Crosscheck:
Hardening replicated multithreaded services

Arthur Martens
TU Braunschweig

Christoph Borchert
TU Dortmund

Tobias Oliver Geißler
TU Braunschweig

Daniel Lohmann
FAU Erlangen-Nürnberg

Olaf Spinczyk
TU Dortmund

Rüdiger Kapitza
TU Braunschweig

***Abstract*—State-machine replication has received widespread attention for the provisioning of highly available services in data centers. However, current production systems focus on tolerating crash faults only and prominent service outages caused by state corruptions have indicated that this is a risky strategy. In the future, state corruptions due to transient faults (such as bit flips) become even more likely, caused by ongoing hardware trends regarding the shrinking of structure sizes and reduction of operating voltages.**

In this paper we present CROSSCHECK, an approach to tolerate arbitrary state corruption (ASC) in the context of fault-tolerant replication of multithreaded services. CROSSCHECK is able to detect silent data corruptions ahead of execution, and by crosschecking state changes with co-executing replicas, even ASCs can be detected. Finally, fault tolerance is achieved by a fine-grained recovery using fault-free replicas. Our implementation is transparent to the application by utilizing fine-grained software-hardening mechanisms using aspect-oriented programming. To validate CROSSCHECK we present a replicated multithreaded key-value store that is resilient to state corruptions.

I. INTRODUCTION

State-machine replication (SMR) is an established means for implementing highly available services in data centers. Prominent examples are coordination services [1], highly available data storage [2], but also wide-area replication of databases [3]. Existing production systems are usually limited in that they tolerate only faults that lead to crash faults. Prominent service outages due to state inconsistencies[1] causing faulty service behavior as well as recent studies [4], [5] indicate that state corruptions need to be addressed for providing highly available services. The latter gains even more importance given that future hardware will be even less reliable due to the shrinking structure sizes, increased clock frequencies, and reduced operating voltages [6].

One way to cope with these problems is tolerating arbitrary faults by means of Byzantine fault tolerance (BFT) [7]. However, despite recent research progress, BFT is more complex and resource demanding than plain crash-tolerant replication schemes and goes far beyond tolerating arbitrary state corruptions [8], [9], [10]. To provide a tailored solution there is a trend to make crash-tolerant systems resilient to state corruptions [11], [12]. In production systems, this is usually handled in an ad-hoc manner by manually introducing checksums for guarding critical data. However, this is a laborious task and incidences, like the Amazon S3 outage in 2007, indicate that in some cases important data has not been protected against state corruptions. Recently, Correia et. al. [11] proposed to cure such problems by extending replicas using mechanisms that contain and detect errors locally by checksums and redundant execution. A successive work by Behrens et. al. [12] focuses on reducing the memory overhead of the previous approach, but still relies on double execution per replica. Both works are not strictly tied to state-machine replication. Therefore, they do not exploit the full potential of co-executing replicas.

This paper presents CROSSCHECK, an approach that hardens state-machine replication against ASCs and explicitly handles replicated multithreaded services, thereby enabling the implementation of resilient highly available services for data centers. Our approach is transparent to the replicated service by utilizing *aspect-oriented programming* [13] and is specifically tailored to state-machine replication. Ahead of executing, CROSSCHECK detects silent data corruption (SDC) by introducing checksums into data structures, that is, *objects* in the terminology of object-oriented programming. However, this only protects objects in memory and fails guarding objects during modification. But, if deterministic execution is enforced, the generated checksums can be exploited to *crosscheck* state changes with co-executing replicas. Due to these measures and periodic self-checks, ASCs are detected in an early stage, thereby preventing corruptions to spread throughout a service, which reduces the risk of failed replicas. In case of detected state corruptions, the effected state can be efficiently recovered from fault-free replicas at the granularity of objects. Finally, CROSSCHECK makes use of STORYBOARD [14], an infrastructure that enforces deterministic execution in multithreaded environments based on the concept of schedule memorization. For validating CROSSCHECK, we present an early prototype of a replicated multithreaded key-value store based on a C++ version of memcached that is resilient to ASC. This was achieved by an automated application of object checksums using the AspectC++ compiler [15] and a request-centric memorization of critical sections using STORYBOARD. Our results indicate that memory-intensive applications, such

[1] http://status.aws.amazon.com/s3-20080720.html

as a key-value store, can efficiently be hardened.

