Fault Tolerance for Remote Memory Access Programming Models

Maciej Besta[*]
Dept. of Computer Science
ETH Zurich
Universitätstr. 6, 8092 Zurich, Switzerland
maciej.besta@inf.ethz.ch

Torsten Hoefler
Dept. of Computer Science
ETH Zurich
Universitätstr. 6, 8092 Zurich, Switzerland
htor@inf.ethz.ch

ABSTRACT

Remote Memory Access (RMA) is an emerging mechanism for programming high-performance computers and datacenters. However, little work exists on resilience schemes for RMA-based applications and systems. In this paper we analyze fault tolerance for RMA and show that it is fundamentally different from resilience mechanisms targeting the message passing (MP) model. We design a model for reasoning about fault tolerance for RMA, addressing both flat and hierarchical hardware. We use this model to construct several highly-scalable mechanisms that provide efficient low-overhead in-memory checkpointing, transparent logging of remote memory accesses, and a scheme for transparent recovery of failed processes. Our protocols take into account diminishing amounts of memory per core, one of the major features of future exascale machines. The implementation of our fault-tolerance scheme entails negligible additional overheads. Our reliability model shows that in-memory checkpointing and logging provide high resilience. This study enables highly-scalable resilience mechanisms for RMA and fills a research gap between fault tolerance and emerging RMA programming models.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of systems—*Fault tolerance*

General Terms

Reliability, Performance, Algorithms

1. INTRODUCTION

Partitioned Global Address Space (PGAS), and the wider class of Remote Memory Access (RMA) programming models enable high-performance communications that often outperform Message Passing [19, 34]. RMA utilizes remote direct memory access (RDMA) hardware features to access memories at remote processes without involving the OS or the remote CPU.

RDMA is offered by most modern HPC networks (InfiniBand, Myrinet, Cray's Gemini and Aries, IBM's Blue Gene, and PERCS) and many Ethernet interconnects that use the RoCE or iWARP protocols. RMA languages and libraries include Unified Parallel C (UPC), Fortran 2008 (formerly known as CAF), MPI-3 One Sided, Cray's SHMEM interface, or Open Fabrics (OFED). Thus, we observe that RMA is quickly emerging to be the programming model of choice for cluster systems, HPC computers, and large datacenters.

Fault tolerance of such systems is important because hardware and software faults are ubiquitous [38]. Two popular resilience schemes used in today's computing environments are coordinated checkpointing (CC) and uncoordinated checkpointing augmented with message logging (UC) [17]. In CC applications regularly synchronize to save their state to memory, local disks, or parallel file system (PFS) [38]; this data is used to restart after a crash. In UC processes take checkpoints independently and use message logging to avoid rollbacks caused by the *domino effect* [37]. There has been considerable research on CC and UC for the message passing (MP) model [6, 17]. Still, no work addresses the exact design of these schemes for RMA-based systems.

In this work we develop a generic model for reasoning about resilience in RMA. Then, using this model, we show that CC and UC for RMA fundamentally differ from analogous schemes for MP. We also construct protocols that enable simple checkpointing and logging of remote memory accesses. We *only* use *in-memory* mechanisms to avoid costly I/O flushes and frequent disk and PFS failures [24, 38]. We then extend our model to cover two features of today's petascale and future exascale machines: (1) the growing complexity of hardware components and (2) decreasing amounts of memory per core. *With this, our study fills an important knowledge gap between fault-tolerance and emerging RMA programming in large-scale computing systems.*

In detail, we provide the following major contributions:

- We design a model for reasoning about the reliability of RMA systems running on flat and hierarchical hardware with limited memory per core. To our knowledge, this is the first work that addresses these issues.
- We construct schemes for in-memory checkpointing, logging, and recovering RMA-based applications.
- We unify these concepts in a topology-aware diskless protocol and we use real data and an analytic model to show that the protocol can endure concurrent hardware failures.

[*]MB is supported by the 2013 Google European Doctoral Fellowship in Parallel Computing.

HPDC'14, June 23–27, Vancouver, BC, Canada.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2749-7/14/06 ...$15.00.
http://dx.doi.org/10.1145/2600212.2600224.

	MPI-3 one sided operation	UPC operation	Fortran 2008 operation	Cat.
comm.	MPI_Put, MPI_Accumulate, MPI_Get_accumulate, MPI_Fetch_and_op, MPI_Compare_and_swap	upc_memput, upc_memcpy, upc_memset, assignment (=), all UPC collectives	assignment (=)	PUT
	MPI_Get, MPI_Compare_and_swap, MPI_Get_accumulate, MPI_Fetch_and_op	upc_memget, upc_memcpy, upc_memset, assignment (=), all UPC collectives	assignment (=)	GET
sync.	MPI_Win_lock, MPI_Win_lock_all	upc_lock	lock	LOCK
	MPI_Win_unlock, MPI_Win_unlock_all	upc_unlock	unlock	UNLOCK
	MPI_Win_fence	upc_barrier	sync_all, sync_team, sync_images	GSYNC
	MPI_Win_flush, MPI_Win_flush_all, MPI_Win_sync	upc_fence	sync_memory	FLUSH

Table 1: Categorization of MPI One Sided/UPC/Fortran 2008 operations in our model. Some atomic functions are considered as both PUTs and GETs. In UPC, the collectives, assignments and upc_memset/upc_memcpy behave similarly depending on the values of pointers to shared objects; the same applies to Fortran 2008. We omit MPI's post-start-complete-wait synchronization and request-based RMA operations for simplicity.

- We present the implementation of our protocol, analyze its performance, show it entails negligible overheads, and compare it to other schemes.

2. RMA PROGRAMMING

We now discuss concepts of RMA programming and present a formalization that covers existing RMA/PGAS models with strict or relaxed memory consistency (e.g., UPC or MPI-3 One Sided). In RMA, each process explicitly exposes an area of its local memory as shared. Memory can be shared in different ways (e.g., MPI windows, UPC shared arrays, or Co-Arrays in Fortran 2008); details are outside the scope of this work. Once shared, memory can be accessed with various language-specific operations.

2.1 RMA Operations

We identify two fundamental types of RMA operations: *communication* actions (often called *accesses*; they transfer data between processes), and *synchronization* actions (synchronize processes and guarantee memory consistency). A process p that issues an RMA action targeted at q is called the *active source*, and q is called the *passive target*. We assume p is active and q is passive (unless stated otherwise).

2.1.1 Communication Actions

We denote an action that transfers data from p to q and from q to p as PUT($p \rightrightarrows q$) and GET($p \leftrightarrows q$), respectively. We use double-arrows to emphasize the asymmetry of the two operations: the upper arrow indicates the direction of data flow and the lower arrow indicates the direction of control flow. The upper part of Table 1 categorizes communication operations in various RMA languages. Some actions (e.g., atomic compare and swap) transfer data in *both* directions and thus fall into the family of PUTs *and* GETs.

We also distinguish between PUTs that "blindly" replace a targeted memory region at q with a new value (e.g., UPC assignment), and PUTs that combine the data moved to q with the data that already resides at q (e.g., MPI_Accumulate). When necessary, we refer to the former type as the *replacing* PUT, and to the latter as the *combining* PUT.

2.1.2 Memory Synchronization Actions

We identify four major categories of memory synchronization actions: LOCK($p \rightarrow q, str$) (locks a structure str in q's memory to provide exclusive access), UNLOCK($p \rightarrow q, str$) (unlocks str in q's memory and enforces consistency of str), FLUSH($p \rightarrow q, str$) (enforces consistency of str in p's and q's memories), and GSYNC($p \rightarrow \diamond, str$) (enforces consistency of str); $\diamond$ indicates that the call targets all processes. Arrows indicate the flow of control (synchronization). When we refer to the whole process memory (and not a single structure), we omit str (e.g., LOCK($p \rightarrow q$)). The lower part of Table 1 categorizes synchronization calls in various RMA languages.

2.2 Epochs and Consistency Order

RMA's relaxed memory consistency enables non-blocking PUTs and GETs. Issued operations are completed by memory consistency actions (FLUSH, UNLOCK, GSYNC). The period between any two such actions issued by p and targeting the same process q is called an *epoch*. Every UNLOCK($p \rightarrow q$) or FLUSH($p \rightarrow q$) *closes* p's current epoch and *opens* a new one (i.e., increments p's epoch number denoted as $E(p \rightarrow q)$). p can be in several independent epochs related to each process that it communicates with. As GSYNC is a collective call, it increases epochs at every process.

An important concept related to epochs is the *consistency order* (denoted as $\xrightarrow{co}$). $\xrightarrow{co}$ orders the visibility of actions: $x \xrightarrow{co} y$ means that memory effects of action x are globally visible before action y. Actions issued in different epochs by process p targeting the same process q are always ordered with $\xrightarrow{co}$. Epochs and $\xrightarrow{co}$ are illustrated in Figure 1. $x \parallel_{co} y$ means that actions x and y are *not* ordered with $\xrightarrow{co}$.

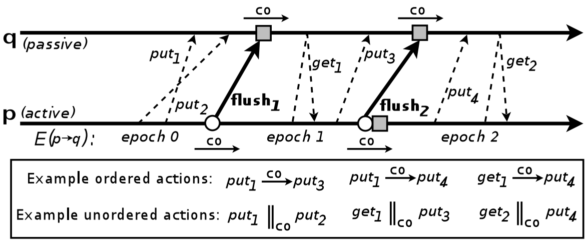

co
co
q (passive)
put1
get1
put3
get2
put2
flush1
flush2
put4
p (active)
E(p→q): epoch 0
co
epoch 1
co
epoch 2
Example ordered actions: put1 →co put3 put1 →co put4 get1 →co put4
Example unordered actions: put1 ||co put2 get1 ||co put3 get2 ||co put4

Figure 1: Epochs and the consistency order $\xrightarrow{co}$ (§ 2.2). White circles symbolize synchronization calls (in this case FLUSH). Grey squares show when calls' results become globally visible in q's or p's memory.

