Robust Shared Objects for Non-Volatile Main Memory

Ryan Berryhill[1], Wojciech Golab[2], and Mahesh Tripunitara[3]

1 **University of Toronto, Toronto, Canada**
`ryan@eecg.utoronto.ca`
2 **University of Waterloo, Waterloo, Canada**
`wgolab@uwaterloo.ca`
3 **University of Waterloo, Waterloo, Canada**
`tripunit@uwaterloo.ca`

Abstract

Research in concurrent in-memory data structures has focused almost exclusively on models where processes are either reliable, or may fail by crashing permanently. The case where processes may recover from failures has received little attention because recovery from conventional volatile memory is impossible in the event of a system crash, during which both the state of main memory and the private states of processes are lost. Future hardware architectures are likely to include various forms of non-volatile random access memory (NVRAM), creating new opportunities to design robust main memory data structures that can recover from system crashes. In this paper we advance the theoretical foundations of such data structures in two ways. First, we review several known variations of Herlihy and Wing's linearizability property that were proposed in the context of message passing systems but also apply in our NVRAM-based model, we discuss the limitations of these properties with respect to our specific goals, and we propose an alternative correctness condition called *recoverable linearizability*. Second, we discuss techniques for implementing shared objects that satisfy such properties with a focus on wait-free implementations. Specifically, we demonstrate how to achieve different variations of linearizability in our model by transforming two classic wait-free constructions.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases non-volatile main memory, concurrency, recovery, data structures

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.20

1 Introduction

Shared data structures are essential building blocks for modern operating systems and applications, which are empowered almost exclusively by multi-core hardware platforms. Although the multi-core revolution has propelled research in this area to new heights over the last decade, questions pertaining to specifying and implementing concurrent data structures were considered in the literature long before thread-level parallelism became mainstream. Dijkstra's pioneering work on concurrent programming dates back to 1965 [10], followed by a series of seminal papers on inter-process communication, wait-free synchronization, and linearizability [14, 16, 22, 23]. The fundamental abstractions introduced in this body of work have been studied widely in the context of various theoretical models of shared memory computation that capture precise assumptions regarding the synchrony and reliability of processes, as well as the set of primitive operations available for accessing memory. In particular, recent research has paid close attention to asynchronous models, in which there is no bound on the amount of time a process takes to transition to its next step, or to complete a memory operation. This assumption reflects in a meaningful way the behavior of modern memory hierarchies, in which different media (e.g., L1 cache vs. L2 cache vs. main memory) incur vastly different and often unpredictable access latencies, as well as the effect of the operating system (e.g., via preemption and interrupts) on the liveness of processes. Aside from capturing these important aspects of real world performance, asynchrony is also related to reliability in a precise way: algorithms that provide non-blocking progress properties (e.g., lock-freedom and wait-freedom) in an asynchronous environment with reliable processes continue to provide the same progress properties if crash failures are introduced. Informally speaking, this property holds because a process that crashes permanently at an arbitrary point in the execution of its algorithm is indistinguishable to the other processes from one that is merely very slow.

Owing to its simplicity and intimate relationship with asynchrony, the crash failure model is almost ubiquitous in the treatment of non-blocking shared memory algorithms and message passing protocols [4, 7, 11, 14, 20]. In comparison, much less attention has been paid to crash-recovery models, in which a failed process may be resurrected after a crash failure. For example, in the message passing paradigm a process may crash and recover state information either from its private stable storage or from another process on a different node [3]. Although similar techniques are in principle applicable in the shared memory paradigm of computation, they are poorly matched to modern multi-core architectures with volatile SRAM-based caches and DRAM-based main memories [29]. Any state stored in main memory is lost entirely in the event of a system crash or power loss, and recording recovery information in non-volatile secondary storage (e.g., on a hard disk drive or solid state drive) imposes overheads that are unacceptable for performance-critical tasks, such as synchronizing threads inside the operating system kernel.

In this paper, we consider correctness properties and implementation techniques for data structures intended for future shared memory architectures that incorporate *non-volatile random-access memory (NVRAM)* – a form of main memory that promises to marry performance comparable to DRAM with the high density and persistence of secondary storage. We assume that NVRAM is accessed using memory operations (e.g., reads, writes, and read-modify-write primitives), similarly to ordinary DRAM, and that such operations can be made both atomic and durable through appropriate extensions to conventional caching and memory ordering mechanisms [9, 18, 25, 28, 30]. Under these assumptions, concurrent data structures such as stacks, queues, and trees may reside directly in NVRAM and algorithms for accessing them may follow conventional techniques for synchronization. However, conventional techniques do not address the problem of recovering such structures following a failure, such as may occur when a multiprocessor suffers a power outage, leading to the loss of all volatile state including the program counter and other vital CPU registers.

Our technical contributions with respect to robust shared objects for multiprocessors that incorporate NVRAM are the following:

1. We refine the conventional abstract model of a shared memory multiprocessor by introducing non-volatility.
2. We survey known correctness properties proposed for shared objects in models with crash and crash-recovery failures, discuss their limitations, and propose an alternative property we call *recoverable linearizability* (or *R-linearizability* for short).
3. We explore techniques for transforming ordinary linearizable implementations into R-linearizable ones, with a special focus on wait-freedom. Our discussion uses as examples Herlihy's universal wait-free construction [14] and a classic wait-free implementation of MRSW registers from SRSW registers [15].

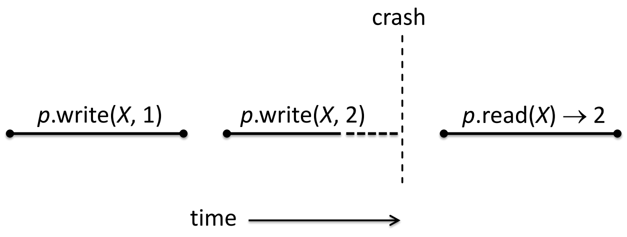

Figure 1 Example execution in which process p writes 1 to object X, then begins writing 2 to X, fails due to a system crash before the write returns, and then reads 2 from X.