In the remainder of the paper, we first define a system model in Section II. Next, we present the core concept of Crosscheck followed by a description of our current prototype in Section IV. Finally, Section VI details related approaches and Section VII concludes the paper.

II. System Model and Assumptions

Crosscheck aims at harding multithreaded services against arbitrary state corruptions. To apply Crosscheck, we require that the in-memory service state is composed of objects, which we denote as *state objects*. During the execution of a request, commands may perform any kind of access (i.e., read, write, create, and delete) to an arbitrary number of state objects. Such a structure is commonly provided when an object-oriented programming language is used.

To address multi-core hardware and recent service implementations, we assume a multithreaded service design where multiple requests can be executed concurrently. Access to critical data sections is guarded by atomic locks (i.e., mutex locks). As a consequence, without further measures, the execution order of multiple requests is non-deterministic in commodity systems, as the order depends on the internal scheduling policy that is influenced by the workload at time of execution. Information is exchanged via message passing, which may fail completely, corrupt or delay messages. Also, messages sent may arrive in a different order.

To tolerate faults, such as node crashes, Crosscheck utilizes *state-machine replication (SMR)* [16]. Thereby, liveness is ensured while tolerating f crash faults with $2f+1$ replicated instances. Given the same ordered set of requests and the same initial state, replicas execute the requests in the same order, transfer into the same state, and externalize the same output.

To implement the requirements of SMR, we need to enforce total ordering of messages and a deterministic execution of the service replicas. The former is achieved by an agreement protocol (e.g., Paxos [17]) or group communication framework (e.g., Spread [18]) supporting total ordering of messages. The latter requires deterministic execution of threads. In the case of Crosscheck this is achieved by applying Storyboard that follows the idea of scheduling memorization.

Beside node crashes, Crosscheck tolerates state corruptions, which can lead to externalization of faulty results and control-flow corruption altering the behavior of the service. In line with previous work, we do not expect this to happen arbitrarily often [11], that is, at most f out $2f+1$ replicas can be faulty at the same time. Thereby, we make no assumptions on the number of state object that can be corrupted. A node crash represents a special case, where all state objects of the crashed machine are considered as faulty. As Crosscheck provides means to recover state objects from fault-free replicas, an arbitrary number of state corruptions can be tolerated over time.

In accordance to current best practice, we require all data that is exchanged via messages between replicas and clients to be protected by checksums as well, to detect corrupt messages and enforce a retransmit.

Furthermore, ASC may also affect the underlying operating system and the ordering of messages itself (i.e., the agreement protocol). As both components are not directly protected by Crosscheck, we assume additional hardening measures for them [19].

III. The Crosscheck Approach

Our approach can be split into three tasks: First, Crosscheck enables the detection of data corruptions as service state is accessed. Second, Crosscheck is able to track state changes during request execution by means of checksums, thereby enabling efficient crosschecking amongst co-executing replicas. Third, corrupted state objects can be recovered from the remaining fault-free replicas.

A. Detecting silent data corruptions

As described above, we assume that all in-memory service state that needs to be protected from data corruptions is captured by *state objects*. Robustness against data corruptions is achieved by *generic object protection (GOP)* [20], which is an automated, compiler-based approach using aspect-oriented programming (AOP). With AOP it is possible to augment the code of an existing program by giving *advice* to a *pointcut*, which defines specific positions in the static structure (classes) and running control flow (execution of methods). The combination of advice and pointcut builds an *aspect*, which, thus, concentrates functionality in a single module that otherwise would be scattered and possibly duplicated across multiple locations.

In case of Crosscheck, all relevant service classes are automatically extended by additional data members that store redundancy. Furthermore, member functions (methods) that compute (update()) and verify (check()) the redundancy are introduced. Whenever a call to any original method of a *state object* is performed, it is modified by the aspect compiler, as depicted in the following listing:

```
stateObj.check();
stateObj.accessMembers(); //original call
stateObj.update();
```

The method check() is called before the original method call is executed. It validates the *state object* by comparing the introduced redundancy with the object's real data. If both diverge an SDC has been detected and the affected *state object* needs to be recovered (see Section III-C). In case no error is detected, the original call to the method is performed. Afterwards update() is invoked, which recomputes the *state object's* redundancy and stores it inside the object itself. On a read-only access (a call to a C++ method qualified as const), the update() operation is omitted for performance and reliability reasons. Likewise, the check() operation may be omitted for write-only access. However, the developer needs to provide a specific pointcut that describes all write-only methods.

