transactional Language Constructs for C++ [7] introduces transactional language support for the C++ language. It specifies how C and C++ programs are to be interfaced with STMs. The specification (or more precisely, an earlier version of it [6]) is supported by Intel C++ STM Compiler (Prototype Edition).[2] Intel also provides a few client applications to be used with their compilers, e.g., an STM-based hashtable.[3]

The draft allows for two different flavors of transactions, supporting two different levels of isolation: *strict transactions* (`__transaction_atomic`), which appear to be executed atomically as an indivisible statement, and *relaxed transactions* (`__transaction_relaxed`), where arbitrary-non transactional code can be executed by a transaction and can be interleaved with other non-transactional code. The former requires called functions to be declared `transaction_safe`, while the latter may include `transaction_unsafe` functions. Unsafe functions may execute *irrevocable operations*: operations with side effects that the system cannot roll back (`transaction_callable`). Note that the programming model proposed by relaxed transactions diverges greatly from the standard model of STMs, and places many challenging problems both for programmers and for verifiers. An extension of the notion of transactional statements is offered for *transactional expressions*, proposed to be used as initializers, that can be evaluated outside of a transaction body as if executed inside a transaction.

The draft also allows for transaction cancellation using the `__transaction_cancel` construct, which ensures that a canceled transaction has no effect and avoids the need to write cleanup code to undo the partial effects of an atomic transaction statement. In addition, a programmer can `throw` an exception from a canceled transaction by combining the cancel statement with a `throw` statement to form the cancel-and-throw idiom.

Transactions can also be *nested* within an `outer` transaction. Only the `outer` transaction can cancel the transaction. The precise semantics of nested transactions is not made clear in the standard; ordering constraints are specified only for the `outer` transactions.

As discussed above, the proposed C++/STM interface provides certain challenging features for formalization and verification, such as the interrelation between transactional and non-transactional code. We note that the standard is still at its formative stages, and request for comments is made for the community of researchers and practitioners in order to gather feedback for the next version of the standard.[4] This provides a unique opportunity for the formal studies performed within the ADVENT project to impact the *future* of STM-based programming.

Scala. ScalaSTM [15] is a proposed programming interface for using STMs in the SCALA language [60], and provides an API that supports multiple STM implementations. ScalaSTM provides an interesting take on the STM-based programming model. Addressing the scalability/performance problem imposed by the need to track every load and store, ScalaSTM manages only Ref-s (mutable memory cells). This means that there are fewer memory locations to manage, and no bytecode instrumentation or compiler modifications are required. The usefulness of Ref-s is multiplied by the language's good support for immutable data structures.

To write transactions, programmers need to wrap their code inside `atomic` blocks. As usual, atomic blocks can contain composed operations. However, special support is given to single-operation instructions: `Ref.single` returns an instance of Ref.View, which acts just like the original Ref except that it can also be accessed outside an atomic block. Each method on `Ref.View` acts like a single-operation transaction, hence the name. `Ref.View` provides several methods that perform both a read and a write, such as `swap`, `compareAndSet` and `transform`, which act as indicated by their names.

ScalaSTM also provides a `retry` facility which can allow an atomic block whose operation cannot be completed on its current input state to roll back, wait for one of its inputs to change,

[2] `http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition`

[3] `https://secure-software.intel.com/en-us/system/files/article/139902/intel-stmtest-hashtable.c`

[4] `http://groups.google.com/group/tm-languages`

and then retry execution. This is roughly analogous to a call to `wait` where ScalaSTM automatically generates the matching `notifyAll`.[5] Another way to proceed after an atomic block ends in `retry` is to provide an *alternative*: It is possible to chain atomic blocks using `orAtomic`; the lower alternatives will be tried if the upper ones call retry. This lets compose functions that block using retry, or to convert to and from blocking behavior.

The use of `retry`, and rollbacks in general is quite delicate: ScalaSTM might need to try an atomic block more than once before it succeed. Any call into the STM might potentially discover the failure and trigger the rollback and retry. Local non-Ref variables that have a lifetime longer than the atomic block *will not* be rolled back, and so they should be avoided. This means that a client can observe the (partial progress) of an aborted transaction!

ScalaSTM provides scalable concurrent sets (`TSet`) and maps (`TMap`) that can be used inside or outside transactions. These data structures support *consistent iterations* by making `TMap.View.iterator` and `TSet.View.iterator` return an iterator over an *atomic snapshot* of the collection. The user can also generate snapshot of these data using the `clone` function: `TMap` and `TMap.View` return an `immutable.Map` from snapshot; `TSet` and `TSet.View` return an `immutable.Set`. The `TMap` resp. `TSet` returned from `clone` is a fully-functional transactional (and concurrent) collection. Implementation-wise, `TMap` and `TSet` use mutable hash tries constructed from Ref-s, with generation numbers that control copy-on-write. The algorithm that supports these features is a novel hybrid of Nathan Bronson's *SnapTree* [18] and *Transactional Predication* [19], described in Chapter 4 of his thesis [17].

To summarize, many STMs implementations exist, different implementations satisfy different correctness conditions, and programmers are expected to use STMs via a variety of programming language interfaces, where different interfaces provide different capabilities and expose different semantics. This makes actual programming with STMs nontrivial, e.g., when relaxed transactions in C++, which breaks the notion of atomicity, are used.

This case study corresponds closely to the goals of WP2 (architecture-driven decomposition and WP5 (Architecture-driven verification of complex software components). Understandings gained from this studies can affect the future development of STMs, in concert with the desired impact discussed above.

Table 3.3 details the particular tasks pertaining to case studies described below.

4 Operating System Components

In the operating systems domain, we will primarily use case studies from the Linux operating system. Linux is a widely-used open-source operating system, used for desktops and servers and also serving as the basis for the ubiquitous Android mobile phone system. It is relatively well-documented, with books on architecture [54] and a code commentary [16] available. As such, it is ideal to motivate the development of reasoning principles for systems software in the project. We will refer both to the most up-to-date version of the system, and to its older version 2.6.11 covered by the existing code commentary [16]. The following components form the most promising case studies relevant to the project.

Table 4.4 details the particular tasks pertaining to case studies described below.

4.1 Scheduler and Process Management

Task 2.2 is concerned with reasoning about virtualizing components, which provide the illusion to the rest of the system of running on a higher-level machine. A scheduler and, more generally,

[5] As part of its implementation of optimistic concurrency, the STM keeps track of the read set of every atomic block, i.e., the set of Ref-s that have been read during the execution of the atomic block. This means that the STM can efficiently block the current thread until another thread has written to an element of its read set, at which time the atomic block can be retried.