2.3 Program, Synchronization, and Happened Before Orders

In addition to $\xrightarrow{co}$ we require three more orders to specify an RMA execution [22]: The *program order* ($\xrightarrow{po}$) specifies the order of actions of a single thread, similarly to the program order in Java [29] ($x \xrightarrow{po} y$ means that x is called before y by some thread). The *synchronization order* ($\xrightarrow{so}$) orders LOCK and UNLOCK and other synchronizing operations. *Happened-before* (HB, $\xrightarrow{hb}$), a relation well-known in message passing [27], is the transitive closure of the union of $\xrightarrow{po}$ and $\xrightarrow{so}$. We abbreviate a *consistent happen-before* as $\xrightarrow{cohb}$: $a \xrightarrow{cohb} b \equiv a \xrightarrow{co} b \wedge a \xrightarrow{hb} b$. To state that actions are *parallel* in an order, we use the symbols $\parallel_{po}$, $\parallel_{so}$, $\parallel_{hb}$. We show the orders in Fig. 2; more details can be found in [22].

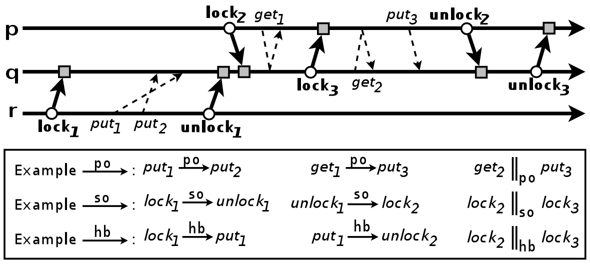

p
q
r
lock2 get1
put3 unlock2
lock3 get2
unlock3
lock1 put1 put2 unlock1
Example $\xrightarrow{po}$: $put_1 \xrightarrow{po} put_2$ $get_1 \xrightarrow{po} put_3$ $get_2 \parallel_{po} put_3$
Example $\xrightarrow{so}$: $lock_1 \xrightarrow{so} unlock_1$ $unlock_1 \xrightarrow{so} lock_2$ $lock_2 \parallel_{so} lock_3$
Example $\xrightarrow{hb}$: $lock_1 \xrightarrow{hb} put_1$ $put_1 \xrightarrow{hb} unlock_2$ $lock_2 \parallel_{hb} lock_3$

Figure 2: Example RMA orderings $\xrightarrow{po}$, $\xrightarrow{so}$, $\xrightarrow{hb}$ (§ 2.3).

2.4 Formal Model

We now combine the various RMA concepts and fault tolerance into a single formal model. We assume fail-stop faults (processes can disappear nondeterministically but behave correctly while being a part of the program). The data communication may happen out of order as specified for most RMA models. Communication channels between non-failed processes are asynchronous, reliable, and error-free. The user code can only communicate and synchronize using RMA functions specified in Section 2.1. Finally, checkpoints and logs are stored in *volatile* memories.

We define a communication action a as a tuple

$$a = \langle type, src, trg, combine, EC, GC, SC, GNC, data \rangle \qquad (1)$$

where *type* is either a put or a get, *src* and *trg* specify the source and the target, and *data* is the data carried by a. *Combine* determines if a is a replacing PUT (*combine* = $false$) or a combining PUT (*combine* = *true*). *EC* (*Epoch Counter*) is the epoch number in which a was issued. GC, SC, and GNC are counters required for correct recovery; we discuss them in more detail in Section 4. We combine the notation from Section 2.1 with this definition and write $\text{PUT}(p \rightrightarrows q).EC$ to refer to the epoch in which the put happens. We also define a *determinant* of a (denoted as $\#a$, cf. [6]) to be tuple a without *data*:

$$\#a = \langle type, src, trg, combine, EC, GC, SC, GNC \rangle. \qquad (2)$$

Similarly, a synchronization action b is defined as

$$b = \langle type, src, trg, EC, GC, SC, GNC, str \rangle. \qquad (3)$$

Finally, a trace of an RMA program running on a distributed system can be written as the tuple

$$\mathcal{D} = \langle \mathcal{P}, \mathcal{E}, \mathcal{S}, \xrightarrow{po}, \xrightarrow{so}, \xrightarrow{hb}, \xrightarrow{co} \rangle, \qquad (4)$$

where

$\mathcal{P}$ is the set of all $\mathcal{P}$rocesses in $\mathcal{D}$ ($|\mathcal{P}| = N$),

$\mathcal{E} = \mathcal{A} \cup \mathcal{I}$ is the set of all $\mathcal{E}$vents:

$\mathcal{A}$ is the set of RMA $\mathcal{A}$ctions,

$\mathcal{I}$ is the set of $\mathcal{I}$nternal actions (reads, writes, checkpoint actions). $\text{READ}(x, p)$ loads local variable x and $\text{WRITE}(x := val, p)$ assigns val to x (in p's memory). C_p^i is the ith checkpoint action taken by p. Internal events are partially ordered with actions using $\xrightarrow{po}$, $\xrightarrow{co}$, and $\xrightarrow{hb}$.

$\mathcal{S}$ is the set of all data $\mathcal{S}$tructures used by the program.

3. FAULT-TOLERANCE FOR RMA

We now present schemes that make RMA codes fault tolerant. We start with the simpler CC and then present the protocols for UC.

3.1 Coordinated Checkpointing (CC)

In many CC schemes, the user explicitly calls a function to take a checkpoint. Such protocols may leverage RMA's features (e.g., direct memory access) to improve the performance. However, these schemes have several drawbacks: they complicate the code because they can only be called when the network is quiet [21] and they do not always fit the optimality criteria such as Daly's checkpointing interval [15]. In this section, we first identify how CC in RMA differs from CC in MP and then describe a scheme for RMA codes that performs CC *transparently* to the application. We model a coordinated checkpoint as a set $C = \{C_{p_1}^{i_1}, C_{p_2}^{i_2}, ..., C_{p_N}^{i_N}\} \subseteq \mathcal{I}, p_m \neq p_n$ for any m, n.

3.1.1 RMA vs. MP: Coordinated Checkpointing

In MP, every C has to satisfy a *consistency condition* [21]: $\forall C_p^i, C_q^j \in C : C_p^i \parallel_{hb} C_q^j$. This condition ensures that C does not reflect a system state in which one process received a message that was *not* sent by any other process. We adopt this condition and extend it to cover all RMA semantics:

DEFINITION 1. *C is RMA-consistent iff $\forall C_p^i, C_q^j \in C$: $C_p^i \parallel_{cohb} C_q^j$.*

We extend $\parallel_{hb}$ to $\parallel_{cohb}$ to guarantee that the system state saved in C does not contain a process affected by a memory access that was *not* issued by any other process. In RMA, unlike in MP, this condition can be easily satisfied because each process can drain the network with a local FLUSH (enforcing consistency at any point is legal [22])

3.1.2 Taking a Coordinated Checkpoint

We now propose two diskless schemes that obey the RMA-consistency condition and target MPI-3 RMA codes. The first ("Gsync") scheme can be used in programs that *only* synchronize with GSYNCs. The other ("Locks") scheme targets codes that *only* synchronize with LOCKs and UNLOCKs. Note that in correct MPI-3 RMA programs GSYNCs and LOCKs/UNLOCKs cannot be mixed [31]. All our schemes assume that a GSYNC may also introduce an additional $\xrightarrow{hb}$ order, which is true in some implementations [31].

The "Gsync" Scheme Every process may take a coordinated checkpoint right after the user calls a GSYNC and before any further RMA calls by: (1) optionally enforcing the global $\xrightarrow{hb}$ order with an operation such as MPI_Barrier (denoted as BAR), and taking the checkpoint. Depending on the application needs, not every GSYNC has to be followed by a checkpoint. We use Daly's formula [15] to compute the best interval between such checkpoints and we take checkpoints after the right GSYNC calls.

THEOREM 3.1. *The Gsync scheme satisfies the RMA-consistency condition and does not deadlock.*

PROOF. We assume correct MPI-3 RMA programs represented by their trace $\mathcal{D}$ [22, 31]. For all $p, q \in \mathcal{P}$, each $\text{GSYNC}(p \to \diamond)$ has a matching $\text{GSYNC}(q \to \diamond)$ such that $[\text{GSYNC}(p \to \diamond) \parallel_{hb} \text{GSYNC}(q \to \diamond)]$. Thus, if every process calls BAR right after GSYNC then BAR matching is guaranteed and the program cannot deadlock. In addition, the GSYNC calls introduce a global consistency order $\xrightarrow{co}$ such that the checkpoint is coordinated and consistent. □

The "Locks" Scheme Every process p maintains a local *Lock Counter* LC_p that starts with zero and is incremented after each LOCK and decremented after each UNLOCK. When $LC_p = 0$, process p can perform a checkpoint in three phases: (1) enforce consistency with a $\textsc{flush}(p \rightarrow \diamond)$, (2) call a BAR to provides the global $\xrightarrow{hb}$ order, and (3) take a checkpoint C_p^i. The last phase, the actual checkpoint stage, is performed collectively thus all processes can take the checkpoint C in coordination.

THEOREM 3.2. *The Locks scheme satisfies the RMA-consistency condition and does not deadlock.*

PROOF. The call to $\textsc{flush}(p \rightarrow \diamond)$ in phase 1 guarantees global consistency at each process. The BAR in phase 2 guarantees that all processes are globally consistent before the checkpoint taken in phase 3.

It remains to proof deadlock-freedom. We assume correct MPI-3 RMA programs [22, 31]. A $\textsc{lock}(p \rightarrow q)$ can only block waiting for an active lock $\textsc{lock}(z \rightarrow q)$ and no BAR can be started at z while the lock is held. In addition, for every $\textsc{lock}(z \rightarrow q)$, there is a matching $\textsc{unlock}(z \rightarrow q)$ in the execution such that $\textsc{lock}(z \rightarrow q) \xrightarrow{po} \textsc{unlock}(z \rightarrow q)$ (for any $z, p, q \in \mathcal{P}$). Thus, all locks must be released eventually, i.e., $\exists a \in \mathcal{E} : \; a \xrightarrow{po} \textsc{write}(LC_p := 0, p)$ for any $p \in \mathcal{P}$. □

The above schemes show that the transparent CC can be achieved much simpler in RMA than in MP. In MP, such protocols usually have to analyze inter-process dependencies due to sent/received messages, and add protocol-specific data to messages [11, 17], which reduces the bandwidth. In RMA this is not necessary.