The generic object protection offers to choose from various levels of redundancy, such as an error-detecting checksum or CRC32 code, full object duplication for instant error correction, and other mechanisms. In this paper, we apply an error-detecting CRC32 code, implemented by the SSE 4.2 instruction set provided by recent x86 processors. However, small *state objects* can be additionally duplicated without incurring serious memory overhead. We will use the term “checksum” in an exchangeable way for describing the CRC32 code – in conformity with the common use of the term “checksum” – even if not mathematically accurate.

The recent version of the generic object protection supports non-blocking synchronization on checksum operations, that is, no additional locks are needed for multithreaded applications.

B. Crosscheck state changes

Using the aforementioned generic object protection, SDCs can efficiently be detected ahead of execution and, depending on the severeness, immediately resolved (see Figure 1 ①). However, this does not protect the service state against corruptions that take place during execution ② and, even more problematic, faults during this phase remain undetected. To address this issue we *crosscheck* state changes with co-executing replicas. As part of the SDC detection, each time a *state object* is modified, its checksum is updated. These updates incorporate all state changes caused by a request and can be exploited to validate execution and control flow.

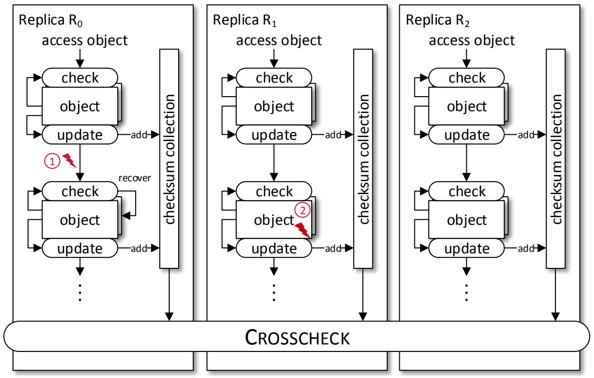


Fig. 1. CROSSCHECK approach

As depicted in Figure 1, each time we update the checksum during the execution of a request, this checksum C_i is added to a *checksum collection* (CC) together with a unique object id, $i \in I = \{1 \ldots n\}$ that is equal across all replicas for the corresponding object. This id fits two purposes: First, we can efficiently compare objects across replica boundaries. Second, it enables us to detect control-flow changes as divergent execution flows result in a different sequence of checksums or divergent checksum collections. We also add a reference to the *state object* exhibiting the checksum and store it in the CC. In case of recovery this enables the identification of corrupted state objects (see Section III-C).

```
1  stateValidation(){
2    broadcast(<CHECK,CC,RM,RC>);
3    while(quorum < QUORUM_NEEDED) {
4      chk=receiveCheckMsg();
5      //compare to own:
6      if( vrfyLocal({chk.CC, chk.RC}) )
7        quorum++;
8      else
9        orderedBroadcast(<RECOVER>);
10     //compare to others:
11     foreach( msgSet.elements() ) {
12       if( vrfyMsgSet({chk.CC, chk.RC}) )
13         quorum++;
14       else
15         orderedBroadcast(<RECOVER>);
16     }
17     msgSet.add(chk);
18   }
19   if(isClientConnected == true)
20     externalize(<REPLY, $RM_u$, $RC_v$>);
21 }
```

Fig. 2. CROSSCHECK state validation algorithm

After executing the request, all replicas perform a state validation as shown in Figure 2. Initially all replicas broadcast their checksum collection CC, together with a potential client reply message RM and its checksum RC as <CHECK, CC, RM, RC> to the other replicas. As *state object* references are only valid at the origin, these are excluded from the <CHECK> message.