Case study	WP2	WP5	WP6
3.1 STM Implementations			
DSTM	T2.1	T5.2	T6.1
LSA	T2.1	T5.2	T6.1
TL2	T2.1	T5.2	T6.1
NOrec	T2.1	T5.2	T6.1
ASTM	T2.1	T5.2	T6.1
3.2 Consistency Conditions			
Opacity	T2.1	T5.4	
VWC	T2.1	T5.4	
TMS1	T2.1	T5.4	
TMS2	T2.1	T5.4	
DU-opacity	T2.1	T5.4	
3.3 Programming Language Interfaces for Transactional Memory			
C++			
strict transactions	T2.1	T5.4	
relaxed transactions	T2.1	T5.4	
rrevocable operations	T2.1	T5.4	
transactional expressions,	T2.1	T5.4	
transaction cancellation	T2.1	T5.4	
ScalaSTM			
Ref.Single	T2.1	T5.3	
Ref.View	T2.1	T5.3	
retry	T2.1	T5.3	
notifyAll	T2.1	T5.3	
TMap.View.iterator	T2.1	T5.3	
TSet.View.iterator	T2.1	T5.3	

Table 2: Case Studies: Software transactional memory

process management code is an example of such a component: it provides the illusion of a machine where every process has a dedicated CPU. We will use the Linux scheduler and process management code to motivate the development of the corresponding reasoning principles. Of particular interest are:

- Implementation of wait queues: the functions manipulating the `wait_queue_t` structure.
- The context switch code: the `switch_to` macro.
- Process creation, such as the `fork()` function.
- Scheduling functionality, such as runqueue manipulations (the `runqueue` structure) and the `scheduler_tick()` function.

4.2 Virtual Memory Subsystem

Virtual memory is another example of a virtualizing component: it provides the illusion of a machine where every process has unbounded memory. The virtual memory implementation in Linux is extremely subtle, primarily due to the desire to achieve maximal performance on multicore architectures. For example, for every user process in Linux, the virtual memory subsystem maintains its virtual address space description and page tables; additionally, memory region descriptors of different processes are linked by data structures used to optimism memory reclamation. All these data structures are interrelated, yet concurrently accessible, with on-demand and copy-on-write memory allocation and memory reclamation running alongside each other. Furthermore, the data structures make references to two layers of physical memory managers, a page cache and underlying filesystems, again, all concurrently accessible. Safety is ensured by a sophisticated combination of fine-grained locking and non-blocking techniques. Considering the virtual memory subsystem will therefore motivate not only the development in Task 2.2, but also the development in Tasks 5.1 and 5.3. The particular pieces of functionality we will pay attention to are as follows:

- The zoned page frame allocator (e.g., the `alloc_page` function) is the primary mechanism of allocating memory on the granularity of pages. Its interesting features include supporting concurrent access and keeping per-CPU caches of free pages.
- The slab allocator (`mm/slab.c`) is implemented on top of the page frame allocator, which allocates objects of the same time. This allocator also allows for concurrent access. Another feature challenging for verification is the deferred freeing of objects using RCU (see below).
- Functions managing the process address space and its mapping to the physical one via page tables. These include routines for allocating and deallocating memory, demand paging, copy-on-write and the page-fault handler (e.g., `do_page_fault()`). They exhibit lots of verification challenges, including the use of interlinked data structures and fine-grained concurrency, as explained above.
- The Virtual File System is a kernel software layer that handles all system calls related to a standard Unix filesystem. In the context of virtual memory, it will be primarily interesting for its interactions with the rest of the virtual memory subsystem, e.g., by swapping unused pages from the memory to the disk and back.

4.3 Interrupt Handling

Communication with external devices in systems software is done with the aid of interrupts, programmed via a hardware circuit called the interrupt controller. Developing reasoning methods for the code interacting with interrupt controllers is the subject of Task 3.3. To motivate

this development, we will consider Linux's code interacting with Intel's I/O Advanced Programmable Interrupt Controller (I/O APIC). This code is very subtle and challenging to reason about because of the level of concurrency involved: we have to deal with multiple asynchronously arriving interrupts that have different priorities and may be handled by different CPUs in the system. Our starting point will be the `do_irq()` function, which is invoked to execute all interrupt service routines associated with an interrupt. We will also consider the implementations of the abstraction of *deferrable functions*, which allow taking non-urgent processing out of the interrupt context. Linux provides several flavors of these:

- *Softirqs* are statically allocated and can run concurrently on several CPUs; thus, they are reentrant functions. The entry point for the relevant code is the `do_softirq()` function.
- Tasklets can be allocated and initialized at runtime and their execution is controlled more strictly by the kernel than that of softirqs: tasklets of the same type are always serialized. The entry point for the relevant code is the `tasklet_schedule()` function.
- *Work queues* allow kernel functions to be activated and later executed by special kernel worker threads. The entry point for the relevant code is the `queue_work()` function.

The implementations of these features on top of the interrupt handling mechanism in Linux will also motivate the development in Task 2.3.

4.4 RCU Synchronization

Read-Copy-Update (RCU) [58, 56] is a non-standard synchronization mechanism that allows multiple readers to access a shared object concurrently with a single writer. RCU synchronization is attractive because it allows efficient implementations in which readers incur a minimal synchronization overhead, if any at all. It has achieved a widespread adoption in the operating systems domain, becoming one of the most used synchronization mechanism in the Linux kernel (see., e.g., `linux/kernel/rcupdate.c`) [16]. In an operating system kernel, its efficiency is achieved by piggy-backing on the implementation of the scheduler; however user-level implementations also exist [24] [5]. RCU has been used to construct a number of concurrent objects, including concurrent hash tables [75], concurrent red-black trees [42] and software transactional memory implementations [41].

The unique aspect of the RCU synchronization mechanism is a `sync` command that allows a writer thread to block until all the reader threads that are accessing the concurrent object *at the time* the `sync` command (called `synchronize_rcu()` in the Linux Kernel) is invoked exit their critical sections. The writer, however, does not have to wait for readers that enter their critical sections *after* `sync` is invoked. Programmers use this primitive in different and subtle ways. For example, one typical pattern is to use `sync` to wait until all the readers drop their references to a certain object before deallocating it. Indeed, RCU-based memory reclamation, implemented by the `__rcu_reclaim()` method, is key in Linux Kernel [57]. Another, more subtle use of `sync`, is to ensure that readers see a sequence of modifications in a particular order. The latter use is exploited in implementations of data structures such as hash tables [75] and red-black trees [75] to ensure their *linearizability* [40]—the property that plays the role of functional correctness for concurrent objects.

The form of synchronization provided by RCU is very different from those provided by classical synchronization primitives, such as mutexes, monitors or reader-writer locks. Algorithms using RCU are also different from non-blocking algorithms [39], as the latter only use low-level compare-and-swap operations, while RCU provides a higher-level synchronization abstraction. All this makes existing methods for reasoning about concurrent algorithms with either high-level [61, 48] or low-level [30, 77, 26, 21] primitives inappropriate for reasoning about RCU-based ones.