3.2 Uncoordinated Checkpointing (UC)

Uncoordinated checkpointing augmented with message logging reduces energy consumption and synchronization costs because a single process crash does not force all other processes to revert to the previous checkpoint and recompute [17, 37]. Instead, a failed process fetches its last checkpoint and replays messages logged beyond this checkpoint. However, UC schemes are usually more complex than CC [17]. We now analyze how UC in RMA differs from UC in MP, followed by a discussion of our UC protocols. Data structures for the protocols are shown in Table 2.

3.2.1 RMA vs. MP: Uncoordinated Checkpointing

The first and obvious difference is that we now log not *messages* but *accesses*. Other differences are as follows:

Storing Access Logs In MP, processes exchange messages that *always* flow *from* the sender (process p) *to* the receiver (process q). Messages can be recorded at the sender's side [17,37]. During a recovery, the restored process interacts with other processes to get and reply the logged messages (see Figure 3 (part (1)).

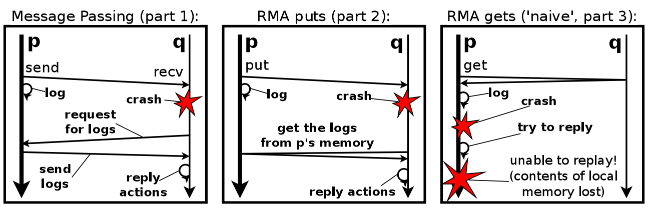

Message Passing (part 1):
p
q
send
recv
log
crash
request for logs
send logs
reply actions
RMA puts (part 2):
p
q
put
log
crash
get the logs from p's memory
reply actions
RMA gets ('naive', part 3):
p
q
get
log
crash
try to reply
unable to replay! (contents of local memory lost)

Figure 3: The logging of messages vs. RMA puts and gets (§ 3.1.1).

In RMA, a $\textsc{put}(p \rightrightarrows q)$ changes the state of q, but a $\textsc{get}(p \leftrightarrows q)$ modifies the state of p. Thus, $\textsc{put}(p \rightrightarrows q)$ can be logged in p's memory, but $\textsc{get}(p \leftrightarrows q)$ cannot because a failure of p would prevent a successful recovery (see Figure 3, part 2 and 3).

Transparency of Schemes In MP, both p and q actively participate in communication. In RMA, q is oblivious to accesses to its memory and thus any recovery or logging performed by p can be *transparent* to (i.e., does not obstruct) q (which is usually *not* the case in MP, cf. [37]).

No Piggybacking Adding some protocol-specific data to messages (e.g., *piggybacking*) is a popular concept in MP [17]. Still, it cannot be used in RMA because PUTs and GETs are accesses, not messages. Yet, issuing additional accesses is cheap in RMA.

Access Determinism Recent works in MP (e.g., [20]) explore *send determinism*: the output of an application run is oblivious to the order of received messages. In our work we identify a similar concept in RMA that we call *access determinism*. For example, in race-free MPI-3 programs the application output does not depend on the order in which two accesses a and b committed to memory if $a \parallel_{co} b$.

Orphan Processes In some MP schemes (called *optimistic*), senders postpone logging messages for performance reasons [17]. Assume q received a message m from p and then sent a message m' to r. If q crashes and m is not logged by p at that time, then q may follow a run in that it *does not* send m'. Thus, r becomes an *orphan*: its state depends on a message m' that was *not* sent [17] (see Figure 4, part 1).

In RMA, a process may also become an orphan. Consider Figure 4 (part 2). First, p modifies a variable x at q. Then, q reads x and conditionally issues a $\textsc{put}(q \rightrightarrows r)$. If q crashes and p postponed logging $\textsc{put}(p \rightrightarrows q)$, then q (while recovering) may follow a run in which it does not issue $\textsc{put}(q \rightrightarrows r)$; thus r becomes an orphan.

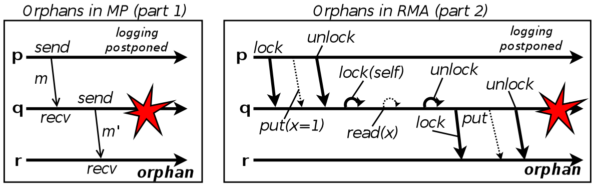


Figure 4: Illustration of orphans in MP and RMA (§ 3.1.1).

3.2.2 Taking an Uncoordinated Checkpoint

We denote the ith uncoordinated checkpoint taken by process p as C_p^i. Taking C_p^i is simple and entails: (1) locking local application data, (2) sending the copy of the data to some remote volatile storage, and (3) unlocking the application data (we defer the discussion on the implementation details until Section 6). After p takes C_p^i, any process q can delete the logs of every $\textsc{put}(q \rightrightarrows p)$ (from $LP_q[p]$) and $\textsc{get}(p \leftrightarrows q)$ (from $LG_q[p]$) that committed in p's memory before C_p^i (i.e., $\textsc{put}(q \rightrightarrows p) \xrightarrow{co} C_p^i$ and $\textsc{get}(p \leftrightarrows q) \xrightarrow{co} C_p^i$).

We demand that every C_p^i is taken *immediately after* closing/opening an epoch and *before* issuing any new communication operations (we call this the *epoch condition*). This condition is required because, if p issues a $\textsc{get}(p \leftrightarrows q)$, the application data is guaranteed to be consistent only after closing the epoch.

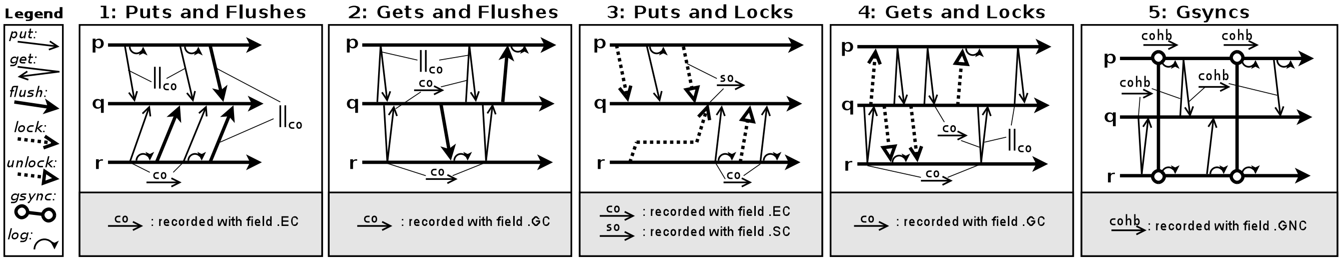

Legend
put:
get:
flush:
lock:
unlock:
gsync:
log:
1: Puts and Flushes
2: Gets and Flushes
3: Puts and Locks
4: Gets and Locks
5: Gsyncs
p
q
r
$\|_{co}$
co
so
cohb
co : recorded with field .EC
co : recorded with field .GC
co : recorded with field .EC
so : recorded with field .SC
co : recorded with field .GC
cohb : recorded with field .GNC

Figure 5: Logging orders $\xrightarrow{so}$, $\xrightarrow{co}$, and $\xrightarrow{hb}$ (§ 4.1). In each figure we illustrate example orderings.

Structure	Description
$LP_p[q] \in \mathcal{S}$	Logs of PUTs issued by p and targeted at q.
$LG_q[p] \in \mathcal{S}$	Logs of GETs targeted at q and issued by p.
$LP_p \in \mathcal{S}$	Logs of PUTs issued and stored by p and targeted at any other process; $LP_p \equiv \bigcup_{r \in \mathcal{P} \wedge r \neq p} LP_p[r]$.
$LG_q \in \mathcal{S}$	Logs of *gets* targeted and stored at q, issued by any other process; $LG_q \equiv \bigcup_{r \in \mathcal{P} \wedge r \neq q} LG_q[r]$.
$Q_p \in \mathcal{S}$	A helper container stored at p, used to temporarily log #GETs issued by p.
$N_q[p] \in \mathcal{S}$	A structure (stored at q) that determines whether or not p issued a non-blocking GET($p \rightrightarrows q$) ($N_q[p] = true$ or $false$, respectively)

Table 2: Data structures used in RMA logging (§ 3.2.3). $LP_p[q]$ and LP_p are stored at p. $LG_q[p]$ and LG_q are stored at q.

3.2.3 Transparent Logging of RMA Accesses

We now describe the logging of PUTs and GETs.

Logging Puts To log a PUT($p \rightrightarrows q$), p first calls LOCK($p \rightarrow p, LP_p$). Self-locking is necessary because there may be other processes being recovered that may try to read LP_p. Then, the PUT is logged ($LP_p[q] := LP_p[q] \cup \{PUT(p \rightrightarrows q)\}$; ":=" denotes the assignment of a new value to a variable or a structure). Finally, p unlocks LP_p. Atomicity between logging and putting is not required because, in the weak consistency memory model, the source memory of the put operation may not be modified until the current epoch ends. If the program modifies it nevertheless, RMA implementations are allowed to return any value, thus the logged value is irrelevant. We log PUT($p \rightrightarrows q$) before closing the epoch PUT($p \rightrightarrows q$)$.EC$. If the PUT is blocking then we log it before issuing, analogously to the *pessimistic* message logging [17].

Logging Gets We log a GET($p \leftrightarrows q$) in two phases to retain its asynchronous behavior (see Algorithm 1). First, we record the determinant #GET($p \leftrightarrows q$) in Q_p (lines 2-3). We cannot access GET($p \leftrightarrows q$)$.data$ as the local memory will only be valid after the epoch ends. We avoid issuing an additional blocking FLUSH($p \rightarrow q$), instead we rely on the user's call to end the epoch. Second, when the user ends the epoch, we lock the remote log LG_q, record GET($p \leftrightarrows q$), and unlock LG_q (lines 4-7).