On each receipt of a <CHECK> message, a replica calls vrfyLocal() to verify its checksums (all C_i in CC and RC) and the modified state object ids by comparison with the received data from the remote replica (Line 6). Additionally, all received messages are compared to each other by vrfyMsgSet() (Line 12). Whenever an error is detected (Line 9 and 15), a <RECOVER> request is send via the ordering stage. This is required for recovery and causes all replicas eventually to enter a quiescent state by finishing all running executions. Meanwhile, error detection proceeds until a quorum of f+1 matching checksums is gathered.

At this point we are able to distinguish the corrupted from the correct-working replicas and the affected *state objects* which are added to a list for later recovery. Since the reply messages RM are also exchanged, the replica responsible for the client connection may externalize a correct reply to the client even in the presence of a fault. As additional faults may occur during reply to the client, the <REPLY, RM_u, RC_v> may contain RM_u and RC_v from any validated replica u and v. This procedure is also described in [11] and enables client-side error detection (if supported).

C. Recovering from faults

As outlined in our system model (see Section II), we consider state corruptions of one or multiple state objects in f replicas. This leaves a pool of $f+1$ fault-free replicas which can be used for recovery. State corruptions are either detected ahead of state object access (see Section III-A), or as part of crosschecking the execution (see Section III-B). In the latter case, control-flow errors are also detected beside plain state corruptions. In the most simple case, recovery from state corruptions that are detected ahead of access can be handled by the generic object protection, if object state duplication has been applied.

In the absence of a local object-state copy, the affected state object has to be requested from any fault-free replica. However, since the remaining fault-free replicas might already have successfully passed and modified their fault-free copy of the affected state object, there is no state object version available that enables the direct replacement and continuation of the effected request execution at the faulty replica. The same problems arise if both state corruptions and control-flow errors are detected during the crosscheck phase.

Therefore, we use a synchronization model where the remaining fault-free replicas finish the execution of ongoing requests to provide updates for the faulty replica. In a naive implementation we would simply compare all state object checksums and replace faulty ones, however, this would be time consuming. To minimize the overhead, we focus on the deltas between the co-executing replicas that are determined by their recent execution history.

If a state corruption is detected, we proceed as follows: First, all replicas need to reach a quiescent state. Therefore, once a fault is detected, a <RECOVER> request is distributed via the ordering stage to all replicas ensuring that running request executions are finished and no new executions are started. More specifically, all requests that have been distributed via the ordering stage before the <RECOVER> request are finished by the fault-free replicas. After this point, all of them are in a consistent state. The faulty replica finishes all already executing requests and aborts request execution in case of detecting a corrupted object. The latter prevents control-flow errors and contains the state corruption. Next, the faulty replica transmits a list of all state-object checksums that have been changed during or after the detection of the state corruption. This explicitly includes requests that were executed concurrently to the fault-detecting or faulty request. Once the fault-free replicas receive the list of checksums, they build their own list and compile a state delta. This state delta consists of state objects with diverging checksums, state objects that are locally changed due to request execution but not at the faulty replica, and state objects that changed at the faulty replica but not locally. The latter group is caused by control-flow errors. The state delta is then transmitted to the faulty replica, which in turn uses the first complete incoming data set to update its local state. All requests that are not finished before recovery need to be discarded by the faulty replica, since the state (delta) update already covers those requests. However, if the faulty replica is responsible for any client connection, it must externalize the reply to the client. Therefore, the state-delta transfer also includes the reply messages.

After finishing recovery, the repaired replica broadcasts a <CONTINUE> message via the ordering stage. Once received by any replica, normal operation can be safely continued.

IV. Implementation

As a case study, we implemented our approach in an actively replicated key-value store based on MEMCACHED++ (Figure 3), a C++ version of *memcached*. In case of MEMCACHED++, all relevant components of memcached are represented by classes that can be individually hardened by applying GOP. These components include a central hash table that manages all key-value pairs, the individual key-value pairs, and a number of management classes.