2 Related Work

Constructing robust shared objects for NVRAM requires two ingredients: a correctness property that provides meaningful guarantees in the presence of failures, and a set of algorithmic techniques that leverage the non-volatility of the storage medium for recovery. Conventional relational databases provide both, namely serializability for correctness and write-ahead logging for recovery (e.g., ARIES [26]), but perform orders of magnitude slower than main memory data structures owing to their internal complexity as well as their use of a centralized recovery log in secondary storage. Coburn et al. propose NV-Heaps as an alternative method of providing transactional access to persistent shared objects [8]. NV-Heaps use NVRAM directly to store both object state and recovery information in the form of *operation descriptors*, which are similar to recovery logs but more fine-grained. Like conventional databases, NV-Heaps use locks for concurrency control and perform recovery by analyzing the operation descriptors while execution of new transactions is temporarily suspended. Mnemosyne is another transactional interface to NVRAM that uses lock-based concurrency control and log-based recovery [35]. Venkataraman et al. define Consistent and Durable Data Structures (CDDSs), which provide lock-free access to readers but rely on mutual exclusion to synchronize writers [33]. These structures are linearizable in failure-free executions, and any updates that are interrupted by a crash failure are discarded on recovery.

Aside from techniques targeted specifically at NVRAM, related work includes methods for recovering process state from stable storage and for dealing with unreliable memory. Schlichting and Schneider consider the problem of restarting a process that halts due to a processor failure by defining *fault-tolerant actions* that leverage stable storage to persist program state [32]. This work focuses on fault tolerance and considers limited interprocess communication through multi-reader single-writer shared state variables. Several papers consider computation with unreliable memory: Afek et al. consider the consensus problem in this general context [1], Moscibroda and Oshman focus on mutual exclusion [27], and Jayanti et al. propose implementations of shared objects from unreliable base objects [20]. In contrast to these techniques, which break if the number of corruptions exceeds a specified bound, Hoepman et al. [17] as well as Johnen and Higham [21] propose self-stabilizing shared objects that can tolerate any number of memory failures. These objects guarantee wait-free progress, and also ensure that operations return correct values except possibly during the period of instability immediately following a failure.

As regards correctness properties, specifically consistency properties, the dominant ideas in research have long been *serializability* in the context of databases [5], and *linearizability* in the context of in-memory shared objects [16]. Informally speaking, both require that actions – transactions or operations on objects – appear to take effect instantaneously in some serial order. Linearizability further constrains this order so that if operation op_1 *happens before* operation op_2 (i.e., op_1 ends before op_2 begins), then op_1 should precede op_2 in the serial order. *Strict serializability* imposes an analogous constraint on the serial order of transactions. Serializability naturally accommodates crash-recovery failures in the following sense: a transaction interrupted by a failure can simply be aborted and excluded from the serial order. In contrast, linearizability (defined formally in Section 3) has no notion of an aborted or failed operation, and requires that a process finish one operation before it invokes the next. Figure 1 illustrates an execution that is outside the scope of such a model because a process fails in the middle of an operation, then recovers and invokes another operation without completing the previous one.

Frølund et al. address the technicality illustrated in Figure 1 by treating the crash of a process as a response, either successful or unsuccessful, to the interrupted operation [12]. A successful response means that the operation takes effect at some point between its invocation and the crash failure, and an unsuccessful response means that the operation does not take effect at all. This correctness property is stated in [12] as an extension of Lamport's atomicity property for read/write registers [22, 23], and generalized to arbitrary object types by Aguilera and Frølund as *strict linearizability* [2]. The same idea is used by Saito et al. in FAB, a fault-tolerant distributed storage system [31].

Aguilera and Frølund show that strict linearizability has an interesting property in the context of shared memory: it precludes wait-free implementations of multi-reader single-writer (MRSW) registers from single-reader single-writer (SRSW) registers [2]. Intuitively, this is because the effect of a write operation on the implemented object can only be made visible to other processes by a non-atomic series of operations on the single-reader base objects. As a result, when a write is interrupted by a crash, it is sometimes possible for a subsequent read to return either the old or the new value of the implemented object, depending on the identity of the reader. This leads to a scenario where the write appears to take effect after the crash because one read returns the old value and a later read returns the new value.

Guerraoui and Levy propose two correctness properties for read/write registers simulated using message passing in a crash-recovery model [13]. *Persistent atomicity* is similar to strict linearizability, but allows an operation interrupted by a failure to take effect before the subsequent invocation of the same process, possibly after the failure. *Transient atomicity* relaxes this criterion even further and allows an interrupted operation to take effect before the subsequent write response of the same process. Although the intent underlying Guerraoui and Levy's definitions was to explore trade-offs between performance and consistency, their properties are quite relevant in the context of shared objects for NVRAM, particularly wait-free implementations that employ *helping mechanisms* whereby an operation invoked by a process p may take effect by the action of another process q after p fails. In contrast, strict linearizability forbids this behavior. Censor-Hillel, Petrank and Timnat recently formalized the concept of helping and showed that without it, certain types of objects lack wait-free linearizable implementations in a conventional shared memory model [6].

Correctness properties for shared objects are easier to reason about when they are *local*, meaning that a collection of objects satisfy a given property P if and only if every object in the collection individually satisfies P [16]. Locality makes it possible to implement and verify shared objects independently, which benefits modularity and concurrency. It is known that linearizability, strict linearizability, and strict serializability are local properties, while ordinary serializability is not. As we explain in Section 4, persistent and transient atomicity are also not local. Vitenberg and Friedman formulate a number of general theorems that can be used to deduce the locality (or lack thereof) of various correctness properties for shared objects [34], however these theorems are proved in a conventional model similar to Herlihy and Wing's, in which a process must complete one operation before it invokes another. Hence, these theorems are not applicable directly in our more general model.