Note that if p fails between issuing GET($p \leftrightarrows q$) and closing the epoch, it will not be able to replay it consistently. To address this problem, p sets $N_q[p]$ at process q to *true* right before issuing the first GET($p \rightrightarrows q$) (line 1), and to *false* after closing the epoch GET($p \rightrightarrows q$)$.EC$ (line 8). During the recovery, if p notices that any $N_q[p] = true$, it falls back to another resilience mechanism (i.e., the last coordinated checkpoint). If the GET is blocking then we set $N_q[p] = false$ after returning from the call.

Algorithm 1: Logging *gets* (§ 3.2.3)

```
  Input: $get :=$ GET$(p \leftrightarrows q)$
  /* Phase 1:  starts right before issuing the $get$     */
1 $N_q[p] := true$
  /* Now we issue the $get$ and log the $\#get$          */
2 issue GET$(p \leftrightarrows q)$
3 $Q_p \leftarrow Q_p \cup \#get$
  /* Phase 2:  begins after ending the epoch $get.EC$    */
4 LOCK$(p \rightarrow q, LG_q)$
5 $LG_q[p] := LG_q[p] \cup get$
6 $Q_p := Q_p \setminus \#get$
7 UNLOCK$(p \rightarrow q, LG_q)$
8 $N_q[p] := false$
```

4. CAUSAL RECOVERY FOR UC

We now show how to causally recover a failed process (*causally* means preserving $\xrightarrow{co}$, $\xrightarrow{so}$, and $\xrightarrow{hb}$). This section describes technical details on how to guarantee all orders to ensure a correct access replay. If the reader is not interested in all details, she may proceed to Section 5 without disrupting the flow. A causal process recovery has three phases: (1) fetching uncoordinated checkpoint data, (2) replaying accesses from remote logs, and (3) in case of a problem during the replay, falling back to the last coordinated checkpoint. We first show how we log the respective orderings between accesses (Section 4.1) and how we prevent replaying some accesses twice (Section 4.2). We finish with our recovery scheme (Section 4.3) and a discussion (Section 4.4). Due to space constraints, we include full proofs in the techreport version of the paper[1].

4.1 Logging Order Information

We now show how to record $\xrightarrow{so}$, $\xrightarrow{hb}$, and $\xrightarrow{co}$. For clarity, but without loss of generality, we separately present several scenarios that exhaust possible communication/synchronization patterns in our model. We consider three processes (p, q, r) and we analyze what data is required to replay q. We show each pattern in Figure 5.

A. Puts and Flushes First, p and r issue PUTs and FLUSHes at q. At both p and r, PUTs separated by FLUSHes are ordered with $\xrightarrow{co}$. This order is preserved by recording epoch counters ($.EC$) with every logged PUT($p \rightrightarrows q$). Note that, however, RMA semantics *do not* order calls issued by p and r: [PUT($p \rightrightarrows q$) $\|_{co}$ PUT($r \rightrightarrows q$)] without additional process synchronization. Here, we assume *access determinism*: the recovery output does not depend on the order in which such PUTs committed in q's memory.

B. Gets and Flushes Next, q issues GETs and FLUSHes targeted at p and r. Again, $\xrightarrow{co}$ has to be logged. However, this time GETs targeted at *different* processes *are* ordered (because they are issued by the same process). To log this

[1] http://spcl.inf.ethz.ch/Research/Parallel_Programming/ftRMA

ordering, q maintains a local *Get Counter* GC_q that is incremented each time q issues a FLUSH($q \rightarrow \diamond$) to any other process. The value of this counter is logged with each GET using the field $.GC$ (cf. Section 2.4).

C. Puts and Locks In this scenario p and r issue PUTs at q and synchronize their accesses with LOCKs and UNLOCKs. This pattern requires logging the $\xrightarrow{so}$ order. We achieve this with a *Synchronization Counter* SC_q stored at q. After issuing a LOCK($p \rightarrow q$), p (the same refers to r) fetches the value of SC_q, increments it, updates remote SC_q, and records it with every PUT using the field $.SC$ (cf. Section 2.4). In addition, this scenario requires recording $\xrightarrow{co}$ that we solve with $.EC$, analogously as in the "Puts and Flushes" pattern.

D. Gets and Locks Next, q issues GETs and uses LOCKs targeted at p and r. This pattern is solved analogously to the "Gets and Flushes" pattern.

E. Gsyncs The final pattern are GSYNCs (that may again introduce $\xrightarrow{hb}$) combined with any communication action. Upon a GSYNC, each process q increments its *GsyNc Counter* GNC_q that is logged in an actions' $.GNC$ field (cf. Section 2.4).

Algorithm 2: The causal recovery scheme (§ 4.3, § 4.4).

```
1  Function recovery()
2      fetch_checkpoint_data()
3      put_logs := {}; get_logs := {}
4      forall the $q \in \mathcal{P}$ : $q \neq p_{new}$ do
5          LOCK($p_{new} \rightarrow q$)
6          if $N_q[p_f] = 1 \vee M_q[p_f] = true$ then
               /* Stop the recovery and fall back to the
                  last coordinated checkpoint                */
7          end
8          put_logs := put_logs $\cup LP_q[p_f]$
9          get_logs := get_logs $\cup LG_q[p_f]$
10         UNLOCK($p_{new} \rightarrow q$)
11     end
12     while $|put\_logs| > 0 \vee |get\_logs| > 0$ do
13         gnc_logs := logsWithMinCnt(GNC, put_logs $\cup$ get_logs)
14         while $|gnc\_logs| > 0$ do
15             gnc_put_logs := gnc_logs $\cap$ put_logs
16             gnc_get_logs := gnc_logs $\cap$ get_logs
17             ec_logs := logsWithMinCnt(EC, gnc_put_logs)
18             gc_logs := logsWithMinCnt(GC, gnc_get_logs)
19             replayEachAction(ec_logs)
20             replayEachAction(gc_logs)
21             gnc_logs := gnc_logs \ (ec_logs $\cup$ gc_logs)
22         end
23         put_logs := put_logs \ gnc_logs
24         get_logs := get_logs \ gnc_logs
25     end
26     return
27 Function logsWithMinCnt(Counter, Logs)
       /* Return a set with logs from Logs that have the
          smallest value of the specified counter (one
          of:$GNC, EC, GC, SC$).                             */
28 Function replayEachAction(Logs)
       /* Reply each log from set Logs in any order.      */
29 Function fetchCheckpointData()
       /* Fetch the last checkpoint and load into the
          memory.                                          */
```

4.2 Preventing Replaying Accesses Twice

Assume that process p issues a PUT($p \rightrightarrows q$) (immediately logged by p in $LP_p[q]$) such that PUT($p \rightrightarrows q$) $\xrightarrow{co} C_q^j$. It means that the state of q recorded in checkpoint C_q^j is affected by PUT($p \rightrightarrows q$). Now assume that q fails and begins to replay the logs. If p did not delete the log of PUT($p \rightrightarrows q$) from $LP_p[q]$ (it was allowed to do it after q took C_q^j), then q replays PUT($p \rightrightarrows q$) and this PUT affects its memory *for the second time*. This is not a problem if PUT($p \rightrightarrows q$)$.combine = false$, because such a PUT always overwrites the memory region with the same value. However, if PUT($p \rightrightarrows q$)$.combine = true$, then q ends up in an inconsistent state (e.g., if this PUT increments a memory cell, this cell will be incremented twice).

To solve this problem, every process p maintains a local structure $M_p[q] \in \mathcal{S}$. When p issues and logs a PUT($p \rightrightarrows q$) such that PUT($p \rightrightarrows q$)$.combine = true$, it sets $M_p[q] := true$. When p deletes PUT($p \rightrightarrows q$) from its logs, it sets $M_p[q] := false$. If q fails, starts to recover, and sees that any $M_p[q] = true$, it stops the recovery and falls back to the coordinated checkpoint. This scheme is valid if access determinism is assumed. Otherwise we set $M_p[q] := true$ regardless of the value of PUT($p \rightrightarrows q$)$.combine$; we use the same approach if q can issue WRITEs to the memory regions accessed with remote PUTs parallel in $||_{co}$ to these WRITEs.

4.3 Recovering a Failed Process

We now describe a protocol for codes that synchronize with GSYNCs; consult the technical report for other schemes. Let us denote the failed process as p_f. We assume an underlying batch system that provides a new process p_{new} in the place of p_f, and that other processes resume their communication with p_{new} after it fully recovers. We illustrate the scheme in Algorithm 2. First, p_{new} fetches the checkpointed data. Second, p_{new} gets the logs of PUTs (put_logs) and GETs (get_logs) related to p_f (lines 3-11). It also checks if any $N_q[p_f] = true$ (see § 3.2.3) or $M_q[p_f] = true$ (see § 4.2), if yes it instructs all processes to roll back to the last coordinated checkpoint. The protocol uses LOCKs (lines 5,10) to prevent data races due to, e.g., concurrent recoveries and log cleanups by q.

The main part (lines 12-26) replays accesses causally. The recovery ends when there are no logs left (line 12; $|logs|$ is the size of the set "logs"). We first get the logs with the smallest $.GNC$ (line 13) to maintain $\xrightarrow{cohb}$ introduced by GSYNCs (see § 4.1 E). Then, within this step, we find the logs with minimum $.EC$ and $.GC$ to preserve $\xrightarrow{co}$ in issued PUTs and GETs, respectively (lines 17-18, see § 4.1 A, B). We replay them in lines 19-20.

4.4 Discussion

Our recovery scheme presents a trade-off between memory efficiency and time to recover. Process p_{new} fetches all related logs and only then begins to replay accesses. Thus, we assume that its memory has capacity to contain put_logs and get_logs; a reasonable assumption if the user program has regular communication patterns (true for most of today's RMA applications [19]). A more memory-efficient scheme fetches logs while recovering. This incurs performance issues as p_{new} has to access remote logs multiple times.