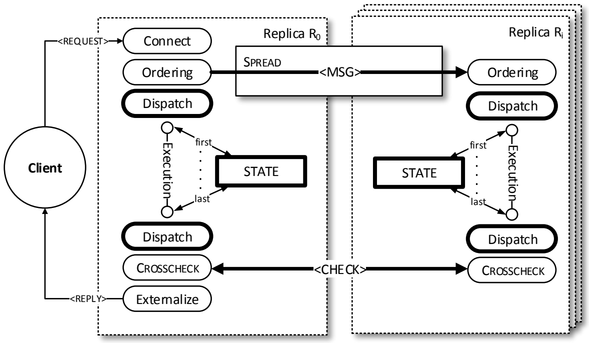


Fig. 3. Prototype implementation

Such a replicated key-value store can be used to provide a highly available source for data exchange (e.g., configuration information) in distributed applications and could be extended to offer coordination support similar to Chubby [1] and Zookeeper [21]. MEMCACHED++ offers an object-oriented design and features the same API and threading model as the original version of memcached.

As described by the system model, we need to enforce an ordered execution of requests. We achieve this by the integration of ordering support into MEMCACHED++: On receipt of a client request, a replica parses the request and broadcasts a client request <MSG, threadID, operation, key, value> to all replicas (including itself) via the *Spread Toolkit* [18]. Spread provides a reliable group communication channel and brings all requests in one defined order. After receiving <MSG>, each replica registers the new request to *Storyboard* [14], which creates a lock-order prediction and ensures deterministic execution. Afterwards, all replicas execute the request in a multi-threaded – but controlled – way.

Before externalizing the reply, we perform the crosscheck by exchanging <CHECK> between the replicas and validating its content (see Section III-B). While waiting for <CHECK> messages to arrive, we continue execution by processing further client requests. Upon arrival of <CHECK>, we validate local results, and in the fault free case, forward them to the respective clients.

V. Preliminary results

As part of our preliminary evaluation, we were interested in the overhead introduced by hardening a multithreaded key-value store via CROSSCHECK. Thereby, we measured the overhead of applying GOP to individual classes, and also when hardening *all* relevant classes.

For our evaluation, we used MEMCACHED++ that is originally based on version 1.4.10 of memcached. For ordering requests, we utilized the most recent version of the Spread toolkit (v. 4.3.0). To simulate clients, we selected the *memslap* benchmark, which is a part of libmemcached.

Class	Description
Assoc	Hashtable
Item	Key-value container
Items	Management of items and statistics
Slab	Container for pre-allocated memory
Slabs	Management of Slab instances

Fig. 4. GOP-hardened classes

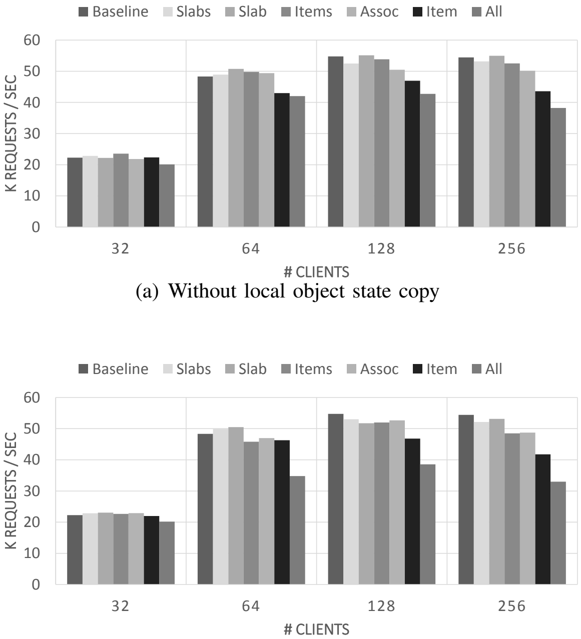
All evaluations were performed on a cluster of four machines with each a Core i7-3770 CPU (4 Cores at 3,4 GHz, supporting 8 parallel threads), equipped with 16 GB RAM and connected over a switched gigabit network. Three machines were used for hosting replicas, whereas the fourth machine was responsible for generating client requests.

We selected a write-intensive workload where each client performs 10.000 *set requests* with a key length of 100 B and a value length of 400 B. To enable concurrent execution, each MEMCACHED++ instance used four worker threads for handling requests. As can be seen in Figure 5, we constantly increased the workload by simulating more clients. The baseline of our measurement builds a plain replicated version of MEMCACHED++ without generic object protection. Furthermore, we individually hardened five classes (see Figure 4) with a CRC32 error detection code. In Figure 5(b), we also evaluated GOP with object state copy, thereby enabling a local recovery from state corruption that are detected ahead of execution.