3 Model

Our model is closely based upon Herlihy and Wing's [16].

Processes and objects. We consider N asynchronous *processes*, denoted $p_0, p_1, ..., p_{N-1}$, that communicate by applying operations on shared *base objects* that support atomic operations such as reads, writes, and read-modify-write primitives. Base objects can be used to construct more complex *implemented objects*, such as queues and stacks, by defining *access procedures* that simulate each operation on the implemented object using operations on base objects. Base objects can be *volatile* or *non-volatile*, which determines their behavior during a failure. The state of a process is a collection of *private variables*, including a *program counter*. We consider only one type of failure, called a *system crash* (or crash for short), which resets all volatile base objects as well as the private variables of all processes to their initial values, but preserves the values of all non-volatile base objects. Following a crash a process may either halt permanently or resume its execution (i.e., recover).

Steps and histories. We model the interaction of processes with implemented objects using *steps* and *histories*. There are three types of steps: (1) an invocation step, denoted (INV, p, X, op), represents the invocation by process p of operation op on implemented object X; (2) a response step, denoted (RES, p, X, ret), represents the completion by process p of the last operation it invoked on object X, with response ret; (3) a crash step, denoted (CRASH), denotes a system crash. We include explicit crash steps to accommodate strict linearizability (defined formally in Section 4), but we do not use explicit recovery steps; a process recovers implicitly following a crash by taking an invocation step.

A history H is a sequence of steps, possibly involving multiple processes and implemented objects. For a given history H, we denote by $H|p$ the projection of H onto the steps of process p. Similarly, we denote by $H|O$ the projection of H onto the steps involving implemented object O. We adopt the convention that both $H|p$ and $H|O$ retain all crash steps in H. A response step is *matching* with respect to an invocation step s by a process p on object X in a history H if it is the first response step by p on X that follows s in H, and it occurs before p's next invocation (if any) in H.

Operations. For any history H and any process p, an operation by p in H comprises an invocation step and its matching response, if it exists. An operation is *complete* if it has a matching response step, and *pending* otherwise. Given two operations op_1 and op_2 in a history H, we say that op_1 *happens before* op_2, denoted by $op_1 <_H op_2$, if op_1 has a matching response that precedes the invocation step of op_2 in H. If neither $op_1 <_H op_2$ nor $op_2 <_H op_1$ holds then we say that op_1 and op_2 are *concurrent* in H.

Properties of histories. A history H is *sequential* if no two operations in it are concurrent. Two histories H and H' are *equivalent* if for every process p, $H|p = H'|p$ holds. Every history H must be *well-formed*, meaning that for each process p two conditions hold: (1) each response step in $H|p$ is immediately preceded by an invocation step for which the response is matching, and (2) each invocation step in $H|p$, except possibly the last one, is immediately followed by a matching response or by a crash step. Note that p may have multiple pending operations in $H|p$, in contrast to Herlihy and Wing's model, where at most the last operation may be pending.

Sequential specifications. Every implemented object O has a *sequential specification* that defines its allowed behaviors and is expressed as a set of possible sequential histories over O. A sequential history H is *legal* if for every implemented object O accessed in H, $H|O$ belongs to the sequential specification of O.

4 Correctness Properties

In this section we consider correctness properties for shared objects in NVRAM. Our goal is to identify a small set of candidate properties that could describe the behavior of a variety of shared object implementations. Specifically, we are interested in variations of Herlihy and Wing's linearizability property as it is widely adopted for shared objects in conventional shared memory models. We will give formal definitions of some of the properties discussed in Section 2, discuss their limitations, and then propose an alternative correctness property.

Linearizability itself, as defined by Herlihy and Wing [16], can be formalized in our model but only for histories that are free of crash steps. Given such a history H, we first define its *completion* H' by appending matching responses for a subset of pending operations, and finally removing any remaining pending operations. Appending the responses rather than inserting them between steps of H ensures that H and H' share the same "happens before" relation (i.e., $<_H = <_{H'}$).

▶ **Definition 1** (Linearizability). A finite history H that does not contain any crash steps is linearizable if it has a completion H' and there exists a legal sequential history S such that:
L1. H' is equivalent to S; and
L2. $<_H \subseteq <_S$ (i.e., if $op_1 <_H op_2$ and both ops. appear in S then $op_1 <_S op_2$).

To formalize strict linearizability we introduce a *strict completion* H', which is obtained from H by inserting matching responses for a subset of pending operations after the operation's invocation and before the next crash step (if any), and finally removing any remaining pending operations and crash steps. Inserting responses in this manner may alter the "happens before" relation but guarantees that $<_H \subseteq <_{H'}$ (i.e., if $op_1 <_H op_2$ then $op_1 <_{H'} op_2$).

▶ **Definition 2** (Strict linearizability). A finite history H is strictly linearizable if it has a strict completion H' and there exists a legal sequential history S such that:
SL1. H' is equivalent to S; and
SL2. $<_{H'} \subseteq <_S$ (i.e., if $op_1 <_{H'} op_2$ and both ops. appear in S then $op_1 <_S op_2$).

Referring to H' in clause **SL2** ensures that any operation invoked before a crash in H and completed in H' happens before any operation invoked after the crash. As an example, consider the history H corresponding to Figure 1, where $\perp$ denotes the response of a write:

(INV, p, X, write(1)), (RES, p, X, $\perp$), (INV, p, X, write(2)), (CRASH),
(INV, p, X, read()), (RES, p, X, 2)

A strict completion H' of H can be obtained by inserting a matching response for the write of 2 immediately before the crash step, and then removing the crash step:

(INV, p, X, write(1)), (RES, p, X, $\perp$), (INV, p, X, write(2)), (RES, p, X, $\perp$),
(INV, p, X, read()), (RES, p, X, 2)

The legal sequential history S that satisfies Definition 2 with respect to H and H' is H' itself, and thus H is strictly linearizable.