Case study	WP2	WP3	WP5	WP6
4.1 Scheduler and Process Management				
wait_q_t	T2.2	T3.1		
switch_to	T2.2	T3.1		
fork()	T2.2	T3.1		
scheduler_tick()	T2.2	T3.1		
runqueue	T2.2	T3.1		
4.2 Virtual Memory Subsystem				
zoned page frame allocator	T2.2	T3.2	T5.1, T5.3	
slab allocator	T2.2	T3.2	T5.1, T5.3	
process address space mapping	T2.2	T3.2	T5.1, T5.3	
virtual file system	T2.2	T3.2	T5.1, T5.3	
4.3 Interrupt Handling				
do_irq()	T2.3	T3.3		
do_softirq()	T2.3	T3.3		
tasklet_schedule()	T2.3	T3.3		
queue_work()	T2.3	T3.3		
4.4 RCU Synchronization				
rcuupdate.c	T2.3	T2.3	T5.1	T6.1
synchronize_rcu()	T2.3	T2.3	T5.1	T6.1
_rcu_reclaim()	T2.3	T2.3	T5.1	T6.1
rcu-based hashmap	T2.3	T2.3	T5.1	T6.1
rcu-based red/black tree	T2.3	T2.3	T5.1	T6.1

Table 3: Case Studies: Operating systems components

We have already obtained initial results that allow to verify RCU implementations [32] for sequentially consistent memory. However, this was done in an idealistic setting, as no existing hardware provides this kind of consistency. Specifically, RCU is designed to work in weak-memory environments, hence extending our reasoning techniques (or developing new ones) to such environments is a challenging problem.

Relativistic programming [42, 75] is a novel programming technique advocated by highly experience system programmers. It offers a programming methodology that leverages the preferential treatment of readers by RCU to implement concurrent data structures that provide high-performance in practice. While attractive, the methodology has no formal basis and no associate reasoning techniques which hider it adoption. An interesting challenge is to provide such a formalization to this methodology and develop the accompanying set of reasoning techniques.

To summarize, specific test cases include RCU-based concurrent hash table [75] and red-black tree [42] data structures as well as (submodules of) the RCU implementation found in Linux (`/linux/kernal/rcuupdate.c`, `linux/kernel/rcutiny.c`, and `/linux/kernel/rcutree.c`). The latter is particularly challenging because of the subtle interactions between the RCU implementation and the scheduler. We have already established contact with Paul Paul McKenney, a Distinguished Engineer at IBM Linux Technology Center (USA), who is the developer of RCU and its maintainer in Linux. He and his group will keep us updated about the developments in the area, which will let us adjust the list of case studies as necessary.

This test case is of interest to WP2 (Architecture-driven decomposition), Tasks 3.1 (Reasoning on weak memory models) and 5.1 (Spatio-temporal reasoning).

Table 4.4 details the particular tasks pertaining to case studies described below.

5 Device Drivers

A device driver is a piece of code that adds support for a particular type of hardware device to an operating system. As a trivial example, the Linux USB mouse driver, `usbmouse.c`, when it detects the presence of a USB mouse device, registers itself with the Linux kernel's USB subsystem as a listener for incoming events from the device. When it receives such an event, such as a "mouse moved" event, it forwards it to the Linux kernel's input subsystem, which in turn offers it through a generic interface to application programs in user space.

Device drivers constitute interesting case studies for the technologies to be developed in this project, for both technical and pragmatic reasons. Technically, firstly, because they are highly concurrent: events come in concurrently from hardware devices and applications, and to optimize throughput and responsiveness, synchronization must be efficient; secondly, device drivers are programmed in a highly asynchronous style; and thirdly, the kernel APIs used by device drivers have particularly complex and intricate semantics. Pragmatically, formal verification of device drivers is especially useful because device drivers are often written by relatively inexperienced hardware device manufacturers rather than the core kernel developers; furthermore, since they are relatively small and self-contained, their formal verification is more likely to be economically feasible.

Device drivers are particularly appropriate case studies for architecture-driven verification, since they are embedded within a particular kernel's architecture. They benefit WP2 (Architecture-driven decomposition), since device drivers are typically combined into stacks of drivers where each layer implements a higher-level abstraction (e.g. a mouse interface) on top of a lower-level abstraction (e.g. a USB interface), which in turn is implemented by another driver on top of bare hardware. Within WP5 (Architecture-driven verification of complex software components), relevant tasks include Task 5.1 (Spatio-temporal reasoning), since device drivers manipulate resources (spatial) in a concurrent and interactive setting (temporal); and Task 5.3 (Modular verification for interlinked structures), since such structures are prevalent in operating system kernels. The case studies benefit also WP6 (Automation), including in particular Tasks 6.1 (Automation for spatio-temporal reasoning) and 6.2 (Pattern- and paradigm-based automation), the latter because device drivers often display recurring patterns and are written in a particular programming paradigm. All case studies will of course critically support (and be supported by) Task 6.4 (Tool infrastructure).

While device drivers for Windows, Linux, Mac OS X, or any other operating system are in principle equally suitable as case studies, we will use Linux device drivers because 1) most Linux device drivers are open source, giving access to a large library of real-world code on which to test our technologies, 2) the operating system itself and the development environment are open source, allowing maximum information and freedom in experimentation, and 3) we already have some experience in verifying Linux device drivers [64]. However, we believe the technologies we will develop will be equally applicable to device drivers for other, more widely deployed operating systems.

Specific examples of interesting and important device drivers include:

- Network interface drivers, such as Intel's open source `e1000e` driver for modern Intel onboard Ethernet adapters or `iwlwifi` for Intel wireless network interface cards
- Mass storage drivers, such as the SCSI driver `sd.c`. Linux uses this driver also for non-SCSI devices which are exposed by lower layers of the Linux driver stack as virtual SCSI devices, including SATA hard disk drives or optical drives and USB mass storage devices.

Notice that the above device drivers are potentially exposed to malicious content; therefore, they should be resilient not just against malformed data caused by random events but also against intentionally crafted malformed data.

Case study	WP2	WP3	WP6
e1000e	T2.1	T5.1, T5.3	T6.1, T6.2, T6.4
iwlwifi	T2.1	T5.1, T5.3	T6.1, T6.2, T6.4
sd.c	T2.1	T5.1, T5.3	T6.1, T6.2, T6.4
usbmouse.c	T2.1	T5.1, T5.3	T6.1, T6.2, T6.4
usbkbd.c	T2.1	T5.1, T5.3	T6.1, T6.2, T6.4

Table 4: Case Studies: Device drivers

To give a better idea of the state of the art and the challenges we will confront, in the remainder of this section we will discuss two device drivers that we have already worked on in earlier work: `usbmouse.c` and `usbkbd.c`. Note that these drivers are much simpler than the ones identified above. The former is more or less the simplest possible device driver, and therefore offered a good first step for the incremental development of our technologies. The latter includes another instance of the simple pattern implemented by `usbmouse.c`; additionally, it also includes a more complex pattern of concurrent interaction, which will likely benefit from spatio-temporal reasoning for simpler specification and verification.

usbmouse.c This driver communicates with USB mouse devices through the USB HID (Human Interface Device) Boot Protocol. (The Boot Protocol is a simplified variant of the more comprehensive Report Protocol and is intended to be used in constrained systems, such as embedded systems.) The driver registers itself with the Linux kernel's USB subsystem to be notified of incoming events from the mouse device (in particular, move events and button events), and forwards these to the Linux kernel's input subsystem, which in turn offers them through a general interface to application programs in user space. `usbmouse.c` has 173 lines of code (excluding comments and blank lines), and consists of the following functions: `usb_mouse_irq`, `usb_mouse_open`, `usb_mouse_close`, `usb_mouse_probe`, and `usb_mouse_disconnect`.