As can be seen in both figures the overhead for the individual classes varies significantly between 2% and 23%. The reasons are twofold: The overhead greatly depends on the state objects size of protected data as well as on the access pattern for the individual object. For Items and Item both issues apply. The size of these objects ranges from 300 B (Items) to 712 B (Item). Furthermore, for each request one object of both classes is checksummed around 40 to 80 times per request. However, objects of type Assoc and Slab are much smaller and accessed only a few times, hence they introduce only little overhead of 2% to 10% decrease in performance.

For the protection of all five classes the overhead sums up to a performance decrease of 30% for 256 clients. The addition of a local object state copy reduces the performance by another 9%, resulting in a 39% performance reduction.

However, these results can be optimized by further tailoring GOP to fit the demands of CROSSCHECK. So far, each time objects are accessed, at least one checksum is generated. In case of Items and Item this leads to a significant overhead. However, during the crosscheck, we only compare the most recent generated checksums. Hence, instead of generating a checksum, we could simply log (e.g. by using a dirty bit) if an object is accessed. Then at the end of executing a request, but before initiating the actual crosscheck, we generate checksums for all logged objects. This way only one checksum is generated for each object per request.



(a) Without local object state copy

(b) With local object state copy

Fig. 5. Overhead of generic object protection

Unfortunately this approach would reduce our error detection capabilities ahead of execution. To mitigate this each class could be individually protected by the GOP. For example, Assoc and Slab could be protected by CRC32 error detection code plus local copy while Items and Item could be protected with the optimization described above. With these optimizations we expect to decrease the overall performance overhead far below the current level.

VI. RELATED WORK

Utilizing checksums to compare and synchronize replicas in distributed system has been proposed in multiple contexts, however support is usually either limited to persistent state [22], [23] or does not consider state corruptions [24]. CROSSCHECK focuses on hardening and recovering the in-memory state of replicated services.

Correia et. al. [11] proposed an approach for hardening distributed applications against arbitrary state corruptions by means of redundant execution at the granularity of requests inside of a single node. This way, state corruptions can be contained at the node level and masked as crash faults. While being effective for hardening distributed applications, this approach demands a certain application structure to access application state and effectively doubles memory and CPU demand.

The approach of Behrens et. al. [12] can be seen as a refinement of [11]. It explicitly addresses the memory overhead by saving checksums of an initial execution instead of a full state copy. Furthermore, data access is intercepted and checked via checksums at the level of memory pages. Behrens et. al. [25] also propose the use of encoded processing via AN-coding. Amongst other things this offers fine-grained control-flow checks but comes attached with an computational overhead of factor five. In comparison to the aforementioned systems, CROSSCHECK achieves a similar fault-tolerance level, but requires only moderate additional resources due to its tight integration with SMR.

Furthermore, there is only limited work that considers faults beyond crashes, and, at the same instance, allows multithreaded execution. Instead of enforcing determinism, Kapritsos et al. [26] cleverly batches requests to minimize concurrent access to state objects. During parallel execution, if the replicas don't reach a consistent state, a revert with sequential re-execution is performed. This introduces a significant overhead when state object access is shared by many requests as is the case for management objects in our prototype. Same issues apply to Kotla et al. [27] who enables the concurrent execution of requests if they do not change shared state. This essentially leaves the middle ground where services can freely utilize threads but determinism is pro-actively persevered.

VII. CONCLUSIONS

CROSSCHECK builds an approach to tolerate arbitrary state corruptions for implementing highly available multithreaded services for data centers. The use of generic object protection enables CROSSCHECK to harden state objects in a generic and flexible kind of fashion. Our initial evaluation based on key-value store showed an overhead of 2% to 23% for the different protected classes. Our next steps are performance optimization, support for efficient recovery, and performing a fault-injection campaign to assess the achieved error coverage.

REFERENCES

[1] M. Burrows, "The chubby lock service for loosely-coupled dist. systems," in *Proc. of the 7th Symp. on Operating Systems Design and Implementation*, 2006, pp. 335–350.