Although strict linearizability is an attractive property, Aguilera and Frølund show that it is somewhat restrictive as it forbids wait-free implementations of multi-reader single-writer (MRSW) registers from single-reader single-writer (SRSW) registers [2]. In contrast, linearizability does allow such an implementation, which we discuss further in Section 5.2. Guerraoui and Levy's definitions are less restrictive in comparison as they allow operations interrupted by a failure to take effect after the failure, for example by the action of another process executing a helping mechanism in a wait-free implementation.

Persistent atomicity, which we refer to later on as *persistent linearizability*, can be formalized in our model as follows. Given a history H, a *persistent completion* H' is obtained from H by inserting matching responses for a subset of pending operations after the operation's invocation and before the next invocation step of the same process, and finally removing any remaining pending operations and crash steps.

▶ **Definition 3** (Persistent linearizability). A finite history H is persistently linearizable if it has a persistent completion H' and there exists a legal sequential history S such that conditions **SL1** and **SL2** from Definition 2 hold.

Defining transient atomicity formally in our model presents two technical difficulties, both related to Guerraoui and Levy's definition of a *weak completion* in which a matching response can be added for a pending operation anywhere after the invocation and "before the subsequent *write* reply of the same process." First, it is not obvious how to generalize this concept to arbitrary object types, which may not even support a write operation. Second, the weak completion may contain complete operations invoked by the same process that overlap, even if we restrict our attention to a single object. This not only violates the well-formedness property defined in Section 3 for histories, but also allows operations executed by the same process to take effect in an order different from the order of invocation. (The model in [13] uses a slightly different well-formedness property but the same issues arise.) This anomaly, which we call *program order inversion*, complicates reasoning about the behavior of implemented objects and goes against the intuition underlying Herlihy and Wing's model, in which one process may have at most one pending operation at any given point in time. On the other hand, some reordering among operations may be justifiable in our crash-recovery model for the following reason: if an operation *op* by process p on object X is interrupted by a crash failure, process p should be free to resume execution and invoke an operation on some other object Y independently of any steps on X, by p or any other process, that may cause p's interrupted operation to take effect later on. We refer to this requirement as *independent recovery*, and suggest that it follows naturally from the *nonblocking* property of linearizability: "processes invoking totally-defined operations are never forced to wait" [16].

A more fundamental drawback of both persistent and transient atomicity is the lack of locality, which we consider essential. In both cases the placement of a matching response for a pending operation is constrained by other operations in a manner that may become more restrictive when single-object histories are merged to create a history in which multiple objects are accessed. Figure 2 illustrates a specific example of this problem. Letting H denote the illustrated history, $H|X$ satisfies persistent atomicity because when p's write$(Y, 1)$ is out of the picture, the remaining operations are permitted to take effect in the following order:

p.write$(X, 1)$, q.read$(X) \rightarrow 1$, p.write$(X, 2)$, p.read$(X) \rightarrow 2$

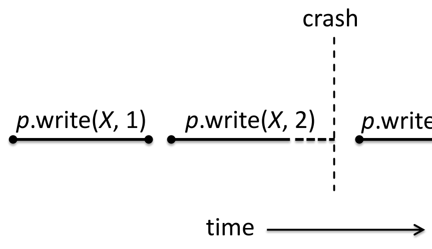

Figure 2 Example execution involving processes p, q and objects X, Y. A failure interrupts p's second write operation before it returns a response.

In particular, p's write$(X, 2)$ is permitted to take effect after q's read(X) as long as it takes effect before the invocation of p's read(X). Similarly $H|Y$ satisfies persistent atomicity because it comprises only a single write operation, namely p's write$(Y, 1)$. However, H itself lacks persistent atomicity because p's write$(X, 2)$ would be forced to take effect before p's write$(Y, 1)$, and hence before q's read(X), which returns 1. An analogous argument shows that transient atomicity is not local because p's write$(X, 2)$ would be forced to take effect before the response of p's write$(Y, 1)$, and hence before q's read(X).

To remedy the technical issues surrounding persistent and transient atomicity, we propose an alternative property called *recoverable linearizability* (or R-linearizability for short) that accommodates arbitrary object types and guarantees locality. Our property is formalized by first defining an appropriate completion procedure, similarly to strict linearizability and persistent atomicity, but deals with the "happens before" relation differently. Given a history H, a *recoverable completion* H' is obtained from H in exactly the same manner as a strict completion. As we discuss shortly, the strictness of the completion does not prevent an operation from taking effect after a failure that interrupts it. However, it does simplify the proof of Theorem 6 later on. Next, we define a precedence order on operations to prevent program order inversion for operations applied to the same object.

▶ **Definition 4.** Given a history H, the *invoked before* relation over pairs of operations in H, denoted $\ll_H$, is an irreflexive partial order defined as follows: if op_1 and op_2 are operations invoked by the same process p on the same object X, and the invocation step of op_1 precedes the invocation step of op_2 in H, then $op_1 \ll_H op_2$.

We use both $<_H$ and $\ll_H$ to constrain the order in which operations appear to take effect:

▶ **Definition 5** (Recoverable linearizability)**.** A finite history H is R-linearizable if it has a recoverable completion H' and there exists a legal sequential history S such that:
RL1. H' is equivalent to S;
RL2. $<_H \subseteq <_S$ (i.e., if $op_1 <_H op_2$ and both ops. appear in S then $op_1 <_S op_2$); and
RL3. $\ll_H \subseteq <_S$ (i.e., if $op_1 \ll_H op_2$ and both ops. appear in S then $op_1 <_S op_2$)

Note that clause **RL2** refers to H and not H', in contrast to clause **SL2** in Definition 2. This ensures that the placement of matching responses in the construction of the recoverable completion, which preserves well-formedness, does not impose undesirable constraints on the order in which operations may appear to take effect. Clause **RL3** compensates for this by disallowing program order inversion at the level of individual objects. Two operations invoked by the same process on different objects may still take effect in an order different from their invocation order, which enables independent recovery. As we show later on in Theorem 7 and Section 5.2, R-linearizability is a local property similarly to strict linearizability, and yet is weak enough to permit a wait-free implementation of MRSW registers from SRSW registers in contrast to strict linearizability.