We have verified memory safety, data-race-freedom, and compliance with API constraints of this driver. However, this verification exercise currently requires too much manual work: we obtain an annotation overhead of 167 lines of annotations for every 100 lines of code when using our verification tool VeriFast [64]; therefore, we will develop automation technology to reduce the verification effort. Furthermore, currently there is no way to modularly specify and verify the correct I/O behavior of this driver. This is another aspect that we will focus on.

usbkbd.c This driver communicates with USB keyboard devices through the USB HID Boot Protocol. Similarly to `usbmouse.c`, it registers to receive key events from the keyboard device and forwards them to the input subsystem; furthermore, it receives keyboard LED state change requests from applications through the input subsystem and forwards them to the keyboard device. While these are in principle two more occurrences of the same pattern exemplified by `usbmouse.c`, a complication is introduced by the fact that a LED change request can come in while another LED change request is being processed. Dealing with this race condition complicates in particular the verification of correct cleanup when the device is concurrently being disconnected. Our current approach to the verification of this pattern, as described in [64], is rather cumbersome. In addition to the goals of automation and I/O verification, we hope to simplify the verification of this pattern using spatio-temporal reasoning.

Table 5 details the particular tasks pertaining to case studies described below.

6 Eventually Consistent Replicated Stores

Cloud computing allows moving services, computation and/or data off-site to an internal or external, centralized facility. By making data available in the cloud, it can be more easily and

ubiquitously accessed, often at much lower cost, increasing its value by enabling opportunities for enhanced collaboration, integration, and analysis on a shared common platform.

To achieve availability and scalability, cloud systems often rely on *replicated stores*, allowing multiple clients to issue operations on shared data on a number of *replicas*, which communicate changes to each other using message passing. For example, large-scale Internet services rely on *geo-replication*, which places data replicas in geographically distinct locations, and applications for mobile devices store replicas locally to support offline use. One benefit of such architectures is that the replicas remain locally available to clients even when network connections fail. Unfortunately, the famous CAP theorem [26] shows that such high **A**vailability and tolerance to network **P**artitions are incompatible with *strong* **C***onsistency*, i.e., the illusion of a single centralized replica handling all operations. For this reason, modern replicated stores often provide weaker forms of consistency, commonly dubbed *eventual consistency* [74]. Here 'eventual' refers to the guarantee that, if clients stop issuing update requests, then the replicas will eventually reach a consistent state.

Geo-replication is a hot research area, and new architectures for eventually consistent systems appear every year [31, 52, 72, 70, 20, 51, 11, 22]. Unfortunately, whereas consistency models of classical relational databases have been well-studied [63, 12], those of geo-replicated systems are poorly understood. The very term eventual consistency is a catch-all buzzword, and different systems claiming to be eventually consistent actually provide subtly different guarantees. This makes eventually consistent replicated stores a fruitful source of case studies for applying formal techniques. Below we describe the replicated eventually consistent stores and their components that we identified as case studies.

6.1 Replicated Data Types

Eventually consistent replicated stores allow the replicas to be temporarily inconsistent. This enables them to satisfy clients' requests from the local replica immediately, and broadcast the changes to the other replicas only after the fact, when the network connection permits this. However, this means that clients can concurrently issue conflicting operations on the same data item at different replicas; furthermore, if the replicas are out-of-sync, these operations will be applied to its copies in different states. For example, two users sharing an online store account can write two different postcodes into the delivery address; the same users connected to replicas with different views of the shopping cart can also add and concurrently remove the same product. In such situations the store needs to ensure that, after the replicas exchange updates, the changes by different clients will be merged and all conflicts will be resolved in a meaningful way. Furthermore, to ensure eventual consistency, the conflict resolution has to be uniform across replicas, so that, in the end, they converge to the same state.

The protocols achieving this are commonly encapsulated within *replicated data types* that implement objects such as registers, counters, sets or lists, with various conflict-resolution strategies. The strategies can be as simple as establishing a total order on all operations using timestamps and letting the last writer win, but can also be much more subtle. Thus, a data type can detect the presence of a conflict and let the client deal with it: e.g., the *multi-value register* used in Amazon's Dynamo key-value store [31] would return both conflicting postcodes in the above example. A data type can also resolve the conflict in an application-specific way. For example, the *observed-remove set* [69] processes concurrent operations trying to add and remove the same element so that an add always wins, an outcome that may be appropriate for a shopping cart. Replicated data type implementations are often nontrivial, since they have to maintain not only client-observable object state, but also metadata needed to detect conflicts and resolve them and to handle network failures. This makes reasoning about their behavior challenging. Replicated data type implementations will provide case studies for Task 5.1; they will also be relevant for the development of techniques for reasoning about weak memory consistency in Task 3.1.

We will consider a number of data types as case studies.

- **Registers**: last-writer-wins register and multi-value register [31].
- **Counters**, including recent highly-optimized implementations [8].
- **Sets** and set-based data types, such as **graphs**, with various conflict resolution policies [70, 69, 13, 14].
- **Sequence data types**, used for collaborative editing [67, 65].

We will consider both published data type designs and, where available, their actual implementations in the context of replicated stores, such as Riak [3] (see below). We have also established contact with Marc Shapiro's group at UPMC/INRIA (Paris) working on replicated data types. This group will keep us updated about the developments in the area, which will let us adjust the list of case studies as necessary.

6.2 Whole-System Architecture

The following replicated stores will serve as case studies both for the high-level aspects of architecture used in distributed systems and for the challenging low-level components. The variants of eventual consistency they provide as an interface to the programmer (causal+, RedBlue, etc.; see below) will also serve as an object of study. In the cases when the systems source code is not publicly available, we will rely on the descriptions of the systems architecture published in systems conferences and on direct contacts with the systems developers. The stores we consider will provide case studies for Tasks 2.2, 5.1 and 5.2.

Riak [4]. Riak is an open-source eventually consistent replicated store written in Erlang. Unlike the systems considered below, it provides a relatively simple key/value data model: data items usually consist of a string which represents the key and the actual data which is considered to be the value in the "key - value" relationship. Using the recent `riak_dt` package [3], a value in Riak can be represented by a replicated data type. This component of Riak will therefore serve to highlight the interfaces between realistic implementations of replicated data types and the rest of the replicated store implementation.

Another interesting aspect of Riak architecture is that data is automatically distributed evenly across nodes using *consistent hashing*, which ensures that new nodes can be added with automatic, minimal reshuffling of data. Namely, when machines are added, data is rebalanced automatically with no downtime. New machines take responsibility for their share of data by assuming ownership of some of the key ranges; existing cluster members hand off the relevant ranges of the key space and the associated data. Programming such data distribution and reconfiguration is challenging and the corresponding code will serve as a fruitful source of challenges for formal verification.