5. EXTENDING THE MODEL FOR MORE RESILIENCE

Our model and in-memory resilience schemes are oblivious to the underlying hardware. However, virtually all of today's systems have a hierarchical hardware layout (e.g., cores reside on a single chip, chips reside in a single node, nodes form a rack, and racks form a cabinet). Multiple elements may be affected by a single failure at a higher level, jeopardizing the safety of our protocols. We now extend our model to cover arbitrary hierarchies and propose *topology-*

aware mechanisms to make our schemes handle concurrent hardware failures. Specifically, we propose three following extensions:

The Hierarchy of Failure Domains A *failure domain* (FD) is an element of a hardware hierarchy that can fail (e.g., a node or a cabinet). FDs constitute an FD hierarchy (FDH) with h levels. An example FDH is shown in Figure 6, $h = 4$. We skip the level of single cores because in practice the smallest FD is a node (e.g., in the TSUBAME2.0 system failure history, there are no core failures [3]). Then, we define $\mathcal{H} = \bigcup_{1 \le j \le h} \left(\bigcup_{1 \le i \le H_j} H_{i,j} \right)$ to be the set of all the FD elements in an FDH. $H_{i,j}$ and H_j are element i of hierarchy level j and the number of such elements at level j, respectively. For example, in Figure 6 $H_{3,2}$ is the third blade (level 2) and $H_2 = 96$.

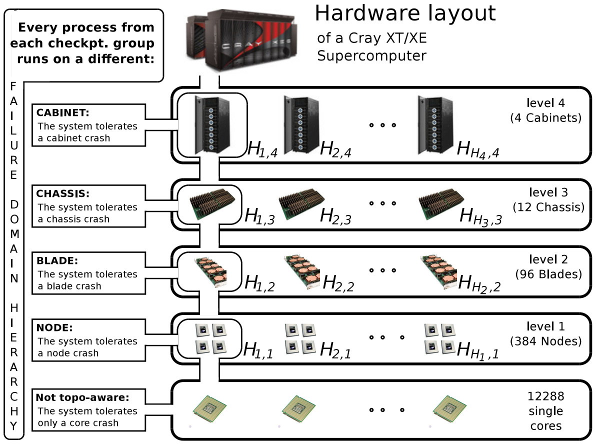


Figure 6: An example hardware layout (Cray XT/XE) and the corresponding FDH (§ 5). In this example, $h = 4$.

Groups of Processes To improve resilience, we split the process set $\mathcal{P}$ into g equally-sized groups G_i and add m *checksum* processes to each group to store checksums of checkpoints taken in each group (using, e.g., the Reed-Solomon [36] coding scheme). Thus, every group can resist m concurrent process crashes. The group size is $|G| = \frac{|\mathcal{P}|}{g} + m$.

New System Definition We now extend the definition of a distributed system $\mathcal{D}$ to cover the additional concepts:

$$\langle \mathcal{P}, \mathcal{E}, \mathcal{S}, \mathcal{H}, \mathcal{G}, \xrightarrow{po}, \xrightarrow{so}, \xrightarrow{hb}, \xrightarrow{co}, \mathscr{M} \rangle. \tag{5}$$

$\mathcal{G} = \{G_1, ..., G_g\}$ is a set of $\mathcal{G}$roups of processes and $\mathscr{M} : \mathcal{P} \times \mathbb{N} \to \mathcal{H}$ is a function that $\mathscr{M}$aps process p to the FD at hierarchy level k where p runs: $\mathscr{M}(p,k) = H_{j,k}$. $\mathscr{M}$ defines how processes are distributed over FDH. For example, if p runs on blade $H_{1,2}$ from Figure 6, then $\mathscr{M}(p,2) = H_{1,2}$.

5.1 Handling Multiple Hardware Failures

More than m process crashes in any group G_i result in a *catastrophic failure* (CF; we use the name from [8]) that incurs restarting the whole computation. Depending on how $\mathscr{M}$ distributes processes, such a CF may be caused by several (or even one) crashed FDs. To minimize the risk of CFs, $\mathscr{M}$ has to be *topology-aware* (t-aware): for a given level n (called a *t-awareness level*), no more than m processes from the same group can run on the same $H_{i,k}$ at any level $k, k \le n$:

$$\begin{aligned} &\forall p_1, p_2, ..., p_m \in \mathcal{P} \quad \forall G \in \mathcal{G} \quad \forall 1 \le k \le n : \\ &(p_1 \in G \land ... \land p_m \in G) \Rightarrow (\mathscr{M}(p_1,k) \ne ... \ne \mathscr{M}(p_m,k)) \end{aligned} \tag{6}$$

Figure 7 shows an example t-aware process distribution.

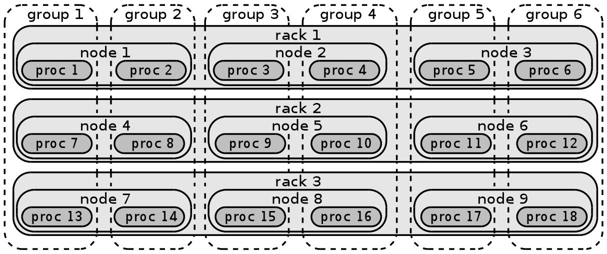


Figure 7: T-aware distribution at the node *and* rack level (§ 5.1).

5.2 Calculating Probability of a CF

We now calculate the probability of a catastrophic failure (P_{cf}) in our model. We later (§ 7.1) use P_{cf} to show that our protocols are resilient on a concrete machine (the TSUMABE2.0 supercomputer [3]). If a reader is not interested in the derivation details, she may proceed to Section 6 where we present the results. We set $m = 1$ and thus use the XOR erasure code, similar to an additional disk in a RAID5 [12]. We assume that failures at different hierarchy levels are independent and that any number x_j of elements from any hierarchy level j ($1 \le x_j \le H_j$, $1 \le j \le h$) can fail. Thus,

$$P_{cf} = \sum_{j=1}^{h} \sum_{x_j=1}^{H_j} P(x_j \cap x_{j,cf}) = \sum_{j=1}^{h} \sum_{x_j=1}^{H_j} P_j(x_j) P_j(x_{j,cf} | x_j). \tag{7}$$

$P(x_j \cap x_{j,cf})$ is the probability that x_j elements of the j hierarchy level will fail *and* result in a catastrophic failure. $P_j(x_j)$ is the probability of the failure of x_j elements from level j of the hierarchy. $P_j(x_{j,cf} | x_j)$ is the probability that x_j given concurrent failures at hierarchy level j are catastrophic to the system. It is difficult to analytically derive $P_j(x_j)$ as it is specific for every machine. For our example study (see Section 7.1) we use the failure rates from the TSUBAME2 failure history [3].

In contrast, $P_j(x_{j,cf} | x_j)$ can be calculated using combinatorial theory. Assume that $\mathscr{M}$ distributes processes in a t-aware way at levels 1 to n of the FDH ($1 \le n \le h$). First, we derive $P_j(x_{j,cf} | x_j)$ for any level j such that $1 \le j \le n$:

$$P_j(x_{j,cf} | x_j) = \frac{D_j \cdot \binom{|G|}{2} \cdot \binom{H_j - 2}{x_j - 2}}{\binom{H_j}{x_j}}. \tag{8}$$

$\binom{|G|}{2}$ is the number of the possible catastrophic failure scenarios *in a single group* ($m = 1$ thus any two process crashes in one group are catastrophic). D_j is the number of such single-group scenarios *at the whole level* j and is equal to $\left\lceil \frac{H_j}{|G|} \right\rceil$ (see Figure 8 for intuitive explanation). $\binom{H_j - 2}{x_j - 2}$ is the number of the remaining possible failure scenarios and $\binom{H_j}{x_j}$ is the total number of the possible failure scenarios. Second, for remaining levels j ($n + 1 \le j \le h$) $\mathscr{M}$ is *not* t-aware and thus in the worst-case scenario any element crash is catastrophic: $P_j(x_{j,cf} | x_j) = 1$. The final formula for P_{cf} is thus

$$P_{cf} = \sum_{j=1}^{n} \sum_{x_j=1}^{H_j} P_j(x_j) \frac{D \cdot \binom{|G|}{2} \cdot \binom{H_j - 2}{x_j - 2}}{\binom{H_j}{x_j}} + \sum_{j=n+1}^{h} \sum_{x_j=1}^{N_j} P_j(x_j). \tag{9}$$

6. HOLISTIC RESILIENCE PROTOCOL

We now describe an example conceptual implementation of holistic fault tolerance for RMA that we developed to understand the tradeoffs between the resilience and performance in RMA-based systems. We implement it as a

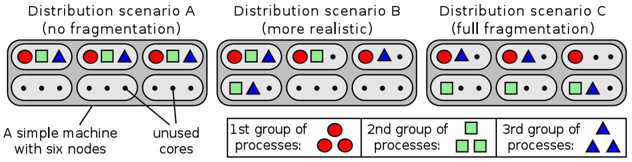


Figure 8: (§ 5.2) Consider three process distribution scenarios by $\mathscr{M}$ (*each* is t-aware). Optimistically, processes can be distributed contiguously (scenario A) or partially fragmented (scenario B). To get the upper bound for P_{cf} we use the worst-case pattern (scenario C). Now, to get the number of single-group CF scenarios at the whole level j (D_j), we need to obtain the number of the groups of *hardware elements* at j that hold process groups: $\lceil H_j/|G|\rceil$.

portable library (based on C and MPI) called FTRMA. We utilize MPI-3's one sided interface, but any other RMA model enabling relaxed memory consistency could be used instead (e.g., UPC or Fortran 2008). We use the publicly available FOMPI implementation of MPI-3 one sided as MPI library [1] but any other MPI-3 compliant library would be suitable. For simplicity we assume that the user application uses one contiguous region of shared memory of the same size at each process. Still, all the conclusions drawn are valid for any other application pattern based on RMA. Following the MPI-3 specification, we call this shared region of memory at every process a *window*. Finally, we divide user processes (referred to as CoMputing processes, CMs) into groups (as described in Section 5) and add one CHecksum process (denoted as CH) per group ($m = 1$). For any computing process p, we denote the CH in its group as $CH(p)$. CHs store and update XOR checksums of their CMs.