To illustrate R-linearizability in action, a recoverable completion H' for the history H shown in Figure 2 can be constructed as follows:

(INV, p, X, write(1)), (RES, p, X, $\perp$), (INV, p, X, write(2)), (RES, p, X, $\perp$), (INV, p, Y, write(1)), (RES, p, Y, $\perp$), (INV, q, X, read()), (INV, p, X, read()), (RES, q, X, 1), (RES, p, X, 2)

The precedence constraints imposed by clause **RL2** on the legal sequential history S are the transitive closure of the following:

$$p.\text{write}(X,1) <_S p.\text{write}(X,2),\ \ p.\text{write}(X,1) <_S p.\text{write}(Y,1)$$
$$p.\text{write}(Y,1) <_S q.\text{read}(X),\ \ p.\text{write}(Y,1) <_S p.\text{read}(X)$$

Clause **RL3** imposes the additional constraint $p.\text{write}(X,2) <_S p.\text{read}(X)$. The legal sequential history S that satisfies Definition 5 with respect to H and H' is the following:

(INV, p, X, write(1)), (RES, p, X, $\perp$), (INV, p, Y, write(1)), (RES, p, Y, $\perp$), (INV, q, X, read()), (RES, q, X, 1), (INV, p, X, write(2)), (RES, p, X, $\perp$), (INV, p, X, read()), (RES, p, X, 2)

Thus, the history H shown in Figure 2 is recoverably linearizable or R-linearizable.

In our model it can be shown that strict linearizability is strictly stronger than persistent linearizability, which is strictly stronger than R-linearizability.

▶ **Theorem 6.** *Let H be a history. If H is strictly linearizable (Definition 2) then H is also persistently linearizable (Definition 3), and if H is persistently linearizable then H is also recoverably linearizable (Definition 5). Furthermore, there exists a history that is R-linearizable but not persistently linearizable, and there exists a history that is persistently linearizable but not strictly linearizable.*

Finally, we consider locality in Theorem 7, whose detailed proof is omitted due to lack of space.

▶ **Theorem 7** (locality)**.** *A history H is R-linearizable if and only if, for every object X accessed in H, $H|X$ is R-linearizable.*

Proof Sketch. The "only if" direction follows easily, and so we focus on the "if" direction. Informally speaking, it suffices to define for each object X a linearization point for each operation in $H|X$, such as a base object step at which the operation appears to take effect, and then order the operations in H according to the same linearization points. The main technicality is to show that the linearization points chosen initially are still applicable after the projections $H|X$ are merged together. This point follows easily as long as the definitions of the linearization points are independent of operations on other objects – a property that holds by design in R-linearizability. For a complete operation, clause **RL2** of Definition 5 restricts the linearization point to occur between its own invocation and response, and does not refer to any other operation. For a pending operation, clause **RL3** of Definition 5 restricts the linearization point to occur between its own invocation and the response of the next operation by the same process on the same object (see Definition 4), and also does not refer to operations on any other object. ◀

5 Implementations

In this section we consider techniques for implementing shared objects that are robust against failures in our crash-recovery model in the sense of providing non-blocking progress guarantees in addition to one (or more) of the safety properties formalized in Section 4. A very general but naive technique for constructing such implementations is to take an algorithm designed for the conventional shared memory model, make all program variables non-volatile, and have each process adopt a new and distinct ID on recovery after a crash failure. This approach circumvents enough of the technical issues discussed in Section 4 to make Herlihy and Wing's linearizability property applicable directly, but suffers from two problems. First, it opens the door to program order inversion among operations applied by the same process under different IDs. Second, unless the number of crash failures in a history is bounded, the process IDs grow without bound, leading to a blowup in space complexity for algorithms that store process IDs in variables or use them to index arrays. Merritt and Taubenfeld show that such a blowup is unavoidable for many fundamental problems [24].

Another general strategy for constructing robust objects is to record housekeeping information in NVRAM as processes execute operations on objects, similarly to write-ahead logging in a database, and use it on recovery to repair any operation that was interrupted by a failure. This strategy requires additional writes to NVRAM during failure-free operation, and in practice relies on a specialized recovery procedure that is executed automatically upon recovery while ordinary operations on objects are temporarily suspended (e.g., as in NV-Heaps [8]). Such a recovery procedure is outside the scope of our model, and also implies the use of mutual exclusion to isolate recovery actions from ordinary operations, which is counter to our goal of non-blocking progress.

As an alternative to renaming processes and database-style logging, we explore in this section the following technique: start with a linearizable implementation for ordinary volatile shared memory, make all variables non-volatile, and then modify the algorithm as needed to achieve the desired correctness properties. We focus specifically on two known wait-free implementations: Herlihy's universal construction [14], and a construction of atomic multi-reader single-writer (MRSW) registers from atomic single-reader single-writer (SRSW) registers [15]. In our subsequent discussion of these constructions, we define time complexity as the number of base object operations executed per implemented operation between its invocation and either its response or a failure, whichever occurs first.

5.1 Herlihy's Wait-Free Universal Construction

Herlihy [14] proposes a construction of wait-free linearizable objects that is *universal* – it can implement any shared object type. The construction, which we reproduce in Figure 3, works as follows. Each process, when it wants to apply an operation on the implemented object, attempts to have a *cell* structure representing its invocation threaded onto a linked list of such structures. This list determines both the subset of operations that have taken effect and their respective linearization order. The invocation is passed to the access procedure Universal as a structure of type *INVOC*, which encodes the operation to be applied and its arguments. At line 2, the process announces its invocation to others by storing a pointer to the corresponding cell at a dedicated element of the array *Announce*. Each process also has a dedicated element in the array *Head*, and uses a scan of this array at lines 3–5 to identify a cell near the end of the linked list. The *max* operator at line 4 compares cell structures by their *seq* member, which is a sequence number indicating the position of a cell in the linked list. Arrays *Announce* and *Head* are initialized with all elements pointing to a special anchor cell, which represents the start of the list and has a sequence number of one.