COPS [52] and Eiger [53]. The COPS system and its successor, Eiger, provide consistency guarantees stronger than those of simple key-value stores such as Riak. Namely, they implement causal consistency with convergent conflict handling, or for short, causal+ consistency. Unlike purely eventually consistent systems, causal+ consistency ensures that the data store respects the causal dependencies between operations. For example, if a user uploads a picture to a web site, the picture is saved, and then a reference to it is added to that user's album, then programmers never have to deal with the situation where they can get the reference to the picture but not the picture itself. COPS and Eiger systems support applications that are hosted from a small number of large-scale datacenters and ensure that data propagation between them respects causal dependencies of the kind just illustrated. The promising aspects of the systems architecture to be investigated are the algorithms used to track and preserve such causal dependencies, the garbage collection of the meta-data needed for this and the algorithms used to implement lightweight forms of transactions over the data.

Gemini [51]. The Gemini system goes further than COPS and Eiger in strengthening the consistency guarantees, by letting several consistency levels coexist within the same system. The resulting consistency model is called Red-Blue consistency. The intuition behind it is that blue operations execute locally and are lazily replicated in an eventually consistent manner. Red operations, in contrast, are serialized with respect to each other and require immediate cross-site coordination. RedBlue consistency preserves causality in all cases. Algorithms implementing the mixed consistency model and the consistency model itself should prove an interesting object of study from the perspective of software verification.

To increase the number of cheaper Blue operation, Gemini allows the programmer to decompose operations on the database into two components: (1) a generator operation that identifies the changes the original operation should make, but has no side effects itself, and (2) a shadow operation that performs the identified changes and is replicated to all sites. Only shadow operations are colored red or blue. This broadens the space of potentially blue operations, but requires careful reasoning that the decomposition into generator and shadow operations does not violate application invariants. Therefore, in addition to the implementation of Gemini, we will also consider the architecture of applications that have been implemented on top of it [51].

We have already established contacts with the team of Gemini's developers at Universidade Nova de Lisboa and MPI-SWS (one of the partners). They will help us in our formal investigations of the system architecture and will keep us updated about ongoing developments.

Walter [72]. Walter is a geo-replicated key-value store that supports transactions with a nontrivial consistency guarantee called Parallel Snapshot Isolation (PSI). This aims to provide a balance between consistency and latency appropriate for web applications. With PSI, clients accessing the same replica observe transactions according to a consistent snapshot and a common ordering of transactions. Across replicas, PSI enforces only causal ordering, not a global ordering of transactions, allowing the system to replicate transactions asynchronously across sites. We will study both the consistency guarantee for transactions exported as an interface to the user and the architecture used to implement transactions with this consistency level.

6.3 Erlang OTP Library

Eventually consistent stores are often written using specialized libraries aiming to make programming distributed systems easier. One such library that has achieved a widespread adoption is Erlang's Open Telecom Platform (OTP) [1], e.g., used by Riak [4]. The library provides abstractions of common protocol patterns, called behaviors. A behavior consists of a library that implements a pattern of communication, plus the expected signatures of the callback functions. An instance of a behavior needs some interface code wrapping the calls to the library plus the implementation callbacks, all largely free of message passing.

We will study the interfaces and architecture of the OTP library and, in particular, the following three main behaviors that are challenging for verification:

- Generic server (`gen_server`). The generic server abstracts the standard request-response message pattern used in client-server or remote procedure call protocols in distributed computing. It provides sophisticated functionality: responses can be delayed by the server or delegated to another process; calls have optional timeouts; the client monitors the server so that it receives immediate notification of a server failure instead of waiting for a timeout.

- Generic finite state machine (`gen_fsm`). Many concurrent algorithms are specified in terms of a finite state machine model, and this behavior implements this pattern. The message protocol that it obeys provides for clients to signal events to the state machine, possibly waiting for a synchronous reply. The application-specific callbacks handle these events, receiving the current state and passing a new state as a return value.

Case study	WP2	WP5
6.1 Replicated Data Structures		
Registers	T2.1	T5.1
Counters	T2.1	T5.1
Sets	T2.1	T5.1
Graphs	T2.1	T5.1
Sequence types	T2.1	T5.1
6.2 Whole-System Architecture		
Riak	T2.1	
COPS	T2.1	
Eiger	T2.1	
Gemini	T2.1	
Walter	T2.1	
6.3 Earlng OTP Library		
gen_server	T2.1	T5.1, T5.3
gen_fsm	T2.1	T5.1, T5.3
gen_Event	T2.1	T5.1, T5.3

Table 5: Case Studies: Eventually consistent replicated stores

- Generic event handler (`gen_event`). An event manager is a process that receives events as incoming messages, then dispatches those events to an arbitrary number of event handlers, each of which has its own module of callback functions and its own private state. Handlers can be dynamically added, changed, and deleted. Event handlers run application code for events, frequently selecting a subset to take action upon and ignoring the rest. This behavior naturally models logging, monitoring, and "pubsub" systems.

These provide case studies relevant to Tasks 2.1 and 5.1. Table 6.3 details the particular tasks pertaining to case studies described below.

7 Weak Memory Models

In a sequential setting, the behavior of memory is simple: when a program reads a memory location, it receives the value that it wrote most recently to that location. In a concurrent setting, where the program does not define a total order on all memory accesses, however, this simple rule no longer applies. The question of what values may be yielded by reads in concurrent programs is known as the platform's *memory consistency model* (or *memory model* for short). The simplest model, which is most commonly assumed in the literature but which is not implemented by any commonly used programming platform, is *sequential consistency* (SC): in a sequentially consistent execution, there is a total order on all memory accesses such that each read yields the value of the most recent preceding write in that total order. Any memory model which admits non-SC executions is called a *weak memory model*.

The memory models offered by processor architectures are weak due to performance-enhancing measures such as local caches, pipelines, and speculation. The memory models offered by programming languages are weak both due to the weakness of the target architecture and due to compiler optimizations such as common subexpression elimination and loop-invariant code motion.

The increasing reliance on parallelism for performance means that upholding the fiction of SC, which requires the insertion of expensive and often unnecessary *memory barriers* into programs, is no longer feasible: developers must program directly against the weak memory models that are offered by programming languages such as C and C++ and by processor architectures

such as x86/x64 and ARM/POWER. However, they currently lack the reasoning principles and tool support to do so safely. The goal of Task 3.1 (Reasoning on weak memory models) is to address this problem. In this section, we identify a number of important weak memory models that can serve as case studies for the technologies developed in Task 3.1 and elsewhere. We also identify a program that interacts with the memory model in an interesting way.

x86-TSO [62] is the memory model offered by Intel and AMD's x86 and x64 processors, and is therefore the relevant processor memory model for most desktop systems and many server systems. It is defined using the concept of a *store buffer*: in the x86-TSO memory model, each hardware thread (hereafter called *processor* for simplicity) has a dedicated *store buffer*, which is a FIFO buffer storing values that have been written by the processor but that have not yet been propagated to main memory. Specifically, whenever the processor (which for the purposes of defining this model is assumed to simply execute the program in textual order) performs a write operation, this operation (i.e. the address and the value) is added to the end of the processor's write buffer. When the processor performs a read operation, there are two cases: if a write to the same location is in the store buffer, the value of the most recent such write is returned. Otherwise, the value in main memory is returned. (Here, main memory is assumed to be SC.) Thirdly, at indeterminate points in time, propagations may happen: a propagation means that a write operation is removed from the front of a write buffer and executed on main memory.

