CrossCheck: A Holistic Approach for Tolerating Crash-Faults and Arbitrary Failures

Arthur Martens[1], Christoph Borchert[2], Manuel Nieke[1], Olaf Spinczyk[2] and Rüdiger Kapitza[1]
[1]TU Braunschweig [2]TU Dortmund

***Abstract*—High availability is no longer optional since more and more Internet-based services provide economical or otherwise critical offerings. Traditionally, crash faults are addressed using state-machine replication (SMR) and critical data is selectively protected by checksums. Both techniques can be efficiently combined, however, large parts of a service remain susceptible to transient errors such as bit-flips or more severe state corruptions.**

To address this weakness and also to reduce the labouring and non-trivial effort of identifying and selectively hardening a complex service, we propose *CrossCheck* – a holistic approach. *CrossCheck* extends the crash-fault protection of SMR to also provide tolerance against arbitrary state corruptions, thereby especially addressing multithreaded applications. This is achieved by a fine-grained state comparison and a precise recovery mechanism using fault-free replicas. The implementation utilizes aspect-oriented programming and therefore requires only minimal manual changes to the underlying software. In our evaluation, we show that a multithreaded key-value store can be made resilient to crashes and hardened against arbitrary state corruptions with moderate overhead.

I. Introduction

High availability is a key concern of todays distributed systems, because loss of data and system outages have substantial economical impact and in some cases can even endanger lives. A recent study estimates that for an average outage the costs are about $8,000 per minute [1]. To prevent service outages, replication approaches represent current best practice. Although many different replication techniques exist, they usually target to tolerate only crash faults.

However, due to shrinking structure sizes of commodity hardware and the economical pressure to reduce the energy consumption of modern servers their reliability decreases. In particular, we see an increasing number of transient errors, which cause bit flips in main memory [2], [3], [4] or inside the CPU. For server hardware ECC-guarded memory is used, however this only allows to tolerate single bit flips. Chipkill-enhanced main memory [5], which increases resilience against memory corruptions, has not entered the mass data center market, presumably due to economical reasons. As a consequence, distributed systems providing critical services hosted by data center infrastructures, such as Clouds, are susceptible to state corruptions even under replicated deployment. As a prominent example the Amazon S3 outage[1] can be named, where a few corrupted messages caused an outage of large parts of the service despite massive replication.

From a consumer point of view it is not possible to change the hardware of a cloud infrastructure and additional hardware mechanisms might be too expensive for a limited number of critical services. Hence, software-based measures represent a viable option to tolerate crashes and address silent data corruptions. Typically, arbitrary faulty behavior in the context of replicated distributed systems is addressed by means of Byzantine fault tolerance (BFT) [6]. However, BFT goes beyond arbitrary state corruption (ASC) and addresses coordinated malicious actions. Therefore it is not surprising that it demands significant more resources compared to plain crash-tolerant replication schemes. Since malicious faults such as intrusions are not assumed in most scenarios, recent research specifically targets to tolerate ASC caused by transient errors [7], [8], [9]. A solution presented by Correia et al. [7] is based on local redundant execution of requests and a subsequent comparison of the change in the application state. Another way to deal with ASC is to employ software encoded processing as presented by Behrens et al. [9]. Both solutions have in common that they solve state corruptions exclusively and do not consider crashes. However crashes are still the most common type of failure in distributed systems. In order to build a holistic solution these approaches need to be combined with replication across different machines. Thereby the combined overhead would be prohibitive to be used in practice.

In this paper we present *CrossCheck*, a holistic approach to tolerate arbitrary state corruptions and crashes. Our approach aims at request-oriented distributed systems and builds on top of state-machine replication (SMR), an established solution for implementing highly available services. Prominent examples are coordination services [10], highly available data storage [11], but also wide-area replication of databases [12]. As an extension to SMR, we apply a fine-grained state change tracking by generating hash values from small entities of the system state, which are accessed in the cause of executing a request. For each request, we aggreate all generated hash values and exchange them between the replicas immediately before a reply gets externalized to the client. Eventually these values are compared by a majority voting and in case of a mismatch a recovery mechanism is initiated. Since we collect one hash value for each accessed entity, we are able to identify precisely the

Accepted for publication in:
12th European Dependable Computing Conference (EDCC 2016)
http://www.edcc2016.eu/

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

corrupted entities. This enables a tailored recovery where only data of the corrupted entities needs to be transmitted from fault free replicas. In conclusion a recovery can be processed in a few milliseconds and does not significantly impact the overall performance of the system. By enforcement of weak determinism, *CrossCheck* is also capable of hardening multi-threaded applications, which is a key technology for systems with high throughput. We achieve this with our story-board framework [13] which utilizes the concept of schedule memorization [14]. For evaluation, we applied *CrossCheck* to *Memcached++*, an object-oriented version of the well known key-value store *Memcached* with an extension for SMR. Our results show that we can tolerate crash-faults and ASC in a SMR-system at a performance overhead below 20%.