Base objects:
Announce, *head*: array[0..N-1, 0..N-1] of
pointer to cell, each element initialized to the address of the anchor cell

Function Universal(*what*: INVOC) for process p

```
1  $mine$: cell := [$seq$: 0, $inv$: $what$,
     $new$: consensus object, $before$: NULL,
     $after$: consensus object]
2  $Announce[p]$ := $mine$
3  foreach process ID $q$ do
4      $Head[p]$ := max($Head[p]$, $Head[q]$)
5  end
6  while $Announce[p].seq = 0$ do
7      $c$: pointer to cell := $Head[p]$
8      $help$: pointer to cell :=
         $Announce[(c.\mathrm{seq} \bmod N) + 1]$
9      if $help.seq = 0$ then
10         $prefer$ := $help$
11     else
12         $prefer$ := $Announce[p]$
13     end
14     $d$ := decide($c.after$, $prefer$)
15     decide($d.new$, apply ($d.inv$, $c.new.state$))
16     $d.before$ := $c$
17     $d.seq$ := $c.seq + 1$
18     $Head[p]$ := $d$
19 end
20 $Head[p]$ := $Announce[p]$
21 return $Announce[p].new.result$
```

Figure 3 Herlihy's universal wait-free construction for N processes [14].

The universal construction deals with concurrency using two consensus objects per cell: *after* is a pointer to the next cell in the list and deals with concurrent attempts to thread a cell onto the list; *new* is a structure that holds the state of the implemented object (in *new.state*) and the corresponding response (in *new.result*), and deals with concurrent attempts to determine the state transition for an invocation when the transition function denoted by "apply" at line 15 is nondeterministic.

For wait-freedom the construction uses a helping mechanism whereby a process attempting to thread a new cell onto the linked list at line 14 may act on a cell announced by another process, which is chosen at lines 8–13. This mechanism ensures that every cell that is announced is threaded onto the linked list in a bounded number of base object operations, as long as some process continues to take steps.

Let *UC* denote Herlihy's universal construction in our model with all base objects made non-volatile, and let *UC*′ denote the same implementation but with the *Announce* array made volatile. In the remainder of this section we show that *UC* provides R-linearizability but not persistent or strict linearizability, and that *UC*′ provides all three properties. Detailed proofs of correctness for Theorems 8–10 are omitted due to lack of space.

▶ **Theorem 8.** *Every finite history H of implementation UC is R-linearizable.*

Proof sketch. Intuitively, we must show that when an operation applied by a process p is interrupted by a failure, it is safe for p to abandon this operation and start executing procedure Universal from line 1 when it invokes its next operation. To that end, we prove that p's abandoned operation takes effect at most once, and moreover it never takes effect out of order with respect to any operation that p invokes after the failure on the same instance of the universal construction. The key to the proof is the observation that each time p executes procedure Universal, it overwrites its element of the *Announce* array at line 2, which has two implications. First, the cell associated with p's interrupted operation becomes inaccessible to future iterations of the helping mechanism, unless that cell has already been threaded onto the linked list. Second, any iteration of the helping mechanism that is acting on that cell and has already begun prior to p's execution of line 2, is doomed to fail if any other cell is threaded onto the linked list first. The detailed proof uses both points to establish R-linearizability for all histories of *UC*. ◀

▶ **Theorem 9.** *Implementation UC for $N \geq 2$ processes has a finite history H that is neither strictly or persistently linearizable.*

Proof sketch. We show that when an operation applied by a process p is interrupted by a failure, this operation may take effect after p's next invocation step by the action of another process that is participating in the helping mechanism. ◀

▶ **Theorem 10.** *Every finite history H of implementation UC' is strictly linearizable.*

Proof sketch. Extending the proof of Theorem 8, we show that when an operation applied by a process p is interrupted by a failure, its cell is either threaded before the failure or not at all, since the array *Announce* is volatile in UC'. ◀

▶ **Theorem 11.** *Implementations UC and UC' for N processes have time complexity $O(N)$.*

Proof. As in the original analysis [14] it can be shown that the while loop has at most $N + 1$ iterations during any execution of the procedure Universal. Furthermore, the loop at lines 3–5 has exactly N iterations. Since each loop iteration applies $O(1)$ base object operations, this implies the claimed time complexity. ◀

Our discussion of *UC* and UC', which are extensions of Herlihy's universal construction, shows that any object type can be implemented in a wait-free and R-linearizable manner in our crash-recovery model using base objects of types read/write register and consensus. Thus, consensus is universal in our model just as in Herlihy's [14]. Furthermore, UC' demonstrates that it is possible to achieve wait-freedom and strict linearizability (hence freedom from program order inversion) simultaneously in a construction that depends crucially on a helping mechanism. Specifically, the use of volatile base objects as elements of the array *Announce* is sufficient to ensure a "clean shutdown" of the helping mechanism during a failure.