Notice that in this model, all processors see all writes *except their own* in the same total order. Hence, the name *Total Store Order*. An alternative way to look at the model is to say that all writes happen in a global total order, but reads by a processor may be satisfied by old writes from the same processor instead of by the current value in main memory.

To enable mutual exclusion, x86 architecture instructions can be augmented with a LOCK prefix. In that case, in the x86-TSO model, propagations by any processor are blocked for the duration of the instruction, and the issuing processor's store buffer is flushed completely at the end of the instruction. For example, the CMPXCHG instruction combined with the LOCK prefix can be used to implement the standard compare-and-swap operation. Furthermore, the architecture includes an MFENCE instruction whose only effect is to flush the local store buffer.

The x86-TSO model is a relatively strong memory model: many useful concurrency constructs can be built without the need for LOCK prefixes or MFENCE instructions. The producer-consumer pattern is an example of this: since store buffers are FIFO, storing a pointer to a data structure only after the data structure is initialized is sufficient to ensure that other processors will see only the initialized state of the data structure.

The memory model of Oracle's SPARC instruction set architecture, SPARC-TSO, is similar to x86-TSO.

Power ISA [68] is the instruction set architecture implemented by IBM's POWER7 and POWER8 processors for server systems, as well as by a range of other processors by various manufacturers for applications ranging from embedded systems to supercomputing, including the processors in Microsoft's Xbox 360, Sony's Playstation 3, and Nintendo's Wii and Wii U game consoles (but not the Xbox One or Playstation 4, which use x64 processors). Its memory model is much more relaxed than x86-TSO. It, too, is defined using a model with a number of processors interacting with a storage subsystem, but there are two major differences. Firstly, a processor does not execute its instruction stream in-order; reads to distinct locations may be satisfied out of order and speculatively, and writes to distinct locations may be committed out of order. Secondly, the storage subsystem does not offer a total store order: writes to distinct locations may be propagated to different processors in different orders.

On the Power ISA, most forms of communication between processors require the use of fence instructions, possibly combined with *instruction dependencies*. The typical message passing pattern, where an object is initialized and then published by setting a flag or writing its address

to a shared variable, requires an `lwsync` (*lightweight synchronization*) fence on the sender side and an address dependency or a control-with-`isync` dependency, or an `lwsync`, on the receiver side between the read that notices the publication and the reads that inspect the object.

To regain SC, which is rarely necessary for inter-thread communication but which may be desirable when implementing a high-level programming language, the heavier `sync` instruction is necessary. For example, in Dekker's test where each thread sets its own flag and then checks the other thread's flag, under SC it is impossible that neither thread sees the other thread's flag. To rule out this case on Power, a `sync` must be inserted between the write and the read in each thread; an `lwsync` is insufficient.

On Power, read-modify-write operations are implemented using the *load-reserved* and *store-conditional* pair of instructions. The store-conditional instruction performs the store operation only if no write by another thread intervened after the preceding load-reserved instruction. In contrast to x86's LOCK prefix approach, the load-reserved and store-conditional pattern is not wait-free: other threads' interference may delay a thread's progress indefinitely.

The memory model of the ARM instruction set architecture, a very popular architecture for mobile devices, is very similar to Power's.

C/C++11 For the first time, the 2011 revision of the standard C and C++ programming language specifications [44, 45] specifies library constructs for thread creation, thread synchronization using mutexes, and so-called *atomic objects*, which allow well-defined concurrent access by multiple threads to the same memory location. It also includes a memory consistency model that specifies the behavior of memory as observed by multithreaded programs. An important rule is that conflicting concurrent accesses to the same non-atomic object (called *data races* by the standard) have undefined behavior. However, conflicting concurrent accesses of atomic objects have well-defined behavior, and atomic operations are provided with various levels of memory ordering and corresponding performance impact, ranging from variants that guarantee SC to variants called *relaxed*, that provide very weak guarantees and are intended to be implementable with minimal performance overhead. Other variants include *release*, *acquire*, and *consume*.

Passing an object between threads can be implemented using a release-acquire pair, where the write that publishes the object is an atomic write with release ordering and the read that notices the publication is an atomic read with acquire ordering. This guarantees that the code that inspects the object on the receiver side sees the initialization performed at the sender side. A slightly better-performing variant of this is the release-consume pair, where the read that notices the publication is an atomic read with consume ordering, and a *dependency* exists in the program between this read and the reads that inspect the object.

For relaxed atomics, roughly the only guarantees offered by the standard are 1) the absence of out-of-thin-air writes; more specifically, each write writes a value obtained by a sequence of program evaluations (possibly out of program order) from the relevant variables' initial values; and 2) coherence, meaning that for any particular atomic object, there is a total order on the writes to that object, consistent with the happens-before order, such that all threads see the writes to that object in that order. However, causal loops are allowed: in the program where each thread sets its flag if it sees the other thread's flag, it is allowed for both threads to see each other's flag.

Table 7 details the particular tasks pertaining to case studies described below.

8 Compilers and Runtime Systems

Since WP4 concerns reasoning about heterogeneous systems, we looked at compilers and runtime systems. Production-quality compilers are typically huge projects, often ranging in the hundreds of thousands to several millions of lines. Given the multitude of programming languages and

Case study	WP2	WP3
x86-TSO	T2.3	T3.1
SPARC-TSO	T2.3	T3.1
POWER ISA	T2.3	T3.1
C/C++11	T2.3	T3.1

Table 6: Case Studies: Weak memory models

hardware platforms, there is a plethora of compilers, though it is common for compilers to support multiple source languages and target architectures.

Regarding compilers, we decided to focus on C/C++ compilers because it is the language of choice when it comes to systems code. Although there have been proposals for type-safe languages for low-level systems programming, such as Cyclone [46], these have not been adopted widely. Moreover, these low-level type-safe languages are typically variants of C with a better type system, which after type-checking is typically ignored during compilation. For instance, the Cyclone compiler simply generates C code which is very similar to the Cyclone input modulo the type annotations. Therefore, even for such languages, looking at the compilation of C/C++ largely suffices.

Among the C/C++ compilers, we have singled out two compilers for further study, CompCert [49] and LLVM [47], because they are representative of two different kinds of compilers that we are interested in. On the one hand, CompCert is a verified compiler written in a mixture of Coq and OCaml, and is the only sizable verified compiler that exists. We found out, however, that the verification of CompCert has some limitations, which we will try to address in Tasks 4.1 and 4.3. On the other hand, LLVM is a production-quality open-source compiler that is widely deployed and used. In comparison to GCC [2], the other widely deployed open-source compiler, LLVM has a much clearer design, partly because it was originally designed by researchers at the University of Illinois.

We also looked at managed languages and their runtimes because of their relevance for tasks 4.2 and 4.4. The most widely used runtime systems are the Java Virtual Machine (JVM) and Microsoft's .NET platform. Both of these platforms support multiple languages: JVM mostly by accident, and .NET by design.