6.1 Protocol Overview

In this section we provide a general overview of the layered protocol implementation (see Figure 9). The first part (layer 1) logs accesses. The second layer takes uncoordinated checkpoints (called *demand* checkpoints) to trim the logs. Layer 3 performs regular coordinated checkpoints. All layers are diskless. Causal recovery replays memory accesses. Finally, our FDH increases resilience of the whole protocol.

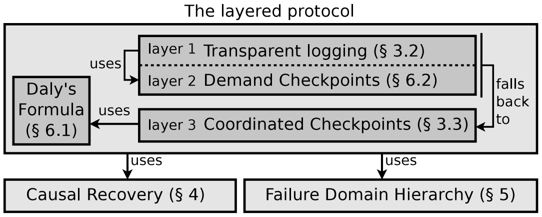

The layered protocol
layer 1 Transparent logging (§ 3.2)
layer 2 Demand Checkpoints (§ 6.2)
layer 3 Coordinated Checkpoints (§ 3.3)
uses
uses
falls back to
Daly's Formula (§ 6.1)
uses
uses
Causal Recovery (§ 4)
Failure Domain Hierarchy (§ 5)

Figure 9: The overview of the protocol (§ 6.1). Layer 1 and 2 constitute the uncoordinated part of the protocol that falls back to the coordinated checkpointing if logging fails or if its overhead is too high.

Daly's Interval Layer 3 uses Daly's formula [15] as the optimum interval between coordinated checkpoints: $\sqrt{2\delta M}\cdot[1+1/3\sqrt{\delta/(2M)}+(1/9)(\delta/(2M))]-\delta$ (for $\delta < 2M$), or M (for $\delta \geq 2M$). M is the MTBF (mean time between failures that FTRMA handles with coordinated checkpointing) for the target machine and δ is the time to take a checkpoint. The user provides M while δ is estimated by our protocol.

Interfacing with User Programs and Runtime FTRMA routines are called after each RMA action. This would entail runtime system calls in compiled languages and we use the PMPI profiling interface [31] in our implementation. During window creation the user can specify: (1) the number of CHs, (2) MTBF, (3) whether to use topology-awareness. After window creation, the protocol divides processes into CMs and CHs. If the user enables t-awareness, groups of processes running on the same FDs are also created. In the current version FTRMA takes into account computing nodes when applying t-awareness.

6.2 Demand Checkpointing

Demand checkpoints address the problem of diminishing amounts of memory per core in today's and future computing centers. If free memory at CM process p is scarce, p selects the process q with the largest $LP_p[q]$ or $LG_p[q]$ and requests a demand checkpoint. First, p sends a *checkpoint request* to $CH(q)$ which, in turn, forces q to checkpoint. This can be done by: closing all the epochs, locking all the relevant data structures, calculating the XOR checksum, and: (1) streaming the result to $CH(q)$ piece by piece or (2) sending the result in one bulk. $CH(q)$ integrates the received checkpoint data into the existing XOR checksum. Variant (1) is memory-efficient, and (2) is less time-consuming. Next, q unlocks all the data structures. Finally, $CH(q)$ sends a confirmation with the epoch number $E(p \to q)$ and respective counters (GNC_q, GC_q, SC_q) to p. Process p can delete logs of actions a where $a.EC < E(p \to q)$, $a.GNC < GNC_q$, $a.GC < GC_q$, $a.SC < SC_q$.

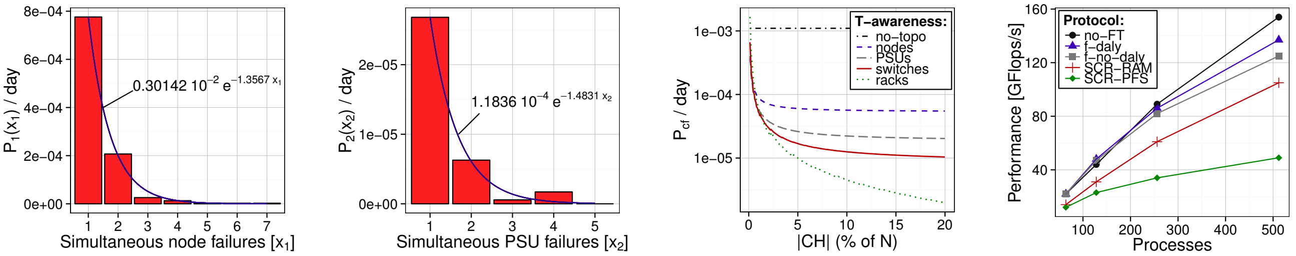
7. TESTING AND EVALUATION

In this section we first analyze the resilience of our protocol using real data from TSUBAME2.0 [3] failure history. Then, we test the performance of FTRMA with a NAS benchmark [14] that computes 3D Fast Fourier Transformation and a distributed key-value store. We denote the number of CHs and CMs as $|CH|$ and $|CM|$, respectively.

7.1 Analysis of Protocol Resilience

Our protocol stores all data in volatile memories to avoid I/O performance penalties and frequent disk and parallel file system failures [24,38]. This brings several questions on whether the scheme is resilient in practical environments. To answer this question, we calculate the probability of a catastrophic failure P_{cf} (using Equations (7) and (9)) of our protocol, applying t-awareness at different levels of FDH.

We first fix model parameters (H_j, h) to reflect the hierarchy of TSUBAME2.0. TSUBAME2.0 FDH has 4 levels [38]: nodes, power supply units (PSUs), edge switches, and racks ($h = 4$) [38]. Then, to get P_{cf}, we calculate distributions $P_j(x_j)$ that determine the probability of x_j concurrent crashes at level j of the TSUBAME FDH. To obtain $P_j(x_j)$ we analyzed 1962 crashes in the history of TSUBAME2.0 failures [3]. Following [8] we decided to use exponential probability distributions, where the argument is the number of concurrent failures x_j. We derived four probability density functions (PDFs) that approximate the failure distributions of nodes ($0.30142\cdot 10^{-2}e^{-1.3567x_1}$), PSUs ($1.1836\cdot 10^{-4}e^{-1.4831x_2}$), switches ($3.9249\cdot 10^{-5}e^{-1.5902x_3}$), and racks ($3.2257\cdot 10^{-5}e^{-1.5488x_4}$). The unit is failures per day. Figures 10a and 10b illustrate two PDF plots with histograms. The distributions for PSUs, switches, and racks are based on real data only. For nodes it was not always possible to determine the exact correlation of failures. Thus, we pessimistically assumed (basing on [8]) that single crashes constitute 75% of all node failures, two concurrent crashes constitute 20%, and other values decrease exponentially.



(a) Distribution of node crashes (samples and the fit) (§ 7.1).

(b) Distribution of PSU crashes (samples and the fit) (§ 7.1).

(c) Probability of a catastrophic failure (§ 7.1.1).

(d) NAS FFT (class C) fault-free runs: checkpointing (§ 7.2.1).

Figure 10: Distribution of PSU & node failures, P_{cf} in TSUBAME2.0 running 4,000 processes, and the performance of NAS 3D FFT.

7.1.1 Comparison of Resilience

Figure 10c shows the resilience of our protocol when using five t-awareness strategies. The number of processes N is 4,000. P_{cf} is normalized to one day period. Without t-awareness (no-topo) a single crash of any FD of TSUBAME2.0 is catastrophic, thus P_{cf} does not depend on $|CH|$. In other scenarios every process from every group runs on a different node (nodes), PSU (PSUs), switch enclosure (switches) and rack (racks). In all cases P_{cf} decreases proportionally to the increasing $|CH|$, however at some point the exponential distributions ($P_j(x_j)$) begin to dominate the results. Topology-awareness at higher hierarchy levels significantly improves the resilience of our protocol. For example, if $CH = 5\%N$, P_{cf} in the switches scenario is ≈4 times lower than in nodes. Furthermore, all t-aware schemes are 1-3 orders of magnitude more resilient than no-topo.

The results show that even a simple scheme (nodes) significantly improves the resilience of our protocol that performs only in-memory checkpointing and logging. We conclude that costly I/O flushes to the parallel file system (PFS) are not required for obtaining a high level of resilience. On the contrary, such flushes may even *increase* the risk of failures. They usually entail stressing the I/O system for significant amounts of time [38], and stable storage is often the element most susceptible to crashes. For example, a Blue Gene/P supercomputer had 4,164 disk fail events in 2011 (for 10,400 total disks) [24], and its PFS failed 77 times, almost two times more often than other hardware [24].

7.2 Analysis of Protocol Performance

We now discuss the performance of our fault tolerance protocol after the integration with two applications: NAS 3D FFT and a distributed key-value store. Both of these applications are characterized by intensive communication patterns, thus they demonstrate worst-case scenarios for our protocol. Integrating FTRMA with the application code was trivial and required minimal code changes resulting in the same code complexity.

Comparison to Scalable Checkpoint/Restart We compare FTRMA to Scalable Checkpoint-Restart (SCR) [2], a popular open-source message passing library that provides checkpoint and restart capability for MPI codes but does not enable logging. We turn on the XOR scheme in SCR and we fix the size of SCR groups [2] so that they match the analogous parameter in FTRMA ($|G|$). To make the comparison fair, we configure SCR to save checkpoints to both in-memory tmpfs (SCR-RAM) and to the PFS (SCR-PFS).

Comparison to Message Logging To compare the logging overheads in MP and RMA we also developed a simple message logging (ML) scheme (basing on the protocol from [37]) that records accesses. Similarly to [37] we use additional processes to store protocol-specific access logs; the data is stored at the sender's or receiver's side depending on the type of operation.