5.2 Implementation of MRSW Registers from SRSW Registers

As our second example we analyze a wait-free implementation of atomic multi-reader single-writer (MRSW) registers from atomic single-reader single-writer (SRSW) registers. The construction, which we refer to as MRSW, is presented in Section 4.2.5 of [15] and is similar to Israeli and Li's [19]. We chose this example because it illustrates the case when a known implementation fails to satisfy R-linearizability "out of the box," even if we assume that

Base objects:

```
A:      shared array[0..N-1, 0..N-1] of record
        [V: value, T: timestamp] initialized to
        (V_0, 0) where V_0 is the implemented
        type's initial value
T_max:  integer, initially 0
```

Function write(V: value) for process p_w

```
$T := T_{\max} + 1$
$T_{\max} := T$
foreach $i$: int in 0..$N$-1 do
    $A[i, i] := (V, T)$
end
```

Function read() for process p_i

```
$(V_i, T_i) := (\text{NULL}, -1)$
foreach $j$: int in 0..$N$-1 do
    $(V_{temp}, T_{temp}) := A[j, i]$
    if $T_{temp} > T_i$ then
        $(V_i, T_i) := (V_{temp}, T_{temp})$
    end
end
foreach $j$: int in 0..$N$-1 do
    if $j \neq i$ then
        $A[i, j] := (V_{temp}, T_{temp})$
    end
end
return $V_{temp}$
```

Figure 4 Implementation of atomic MRSW registers from atomic SRSW registers [15].

base objects are non-volatile. Furthermore, as we explain later on, a modified R-linearizable version of this implementation separates R-linearizability from strict linearizability. For completeness the pseudo-code for the implementation is included in Figure 4.

The implemented object is represented using an array $A[0..N\text{-}1, 0..N\text{-}1]$ of SRSW register base objects. The distinguished writer process, denoted p_w for some $w \in 0..N\text{-}1$, maintains a variable $T_{\max}$, initially 0, for timestamping write operations. Each process p_i is able to read column i, and write row i with the exception of the diagonal element $A[i, i]$, which is only written by p_w. Each element of A is of the form (V, T), where V is a value written to the implemented MRSW register and T is a timestamp assigned by the writer. In the initial state, V holds the implemented register's initial value and $T = 0$ for each array element. To apply a write(V) operation, process p_w increments $T_{\max}$ to obtain a timestamp T higher than any prior timestamp, and writes (V, T) to the diagonal elements of A. To apply a read operation, process p_i reads the highest timestamp T_i in column i to determine the corresponding latest value V, then writes (V, T_i) to each element in row i except $A[i, i]$, and finally returns V. By writing (V, T_i) in row i, p_i announces its response to other readers, which ensures that subsequent reads do not return older values. (Values are ordered naturally in terms of "age" because there is only one writer.) This register implementation has time complexity $O(N)$ for N processes, and is linearizable in the absence of failures.

Before analyzing the implementation in the crash-recovery model, a subtle detail must be settled: we assume that by default p_w writes the diagonal elements in the order specified by the pseudo-code, namely from $A[0, 0]$ to $A[N\text{-}1, N\text{-}1]$. This point is irrelevant in a conventional asynchronous model with permanent crash failures, but as we explain shortly, it is crucial for correctness in our crash-recovery model. Assuming that both A and $T_{\max}$ are non-volatile base objects, which is necessary for $T_{\max}$ to increase monotonically whenever the implemented register object is written, the MRSW register construction violates R-linearizability in our crash recovery model. Thus, the construction does not work correctly "out of the box," in contrast to Herlihy's construction. This result is stated in Theorem 12, whose detailed proof is omitted due to lack of space.

▶ **Theorem 12.** *For any number of processes $N \geq 2$, implementation MRSW has a history that is not R-linearizable.*

```
Function write($V$: value) for process $p_w$
$T := T_{\max} + 1$
$T_{\max} := T$
$A[w, w] := (V, T)$
foreach $i$: int in 0..$N$-1 do
    if $i \neq w$ then
        $A[i, i] := (V, T)$
    end
end
```

Figure 5 Access procedure for the write operation of implementation MRSW′.

Proof sketch. Consider $N = 2$, with p_1 being the designated writer. The implementation has a history where p_1 begins a write(1) operation and a crash failure occurs after p_1 has written only one of the diagonal elements of A, namely $A[0, 0]$. Next, p_1 executes a read() and obtains the initial value, say 0, from $A[0, 1]$ and $A[1, 1]$. Finally, p_0 executes a read() and obtains the new value, namely 1, from $A[0, 0]$. Thus, p_1's interrupted write appears to take effect not only after the failure but also after its subsequent read() operation.[1] ◀

▶ **Corollary 13.** *For any number of processes $N \geq 2$, implementation MRSW has a history that is not persistently linearizable or strictly linearizable.*

The proof of Theorem 12 not only illustrates a weakness of implementation MRSW with respect to R-linearizability, but it also suggests a remedy. That is, while executing a write operation, the distinguished writer process p_w should overwrite the diagonal elements of array A starting with the row and column corresponding to its own process ID. We refer to the modified implementation as MRSW′, and present the pseudo-code for the modified write access procedure in Figure 5. The correctness of MRSW′ is established in Theorems 14–16. The detailed proofs are omitted due to lack of space.

▶ **Theorem 14.** *Every finite history H of implementation MRSW′ is R-linearizable.*

Proof sketch. Writing the diagonal elements of A in the modified order addresses the specific problem described in the proof sketch of Theorem 12 because p_1's read() operation correctly observes the value assigned by p_1's earlier write(1). Thus, the interrupted write appears to take effect before the writer's subsequent operation on the same object. The case of an interrupted write followed by another write is also dealt with correctly, as the latter operation overwrites all the base objects accessed by the former. ◀

▶ **Theorem 15.** *For any number of processes $N \geq 2$, implementation MRSW′ has a finite history that is neither persistently nor strictly linearizable.*

Proof sketch. Consider $N = 2$, with p_1 being the designated writer. The implementation has a history where p_1 begins a write(1) operation and a crash failure occurs after p_1 has written only one of the diagonal elements of A, namely $A[1, 1]$. Next, p_1 invokes a read() operation

[1] In a conventional crash failure model, there is no need to implement a read() operation for the designated writer because this process can record the last value written to the implemented register using a private variable. In contrast, in our model the value of such a private variable would be lost during a failure.

but does not yet access any base objects. Process p_0 then races ahead and completes a read() that obtains the initial value, say 0, from $A[0, 0]$ and $A[1, 0]$. Finally, p_1 completes its read() and obtains the new value, namely 1, from $A[1, 1]$. In this history, p_1's interrupted write(1) operation appears to take effect after p_0's read(), hence after the invocation step of p_1's read(), which violates both persistent and strict linearizability. ◀

▶ **Theorem 16.** *Implementations MRSW and MRSW′ for N processes have time complexity $O(N)$.*

Theorems 14–15 separate R-linearizability from strict linearizability in the following sense: whereas a strictly linearizable wait-free implementation of MRSW registers from atomic SRSW registers is impossible in an asynchronous model with crash failures, an R-linearizable wait-free implementation is possible in our asynchronous model with crash-recovery failures.