8.1 CompCert

In more detail, CompCert is a verified compiler from a sequential subset of the C language, CompCert C, to PowerPC, ARM, and x86 assembly language. Currently in version 2.0, it is written in a combination of Coq and OCaml (about 100,000 lines of Coq code, of which 40,000 lines of compiler code and statements of lemmas, 40,000 lines dedicated to the proofs of those lemmas, and another 20,000 blank lines or with documentation comments, as well as 30,000 lines of OCaml), and consists of 17 passes that gradually transform C programs into assembly code, and perform standard optimizations: register coloring, constant propagation, common subexpression elimination, function inlining, branch tunneling, and tailcall optimizations. The soundness proof of CompCert covers only sequential executions, though a forked-off project, CompCertTSO, developed a variant of the compiler (and of its soundness proof) that is sound under a concurrent semantics. Besides being the only sizable verified compiler, we chose it as a reference point because both its source code and the proof of its correctness are freely available and may be used for academic purposes. Moreover, our consortium has some local expertise on CompCert, as one of its members (Vafeiadis) took part in the CompCertTSO project and thus knows the code base fairly well.

The CompCert verified compiler makes use of a number of important techniques. First, it is structured as a sequence of many theoretically independent phases: while some phases, such as the intermediate type checking passes or the `Renumber` pass, are there only so as to

improve the effect of the optimizations performed in later passes, in terms of verification, they are independent. Second, for several of the complicated passes, CompCert relies on translation validation. For example, the register allocation algorithm is not verified, but its outcome is checked for correctness by a verified checker. Third, the proofs of all the compilation passes rely heavily on the semantics of each language being internally deterministic. This allows one to use "downward" or "forward" simulations—that is, in the same direction as the translation—rather than "upward"/"'backward" simulations.

One important limitation of CompCert is the very statement of correctness that has been proved. The correctness statement of CompCert says that given a complete program, if that is compiled by CompCert, then executing the generated assembly code according to the assembly semantics should only produce results that can be generated by executing the source program according to the CompCert C semantics. Clearly, this statement does not tell us anything about the typical usage of a compiler, which is to compile the various source files representing the various modules separately and then to link the resulting object files together. Moreover, it does not tell us anything about programs that are statically or dynamically linked to a library that might have been compiled by a different compiler, or that was perhaps even directly written in an assembly to start with. Therefore, in Task 4.1 we plan to develop a simple verified compiler that is compositionally correct.

8.2 LLVM

LLVM [47] is a compiler framework initially developed by researchers at Illinois, and is a widely deployed C compiler. The compiler itself is open source and distributed under a BSD-style license.

LLVM defines a low-level intermediate representation of programs (the LLVM IR), which is essentially three-address code. One particularly interesting aspect of this intermediate representation, which is relevant to Tasks 3.1 and 4.3, is its memory model. LLVM defines a memory model which is supposed to be a union of the Java and C11 memory models. It supports the following atomic access modes: `unordered`, `monotonic`, `acquire`, `release`, `acq_rel`, and `seq_cst`. In this list, `unordered` is supposed to correspond to Java's non-volative writes (which can read any value not happening after it); `monotonic` is supposed to correspond to the C11 atomic relaxed accesses (which in addition provides a per-location coherence guarantee); and the stronger atomicity types map exactly to the corresponding C11 notions. It is worth pointing out that LLVM does not have anything corresponding to the C11 consume atomics; these would therefore have to be compiled as though they were acquire atomics, thereby unnecessarily introducing a memory barrier. The core research challenge raised by the LLVM's memory model is to determine which of the standard sequential compiler optimizations are sound and which are not under the LLVM model.

8.3 JVM and JNI

The JVM (in particular, Oracle's reference implementation) and is the platform for executing Java programs and contains a Java byte code interpreter, a just-in-time compiler, a garbage collector, and the native method interface that allows interoperation between Java and C/C++. The JVM itself is written in a combination of C++ and assembly code.

JVM is an example of a program that relies on the memory model of the platform it runs on in interesting ways. In particular, an interesting aspect of the JVM with respect to weak memory is the way it deals with data races in the input Java program. To maximize performance, it must interpret or JIT-compile memory accesses in the input program using the least expensive possible memory accesses of the target platform. On the other hand, the JVM must protect its own integrity; in particular, it must ensure that, when interpreting or JIT-compiling a dereference of a Java object reference, the corresponding pointer points to a properly initialized

representation of a Java object. These two goals may conflict with each other: consider a Java program where one thread creates an object and writes a reference to that object into some static field, and another thread races to read this static field and use the object, e.g. by executing `System.out.println(x.getClass().getName());`. The interpretation or JIT-compilation of Java object creations and field accesses must be such that the accesses of the object by the second thread do not see an incompletely initialized state of the object. Existing approaches for reasoning about weak memory models do not support reasoning about patterns such as the one appearing in the Oracle JVM sketched above. Therefore, the Oracle JVM serves as a useful case study for the research on reasoning about weak memory models to be conducted in this project.

The Java native method interface (JNI) is rather complex platform-agnostic specification of how Java and C/C++ can interoperate. It allows methods of Java classes to be implemented in C/C++ and provides various ways to access Java data structures and call Java methods from C/C++. JNI itself is an improvement over a number of earlier interfaces (JDK 1.0 native method interface, Netscape's Java Runtime Interface, Microsoft's Raw Native Interface, and the Java/COM interface) that aims to achieve not only efficiency but also binary compatibility across JVM implementations.

An important reason why JNI is complicated is because it has to ensure that the C code respects any implicit assumptions made by the JVM implementation. For example, to call a Java method from C, one must first compute a "method identifier" for the method to be called. This is achieved by querying the JVM environment giving as arguments the name of a method as a string and an appropriate textual representation of the argument types. (The latter are required to resolve method overloading.) Next, given the method id, the method may be called. Naturally, if the C code wants to call the same method multiple times, it can remember its method identifier, and avoid recomputing it at each step. For this, however, to work correctly, the C code must ensure that there is a live reference to an object of the class defining the method visible to the JVM because otherwise the JVM may decide to unload the class, and so the method identifier will no longer be valid.

8.4 .NET

As Microsoft's .NET framework is very similar to JVM, we will focus on one crucial thing that is better supported in .NET than in JVM—that is, multiple languages. While multiple languages have been compiled to the Java bytecode and run on the JVM, the JVM's ability to handle multiple languages is largely accidental, whereas .NET was specifically designed to support multiple managed languages. There are over twenty programming languages that compile directly to .NET including Microsoft's C# (an object-oriented language similar to Java), Visual Basic, and F# (a higher-order functional programming language).

The main way of achieving this is via the Common Language Runtime (CLR) and its Common Intermediate Language (CIL). The latter is analogous to Java bytecode but supports an advanced type system that is common to all the languages that map onto CIL. One can think of this common type system as a union of the type systems of the various languages mapping onto CIL, though in practice the type systems of several of those languages have evolved to match closely the common type system, as this simplifies inter-language calling conventions. The specifics of the type system are not all that important for our purpose, but the important take-home message for Task 4.4 is that the state of the art in supporting interoperability between multiple managed languages is for them to have a common underlying type system.

Table 8.4 details the particular tasks pertaining to case studies described below.

References

[1] Erlang otp library. http://www.erlang.org/doc/design_principles/users_guide.html.