We execute all benchmarks on the Monte Rosa system and we use Cray XE6 computing nodes. Each node contains four 8-core 2.3 GHz AMD Opterons 6276 (Interlagos) and is connected to a 3D-Torus Gemini network. We use the Cray Programming Environment 4.1.46 to compile the code.

7.2.1 NAS 3D Fast Fourier Transformation

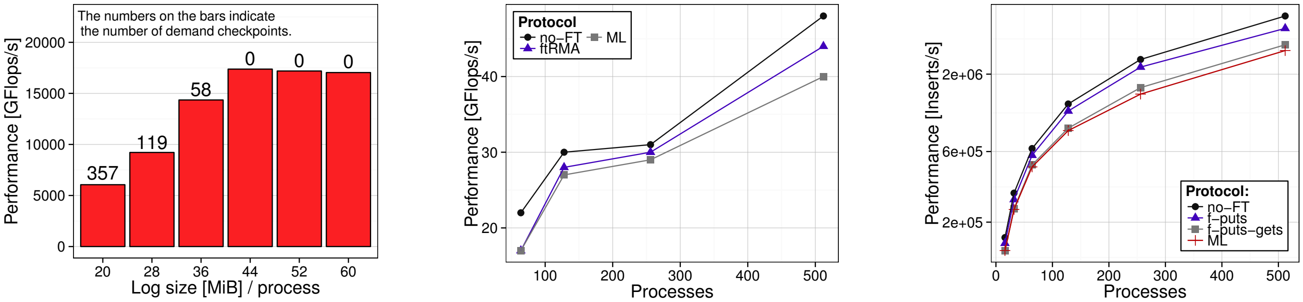
Our version of the NAS 3D FFT [14] benchmark is based on MPI-3 nonblocking PUTs (we exploit the overlap of computation and communication). The benchmark calculates 3D FFT using a 2D decomposition.

Performance of Coordinated Checkpointing We begin with evaluating our checkpointing "Gsync" scheme. Figure 10d illustrates the performance of NAS FFT fault-free runs. We compare: the original application code without any fault-tolerance (no-FT), FTRMA, SCR-RAM, and SCR-PFS. We fix $|CH| = 12.5\%|CM|$. We include two FTRMA scenarios: f-daly (use Daly's formula for coordinated checkpoints), and f-no-daly (fixed frequency of checkpoints without Daly's formula, ≈2.7s for 1024 processes). We use the same t-awareness policy in all codes (nodes). The tested schemes have the respective fault-tolerance overheads over the baseline no-FT: 1-5% (f-daly), 1-15% (f-no-daly), 21-37% (SCR-RAM) and 46-67% (SCR-PFS). The performance of SCR-RAM is lower than f-daly and f-no-daly because FTRMA is based on the Gsync scheme that incurs less synchronization. SCR-PFS entails the highest overheads due to costly I/O flushes.

Performance of Demand Checkpointing We now analyze how the size of the log impacts the number of demand checkpoints and the performance of fault-free runs (see Figure 11a). Dedicating less than 44 MiB of memory for storing logs (per process) triggers demand checkpoint requests to clear the log; checkpoints are taken every ≈0.25s on average (when the size of the log is 36 MiB). This results in performance penalties but leaves more memory available to the the user.

Performance of Access Logging As the next step we evaluate our logging scheme. Figure 11b illustrates the performance of fault-free runs. We compare no-FT, FTRMA, and our ML protocol (ML). FTRMA adds only ≈8-9% of overhead to the baseline (no-FT) and consistently outperforms ML by ≈9% due to the smaller amount of protocol-specific interaction between processes.

Varying |CH| and T-Awareness Policies Here, we analyze how $|CH|$ and t-awareness impact the performance of NAS FFT fault-free runs. We set $|CH| = 12.5\%|CM|$ and



(a) NAS FFT (class A) fault-free runs: demand checkpointing.

(b) NAS FFT (class A) fault-free runs: logging.

(c) Key-value store fault-free runs.

Figure 11: Performance of the NAS FFT code (§ 7.2.1) and the key-value store (§ 7.2.2).

$|CH| = 6.25\%|CM|$, and we use the no-topo and nodes t-awareness policies. The results show that all these schemes differ negligibly from no-FT by 1-5%.

7.2.2 Key-Value Store

Our key-value store is based on a simple distributed hashtable (DHT) that stores 8–Byte integers. The DHT consists of parts called *local volumes* constructed with fixed-sized arrays. Every local volume is managed by a different process. Inserts are based on MPI-3 atomic Compare-And-Swap and Fetch-And-Op functions. Elements after hash collisions are inserted in the overflow heap that is the part of each local volume. To insert an element, a thread atomically updates the pointers to the next free cell and the last element in the local volume. Memory consistency is ensured with flushes. One GET and one PUT are logged if there is no hash collision, otherwise 6 PUTs and 4 GETs are recorded.

Performance of Access Logging We now measure the relative performance penalty of logging PUTs and GETs. During the benchmark, processes insert random elements with random keys. We focus on inserts only as they are perfectly representative for the logging evaluation. To simulate realistic requests, every process waits for a random time after every insert. The function that we use to calculate this interval is based on the exponential probability distribution: $f\delta e^{-\delta x}$, where f is a scaling factor, δ is a rate parameter and $x \in [0; b)$ is a random number. The selected parameter values ensure that every process spends ≈5-10% of the total runtime on inserting elements. For many computation-intense applications this is already a high amount of communication. We again compare no-FT, ML, and two FTRMA scenarios: f-puts (logging only PUTs) and f-puts-gets (logging PUTs and GETs). We fix $|CH| = 12.5\%|CM|$ and use the nodes t-awareness. We skip SCR as it does not enable logging.

We present the results in Figure 11c. For $N = 256$, the logging overhead over the baseline (no-FT) is: ≈12% (f-puts), 33% (f-gets), and 40% (ML). The overhead of logging PUTs is due to the fact that every operation is recorded directly after issuing. Traditional message passing protocols suffer from a similar effect [17]. The overhead generated by logging GETs in f-puts-gets and ML is more significant because, due to RMA's one-sided semantics, every GET has to be recorded remotely. In addition, f-puts-gets suffers from synchronization overheads (caused by concurrent accesses to LG), while ML from inter-process protocol-specific communication. Discussed overheads heavily depend on the application type. Our key-value store constitutes a worst-case scenario because it does not allow for long epochs that could enable, e.g., sending the logs of multiple GETs in a bulk. The performance penalties would be smaller in applications that overlap computation with communication and use non blocking GETs.

8. RELATED WORK

In this section we discuss existing checkpointing and logging schemes (see Figure 12). For excellent surveys, see [6, 17, 40]. Existing work on fault tolerance in RMA/PGAS is scarce, an example scheme that uses PGAS for data replication can be found in [5].

8.1 Checkpointing Protocols

These schemes are traditionally divided into *uncoordinated*, *coordinated*, and *communication induced*, depending on process coordination scale [17]. There are also *complete* and *incremental* protocols that differ in checkpoint sizes [40].

Uncoordinated Schemes Uncoordinated schemes do not synchronize while checkpointing, but may suffer from the *domino effect* or complex recoveries [17]. Example protocols are based on *dependency* [9] or *checkpoint graphs* [17]. A recent scheme targeting large-scale systems is Ken [41].

Coordinated Schemes Here, processes synchronize to produce consistent global checkpoints. There is no domino effect and recovery is simple but synchronization may incur severe overheads. Coordinated schemes can be *blocking* [17] or *non-blocking* [11]. There are also schemes based on *loosely synchronized clocks* [39] and *minimal coordination* [26].

Communication Induced Schemes Here, senders add scheme-specific data to application messages that receivers use to, e.g., avoid taking useless checkpoints. These schemes can be *index-based* [21] or *model-based* [17, 32].

Incremental Checkpointing An incremental checkpoint updates only the data that changed since the previous checkpoint. These protocols are divided into page-based [40] and hash-based [4]. They can reside at the level of an *application*, a *library*, an *OS*, or *hardware* [40]. Other schemes can be *compiler-enhanced* [10] or *adaptive* [4].

Others Recently, *multi-level* checkpointing was introduced [8, 30, 38]. *Adaptive* checkpointing based on failure prediction is discussed in [28]. [35] presents diskless checkpointing. Other interesting schemes are based on: Reed-Solomon coding [8], cutoff and compression to reduce checkpoint sizes [23], checkpointing on clouds [33], reducing I/O bottlenecks [25], and performant checkpoints to PFS [7].

8.2 Logging Protocols

Logging enables restored processes to replay their execution beyond the most recent checkpoint. Log-based pro-

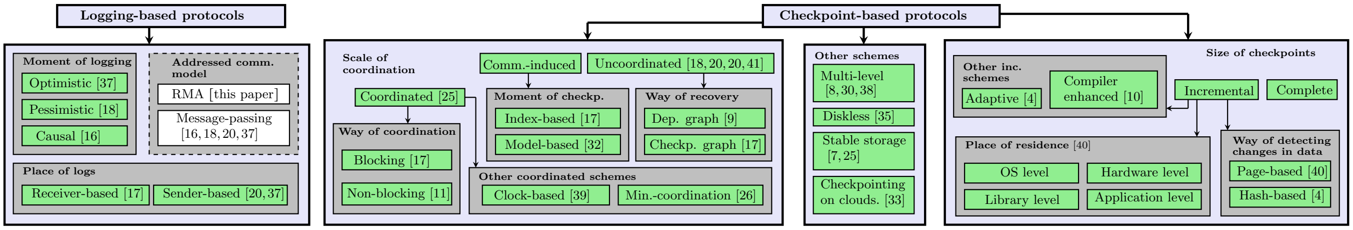


Figure 12: An overview of existing checkpointing and logging schemes (§ 8). A dashed rectangle illustrates a new sub-hierarchy introduced in the paper: dividing the logging protocols with respect to the *communication model* that they address.

tocols are traditionally categorized into: *pessimistic*, *optimistic*, *causal* [17]; they can also be *sender-based* [20,37] and *receiver-based* [17] depending on which side logs messages.