6 Conclusion

In this paper we defined a shared memory model with crash-recovery failures and a combination of volatile and non-volatile main memory. We then surveyed a number of safety properties inspired by linearizability that address the behavior of operations interrupted by failures in our model, identified the limitations of these properties, and proposed an alternative property called R-linearizability. Finally, we discussed implementation techniques.

Our coverage of implementation techniques centers around an approach where a known linearizable implementation designed for a conventional shared memory model is instantiated by making all base objects non-volatile, and then transformed as needed to yield R-linearizability. We showed that Herlihy's construction is R-linearizable "out of the box", and can be made strictly linearizable using an additional transformation that prevents the helping mechanism from acting on operations invoked prior to the most recent failure. In contrast, we showed that the MRSW register construction is not immediately R-linearizable, but can be made R-linearizable using a transformation that changes the order in which base objects are written. As shown by Aguilera and Frølund, a wait-free strictly linearizable MRSW register implementation from SRSW registers is impossible [2], and thus our transformed MRSW construction separates R-linearizability formally from strict linearizability.

Acknowledgments. Ryan Berryhill contributed to this research while enrolled in undergraduate studies at the University of Waterloo, ECE Department. Wojciech Golab and Mahesh Tripunitara are supported in part by the NSERC Discovery Grants Program. The authors are grateful to the anonymous referees and to Vassos Hadzilacos for their feedback on this work.

References

1 Yehuda Afek, David S. Greenberg, Michael Merritt, and Gadi Taubenfeld. Computing with faulty shared memory. In *Proc. 11th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 47–58, 1992.

2 Marcos K. Aguilera and Svend Frølund. Strict linearizability and the power of aborting. Technical Report HPL-2003-241, HP Labs, Palo Alto, CA, USA, November 2003.

3 Marcos K. Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, 2000.

4 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.

5 Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

6 Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. Help! In *Proc. of the 34th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 241–250, 2015.

7 Tushar D. Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996.

8 Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *Proc. of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 105–118, 2011.

9 Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, pages 133–146, 2009.

10 Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, August 1965.

11 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

12 Svend Frølund, Arif Merchant, Yasushi Saito, Susan Spence, and Alistair Veitch. Building storage registers from crash-recovery processes. Technical Report HPL-SSP-2003-14, HP Labs, Palo Alto, CA, USA, 2003.

13 Rachid Guerraoui and Ron R. Levy. Robust emulations of shared memory in a crash-recovery model. In *Proc. of the 24th International Conference on Distributed Computing Systems (ICDCS)*, pages 400–407, 2004.

14 Maurice Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, January 1991.

15 Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2012.

16 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, July 1990.

17 Jaap-Henk Hoepman, Marina Papatriantafilou, and Philippas Tsigas. Self-stabilization of wait-free shared memory objects. *J. Parallel Distrib. Comput.*, 62(5):818–842, 2002.

18 Engin Ipek, Jeremy Condit, Edmund B. Nightingale, Doug Burger, and Thomas Moscibroda. Dynamically replicated memory: building reliable systems from nanoscale resistive memories. In *Proc. of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 3–14, 2010.

19 Amos Israeli and Ming Li. Bounded time-stamps. In *Proc. of the 28th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 371–382, 1987.

20 Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45:451–500, 1998.

21 Colette Johnen and Lisa Higham. Fault-tolerant implementations of regular registers by safe registers with applications to networks. In *Proc. of 10th International Conference of Distributed Computing and Networking (ICDCN)*, pages 337–348, 2009.

22 Leslie Lamport. On interprocess communication, Part I: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

23 Leslie Lamport. On interprocess communication, Part II: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

24 Michael Merritt and Gadi Taubenfeld. Computing with infinitely many processes. In *Proc. of the 14th International Conference on Distributed Computing (DISC)*, pages 164–178, 2000.

25 Jeffrey C. Mogul, Eduardo Argollo, Mehul A. Shah, and Paolo Faraboschi. Operating system support for NVM+DRAM hybrid main memory. In *Proc. of the 12th Workshop on Hot Topics in Operating Systems (HotOS)*, 2009.

26 C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992.

27 Thomas Moscibroda and Rotem Oshman. Resilience of mutual exclusion algorithms to transient memory faults. In *Proc. of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 69–78, 2011.

28 Stan Park, Terence Kelly, and Kai Shen. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *Proc. of the 8th ACM European Conference on Computer Systems (EuroSys)*, pages 225–238, 2013.

29 David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufmann Publishers, second edition, 1997.

30 Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *Proc. of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, pages 265–276, 2014.

31 Yasushi Saito, Svend Frølund, Alistair C. Veitch, Arif Merchant, and Susan Spence. FAB: building distributed enterprise disk arrays from commodity components. In *Proc. of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 48–58, 2004.

32 Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, August 1983.

33 Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proc. of the 9th USENIX conference on File and Storage Technologies (FAST)*, 2011.

34 Roman Vitenberg and Roy Friedman. On the locality of consistency conditions. In *Proc. of the 17th International Symposium on Distributed Computing (DISC)*, pages 92–105, 2003.

35 Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: lightweight persistent memory. In *Proc. of the 16th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 91–104, 2011.