Case study	WP2	WP3	WP4
8.1 CompCert			
Renumber			T4.1, T4.4
Simple verifier compositionally correct compiler			T4.1, T4.4
8.2 LLVM			
Atomic access modes:	T2.3	T3.1	T4.3
8.3 JVM and JNI			
OracleJVM			T4.2, T4.4
JNI			T4.2, T4.4

Table 7: Case Studies: Compilers and runtime systems

[2] GCC compiler. gcc.gnu.org.

[3] Riak crdt implementation. https://github.com/basho/riak_dt.

[4] Riak key-value store. http://basho.com/products/riak-overview/.

[5] User-level RCU. http://lttng.org/urcu.

[6] A.-R. Adl-Tabatabai, T. Shpeisman, and J. Gottschlich. Draft specification of transactional language constructs for c++ (ver 1.0), 2009. *https://sites.google.com/site/tmforcplusplus/*.

[7] A.-R. Adl-Tabatabai, T. Shpeisman, and J. Gottschlich. Draft specification of transactional language constructs for c++ (ver 1.1), 2013. *https://sites.google.com/site/tmforcplusplus/*.

[8] P. S. Almeida and C. Baquero. Scalable eventually consistent counters over unreliable networks. http://arxiv.org/abs/1307.3207, 2013.

[9] H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky. A programming language perspective on transactional memory consistency. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2013.

[10] H. Attiya, S. Hans, P. Kuznetsov, and S. Ravi. Safety of deferred update in transactional memory. In *ICDCS*, 2013. To appear.

[11] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. The potential dangers of causal consistency and an explicit solution (vision paper). In *SOCC*, 20012.

[12] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.

[13] A. Bieniusa, M. Zawirski, N. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. An optimized conflict-free replicated set. Technical Report 8083, INRIA, 2012.

[14] A. Bieniusa, M. Zawirski, N. M. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. Brief announcement: Semantics of eventually consistent replicated sets. In *DISC*, 2012.

[15] J. Boner, N. Bronson, G. Korland, A. Prokopec, K. Sankar, and D. S. P. Veentje. Scala stm. *http://nbronson.github.io/scala-stm/index.html*.

[16] D. Bovet and M. Cesati. *Understanding the Linux Kernel, 3rd ed.* O'Reilly, 2005.

[17] N. G. Bronson. Composable operations on high-performance concurrent collections. Ph.D. Dissertation, Stanford University. *http://purl.stanford.edu/gm457gs5369*, 2011.

[18] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. *SIGPLAN Not.*, 45(5):257–268, Jan. 2010.

[19] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. Transactional predication: high-performance concurrent sets and maps for stm. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 6–15, New York, NY, USA, 2010. ACM.

[20] S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv. Eventually consistent transactions. In *ESOP*, 2012.

[21] E. Cohen, W. Schulte, and S. Tobies. Local verification of global invariants in concurrent programs. In *CAV*, 2010.

[22] N. Conway, R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *SOCC*, 2012.

[23] L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: streamlining stm by abolishing ownership records. In *PPOPP*, pages 67–78, 2010.

[24] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. User-level implementations of read-copy update. Submitted, 2011.

[25] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *DISC*, pages 194–208, 2006.

[26] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, 2010.

[27] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE*, pages 97–114, 2004.

[28] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Electron. Notes Theor. Comput. Sci.*, 259:245–261, December 2009.

[29] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, pages 1–31, March 2012.

[30] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, 2007.

[31] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[32] A. Gotsman, N. Rinetzky, and H. Yang. Verifying concurrent memory reclamation algorithms with grace. In *European Symposium on Programming (ESOP)*, 2013.

[33] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPOPP*, pages 175–184, 2008.

[34] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPOPP*, pages 175–184, 2008.

[35] R. Guerraoui and M. Kapalka. *Principles of Transactional Memory*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool, 2011.

[36] J. He, C. Hoare, and J. Sanders. Data refinement refined. In *ESOP*, pages 187–196, 1986.

[37] J. He, C. Hoare, and J. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25(2):71 – 76, 1987.

[38] M. Herlihy, V. Luchangco, M. Moir, and W. N. S. III. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101, 2003.

[39] M. Herlihy and N. Shavit. *The art of multiprocessor programming.* 2008.

[40] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[41] P. W. Howard and J. Walpole. A relativistic enhancement to software transactional memory. In *HotPar*, 2011.

[42] P. W. Howard and J. Walpole. Relativistic red-black trees. Submitted, 2011.

[43] D. Imbs, J. R. G. de Mendívil, and M. Raynal. Brief announcement: virtual world consistency: a new condition for stm systems. In *PODC*, pages 280–281, 2009.

[44] ISO. *Programming Languages — C.* 2011. ISO/IEC 9899:2011.

[45] ISO. *Programming Languages — C++.* 2011. ISO/IEC 14882:2011.

[46] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[47] C. Lattner and V. S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.

[48] K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In *ESOP*, 2009.

[49] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[50] M. Lesani, V. Luchangco, and M. Moir. Putting opacity in its place. In *WTTM*, 2012.

[51] C. Li, D. Porto, A. Clement, R. Rodrigues, N. Preguiça, and J. Gehrke. Making geo-replicated systems fast if possible, consistent when necessary. In *OSDI*, 2012.

[52] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.

[53] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, pages 313–328, 2013.

[54] R. Love. *Linux Kernel Development, 3rd ed.* Addison Wesley, 2010.

[55] V. J. Marathe, W. N. S. III, and M. L. Scott. Adaptive software transactional memory. In *DISC*, pages 354–368, 2005.

[56] P. E. McKenney. Exploiting deferred destruction: an analysis of read-copy-update techniques in operating system kernels. PhD Thesis. OGI, 2004.

[57] P. E. McKenney. Structured deferral: synchronization via procrastination. *Commun. ACM*, 56(7):40–49, July 2013.

[58] P. E. McKenney and J. D. Slingwine. Read-copy update: using execution history to solve concurrency problems. In *PDCS*, 1998.

[59] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96*. ACM, 1996.

[60] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide, 2nd Edition*. Artima Incorporation, USA, 2nd edition, 2011.

[61] P. O'Hearn. Resources, concurrency and local reasoning. *TCS*, 2007.

[62] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOLs*, 2009.

[63] C. Papadimitriou. *The theory of database concurrency control*. 1986.

[64] W. Penninckx, J. T. Mühlberg, J. Smans, B. Jacobs, and F. Piessens. Sound formal verification of Linux's USB BP keyboard driver. In *NFM 2012*, volume 7226 of *LNCS*, pages 210–215, Heidelberg, 2012. Springer.

[65] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia. A commutative replicated data type for cooperative editing. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, pages 395–403, 2009.

[66] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *DISC*, pages 284–298, 2006.

[67] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel Distrib. Comput.*, 71(3), 2011.

[68] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *PLDI*, 2011.

[69] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report 7506, INRIA, 2011.

[70] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *SSS'11*.

[71] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213. ACM, 1995.

[72] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.

[73] B. Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1997.

[74] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

[75] J. Triplett, P. E. McKenney, and J. Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. In *USENIX ATC*, 2011.

[76] V. Vafeiadis. Automatically proving linearizability. In *CAV*, 2010.

[77] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, 2007.