[2] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li, "Paxos replicated state machines as the basis of a high-performance data store," in *Proc. of the 8th USENIX Conf. on Networked Systems Design and Implementation*, 2011.

[3] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete, "Mdcc: Multi-data center consistency," in *Proc. of the 8th ACM European Conf. on Comp. Systems*, 2013, pp. 113–126.

[4] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: Understanding the nature of dram errors and the implications for system design," *SIGARCH Comp. Architecture News*, vol. 40, no. 1, pp. 111–122, 2012.

[5] V. Sridharan and D. Liberty, "A study of dram failures in the field," in *Proc. of High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, pp. 1–11.

[6] S. Y. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[7] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, 2002.

[8] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for Byzantine fault tolerant services," in *Proc. of the 19th Symp. on Operating Systems Principles*, 2003, pp. 253–267.

[9] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo, "Efficient Byzantine fault tolerance," *IEEE Transactions on Computers*, 2011.

[10] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet, "ZZ and the art of practical BFT execution," in *Proc. of the 6th EuroSys Conf.*, 2011, pp. 123–138.

[11] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini, "Practical hardening of crash-tolerant systems," in *Proc. of the 2012 USENIX Annual Technical Conf.*, vol. 12, 2012.

[12] D. Behrens, C. Fetzer, F. P. Junqueira, and M. Serafini, "Towards transparent hardening of distributed systems," in *Proc. of the 9th Work. on Hot Topics in Dependable Systems*, 2013, pp. 4:1–4:6.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proc. of the Eleventh European Conf. on Object-Oriented Programming*, 1997, pp. 220–242.

[14] R. Kapitza, M. Schunter, C. Cachin, K. Stengel, and T. Distler, "Storyboard: optimistic deterministic multithreading," in *Proc. of the 6th Int. Work. on Hot Topics in System Dependability*, 2010.

[15] O. Spinczyk and D. Lohmann, "The design and implementation of AspectC++," *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software*, vol. 20, no. 7, pp. 636–651, 2007.

[16] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comp. Survey*, vol. 22, no. 4, pp. 299–319, 1990.

[17] L. Lamport, "The part-time parliament," *ACM Transaction Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1998.

[18] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton, "The spread toolkit: Architecture and performance," *Johns Hopkins University, Center for Networking and Dist. Systems (CNDS) Technical report CNDS-2004-1*, 2004.

[19] M. Hoffmann, C. Borchert, C. Dietrich, H. Schirmeier, R. Kapitza, O. Spinczyk, and D. Lohmann, "Effectiveness of Fault Detection Mechanisms in Static and Dynamic Operating System Designs," in *Proc. of the 17th IEEE Int. Symp. on Object-Oriented Real-Time Dist. Comp.*, 2014.

[20] C. Borchert, H. Schirmeier, and O. Spinczyk, "Generative software-based memory error detection and correction for operating system data structures," in *Proc. of the 43rd Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks*, 2013, pp. 1–12.

[21] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for Internet-scale systems," in *Proc. of the 2010 USENIX Annual Technical Conf.*, 2010, pp. 145–158.

[22] A. Tridgell and P. Mackerras, "The rsync algorithm," 1996.

[23] M. Castro, R. Rodrigues, and B. Liskov, "Base: Using abstraction to improve fault tolerance," *ACM Transaction Computer Systems*, vol. 21, no. 3, pp. 236–269, Aug. 2003.

[24] R. Kapitza, T. Zeman, F. J. Hauck, and H. P. Reiser, "Parallel State Transfer in Object Replication Systems," in *Distributed Applications and Interoperable Systems*, vol. 4531, 2007, pp. 167–180.

[25] D. Behrens, S. Weigert, and C. Fetzer, "Automatically tolerating arbitrary faults in non-malicious settings," in *Proc. of 6th Latin-American Symp. on Dependable Comp.*, 2013, pp. 114–123.

[26] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, "All about eve: Execute-verify replication for multi-core servers," in *Proc. of the 10th USENIX Conf. on Operating Systems Design and Implementation*, 2012, pp. 237–250.

[27] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *Proc. of the 2004 Int. Conf. on Dependable Systems and Networks*, 2004, pp. 575–.