Pessimistic Schemes Such protocols log events before they influence the system. This ensures no orphan processes and simpler recovery, but may incur severe overheads during fault-free runs. An example protocol is V-MPICH [18].

Optimistic Schemes Here, processes postpone logging messages to achieve, e.g., better computation-communication overlap. However, the algorithms for recovery are usually more complicated and crashed processes may become orphans [17]. A recent scheme can be found in [37].

Causal Schemes In such schemes processes log and exchange (by piggybacking to messages) dependencies needed for recovery. This ensures no orphans but may reduce bandwidth [17]. An example protocol is discussed in [16].

8.3 Other Important Studies & Discussion

Deriving an optimum checkpointing interval is presented in [15]. Formalizations targeting resilience can be found in [17,32]. *Containment domains* for encapsulating failures within a hierarchical scope are discussed in [13]. Modeling and prediction of failures is addressed in [8, 13]. Work on send determinism in MP can be found in [20].

Our study goes beyond the existing research scope presented in this section. First, we develop a fault tolerance model that covers virtually whole rich RMA semantics. Other existing formalizations (e.g., [6, 17, 32]) target MP only. We then use the model to formally analyze why resilience for RMA differs from MP and to design checkpointing, logging, and recovery protocols for RMA. We identify and propose solutions to several challenges in resilience for RMA that *do not* exist in MP, e.g.: consistency problems caused by the relaxed RMA memory model (§ 3.1, § 3.2.2, § 3.2.3), access non-determinism (§ 4.2), issues due to one-sided RMA communication (§ 3.2.1), logging multiple RMA-specific orders (§ 4.1), etc. Our model enables proving correctness of proposed schemes; all proofs omitted due to space constraints can be found in the technical report. Extending our model for arbitrary hardware hierarchies generalizes the approach from [8] and enables formal reasoning about crashes of hardware elements and process distribution. Finally, our protocol leverages and combines several important concepts and mechanisms (Daly's interval [15], multi-level design [30], etc.) to improve the resilience of RMA systems even further and is the first implementation of holistic fault tolerance for RMA.

9. CONCLUSION

RMA programming models are growing in popularity and importance as they allow for the best utilization of hardware features such as OS-bypass or zero-copy data transfer. Still, little work addresses fault tolerance for RMA.

We established, described, and explored a complete formal model of fault tolerance for RMA and illustrated how to use it to design and reason about resilience protocols running on flat and hierarchical machines. It will play an important role in making emerging RMA programming fault tolerant and can be easily extended to cover, e.g., stable storage.

Our study does not resort to traditional less scalable mechanisms that often rely on costly I/O flushes. The implementation of our holistic protocol adds negligible overheads to the applications runtime, for example 1-5% for in-memory checkpointing and 8% for fully transparent logging of remote memory accesses in the NAS 3D FFT code. Our probability study shows that the protocol offers high resilience. The idea of demand checkpoints will help alleviate the problem of limited memory amounts in today's petascale and future exascale computing centers.

Finally, our work provides the basis for further reasoning about fault-tolerance not only for RMA, but also for all the other models that can be constructed upon it, such as task-based programming models. This will play an important role in complex heterogeneous large-scale systems.

10. REFERENCES

[1] foMPI, 2013. `http://spcl.inf.ethz.ch/Research/Parallel_Programming/foMPI`.

[2] Scalable Checkpoint / Restart, 2013. `http://sourceforge.net/projects/scalablecr/`.

[3] TSUBAME2.0: Failure History, April 2013. `http://mon.g.gsic.titech.ac.jp/trouble-list/index.htm`.

[4] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proc. of the Ann. Intl. Conf. on Supercomp.*, ICS '04, pages 277–286, 2004.

[5] N. Ali, S. Krishnamoorthy, N. Govind, and B. Palmer. A Redundant Communication Approach to Scalable Fault Tolerance in PGAS Programming Models. In *Par., Dist. and Net. Proc. (PDP), the Eur. Intl. Conf. on*, pages 24–31, 2011.

[6] L. Alvisi and K. Marzullo. Message Logging: Pessimistic, Optimistic, Causal, and Optimal. *IEEE Trans. Softw. Eng.*, 24(2):149–159, Feb. 1998.

[7] D. Arteaga and M. Zhao. Towards Scalable Application Checkpointing with Parallel File System Delegation. In *Proc. of the IEEE Intl. Conf. on Net., Arch., and Stor.*, NAS '11, pages 130–139, 2011.

[8] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. FTI: high performance Fault Tolerance Interface for hybrid systems. In *Proc. of the ACM/IEEE Supercomputing*, SC '11, pages 32:1–32:32.

[9] B. Bhargava and S.-R. Lian. Independent checkpointing and concurrent rollback for recovery in distributed systems-an optimistic approach. In *Rel. Dist. Syst., 1988. Proc.., Symp. on*, pages 3 –12.

[10] G. Bronevetsky, D. J. Marques, K. K. Pingali, R. Rugina, and S. A. McKee. Compiler-enhanced incremental checkpointing for OpenMP applications. In *Proc. of the ACM SIGPLAN Symp. on Prin. and Prac. of Par. Prog.*, PPoPP '08, pages 275–276.

[11] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.

[12] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Comput. Surv.*, 26(2):145–185, June 1994.

[13] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez. Containment domains: a scalable, efficient, and flexible resilience scheme for exascale systems. In *Proc. of the ACM/IEEE Supercomputing*, SC '12, pages 58:1–58:11.

[14] D. H. Bailey et al. The NAS parallel benchmarks. Technical report, The Intl. J. of Super. App., 1991.

[15] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312, Feb. 2006.

[16] E. Elnozahy and W. Zwaenepoel. Manetho: transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *Comp., IEEE Trans. on*, 41(5):526 –531, 1992.

[17] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, Sept. 2002.

[18] G. Bosilca et al. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *Supercomputing, the ACM/IEEE Conf.*, pages 29–29, 2002.

[19] R. Gerstenberger, M. Besta, and T. Hoefler. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. In *Proc. of the ACM/IEEE Supercomputing*, SC '13, pages 53:1–53:12, 2013.

[20] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello. Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic MPI Applications. In *Par. Dist. Proc. Symp., the IEEE Intl.*, pages 989–1000.

[21] J.-M. Helary, A. Mostefaoui, R. Netzer, and M. Raynal. Preventing useless checkpoints in distributed computations. In *Rel. Dist. Sys., 1997. Proc.., the Symp. on*, pages 183 –190, 1997.

[22] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, and K. Underwood. Remote Memory Access Programming in MPI-3. *Argonne National Laboratory, Tech. Rep*, 2013.

[23] S. Hogan, J. Hammond, and A. Chien. An evaluation of difference and threshold techniques for efficient checkpoints. In *Dep. Sys. and Net. Work. (DSN-W), IEEE/IFIP Intl. Conf.*, pages 1–6, 2012.

[24] F. Isaila, J. Garcia, J. Carretero, R. Ross, and D. Kimpe. Making the case for reforming the I/O software stack of extreme-scale systems. In *Ex. App. and Soft. Conf. (EASC'13)*, 2013.

[25] H. Jin and K. Hwang. Distributed Checkpointing on Clusters with Dynamic Striping and Staggering. In *Advances in Computing Science - ASIAN 2002*, volume 2550, pages 19–33. 2002.

[26] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *Soft. Eng., IEEE Trans. on*, SE-13(1):23 – 31, 1987.

[27] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. ACM*, 21(7):558–565, 1978.

[28] Y. Li and Z. Lan. Using adaptive fault tolerance to improve application robustness on the Teragrid. *Proc. of TeraGrid*, 322, 2007.

[29] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *Proc. of ACM Symp. on Prin. of Prog. Lang.*, POPL '05, pages 378–391, 2005.

[30] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *Proc. of ACM/IEEE Supercomputing*, SC '10, pages 1–11.

[31] MPI Forum. MPI: A Message-Passing Interface Standard. Version 3, September 2012. available at: http://www.mpi-forum.org (Sep. 2012).

[32] R. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots. *Par. and Dist. Sys., IEEE Trans. on*, 6(2):165 –169, 1995.

[33] B. Nicolae and F. Cappello. BlobCR: efficient checkpoint-restart for HPC applications on IaaS clouds using virtual disk image snapshots. In *Proc. of ACM/IEEE Supercomputing*, SC '11, pages 34:1–34:12.

[34] D. Petrović, O. Shahmirzadi, T. Ropars, and A. Schiper. High-performance RMA-based broadcast on the Intel SCC. In *Proc. of ACM Symp. Par. Alg. Arch.*, SPAA '12, pages 121–130, 2012.

[35] J. Plank, K. Li, and M. Puening. Diskless checkpointing. *Par. and Dist. Sys., IEEE Trans. on*, 9(10):972–986, 1998.

[36] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *J. of the Soc. for Indust. & Appl. Math.*, 8(2):300–304, 1960.

[37] R. Riesen, K. Ferreira, D. Da Silva, P. Lemarinier, D. Arnold, and P. G. Bridges. Alleviating scalability issues of checkpointing protocols. In *Proc. of ACM/IEEE Supercomputing*, SC '12, pages 18:1–18:11.

[38] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, and S. Matsuoka. Design and modeling of a non-blocking checkpointing system. In *Proc. of the ACM/IEEE Supercomputing*, SC '12, pages 19:1–19:10.

[39] Z. Tong, R. Y. Kain, and W. T. Tsai. Rollback Recovery in Distributed Systems Using Loosely Synchronized Clocks. *IEEE Trans. Par. Dist. Sys.*, 3(2):246–251, 1992.

[40] M. Vasavada, F. Mueller, P. H. Hargrove, and E. Roman. Comparing different approaches for incremental checkpointing: The showdown. In *Linux Symposium*, page 69, 2011.

[41] S. Yoo, C. Killian, T. Kelly, H. K. Cho, and S. Plite. Composable Reliability for Asynchronous Systems. In *Proc. of the USENIX Ann. Tech. Conf.*, USENIX