In summary, we provide three contributions:

- We present a novel concept called *CrossCheck* that exploits already available redundancy and determinism in SMR to achieve resilience against ASC at low overhead. *CrossCheck* also includes a recovery mechanism that can repair corrupted state at object granularity. Furthermore we identify challenges that are introduced by multi-threaded services and provide sound solutions.
- *CrossCheck* requires tailored extensions for any class that needs to be protected. We show how these extensions can be implemented with automatic code transformation so no manual changes to protected classes are required. In particular we present generic solutions for tracking state changes, validation and recovery that we have implemented in our *CrossCheck* library framework.
- We have evaluated our concept based on *Memcached++*, a multi-threaded prototype service that we hardened with *CrossCheck*. We present the results for throughput, latency and recovery times in this paper.

We first present related approaches on arbitrary faults in Section II and our system model in Section III. Next, we explain the detailed concept of *CrossCheck* in Section IV. In Section V, we introduce aspect-oriented programming a core technology, which we rely on for our implementation presented in Section VI. The evaluation of our implementation is shown in Section VII, while Section VIII concludes the paper.

II. Related Work

In the high performance computing (HPC) domain, tolerance against hard and soft errors in DRAM is usually provided through special error correction code (ECC) protected hardware. Among various ECC solutions, single-bit error correction, double-bit error detection (SEC-DED) and Chipkill ECC are the most common ones. Compared to SEC-DED, Chipkill reduces the uncorrected DRAM errors by a factor of 42x [3]. Judging from the results of recent large-scale field studies [4], [2], [15], only Chipkill ECC offers sufficient error protection. However, commodity data center hardware, due to cost pressure, utilizes at best DRAM with SEC-DED ECC, leaving many errors uncorrected. Although very seldom, even with Chipkill ECC uncorrectable errors still occur. Therefore, we argue that additional software protection is needed to secure critical services in order to address arbitrary state corruptions.

Tolerance against arbitrary faulty behavior and crashes is traditionally provided by BFT protocols [6], [16], [17], [18]. However, BFT is rather resource demanding as $3f+1$ nodes are required to tolerate f Byzantine faults. Therefore, a recent trend targets to address arbitrary state corruptions which have their origin in hardware malfunctioning. Correia et al. [7] formulated the ASC error model. In this work, tolerance against ASC is provided through the PASC library, which duplicates the application state and executes all state modifications redundantly. PASC achieves this transparently, if a strict programming model is followed. All application state has to be placed in one object and all modifications need to be carried out by event handlers. A follow-up work by Behrens et al. [8] generalized the aforementioned work. They removed the notion of a specific programming model and addressed the memory overhead by storing checksums of modified pages during the initial execution instead of managing a full state copy. Software encoded processing [19], [9] is another way of dealing with unreliable hardware. Critical data is encoded in a special way, and all processing is conducted on encoded data. Data and control flow faults result in wrong codewords and are thus detected. All these ASC-tolerance approaches share a significant computational performance overhead ranging from 2x (redundant execution) up to 5x (encoded processing). In a real world scenario where crash fault tolerance is required in the first place, this would be needed on every replica, therefore the overall overhead would be multiplied. Moreover, none of these approaches covers multi-threaded applications.

Since concurrent operation on shared data makes reasoning about the application state challenging, there is only limited work that considers faults beyond crashes, and, at the same instance, allows multithreaded execution. Kapritsos et al. [20] proactively allow concurrent execution as long as the application remains consistent. In the case of inconsistencies, a revert to a save state followed by sequential re-execution is performed. This assumes efficient support for micro checkpoints and a transactional application behavior. Furthermore, application workloads with high contention and large shared state will suffer from a high rate of re-executions. Kotla et al. [17] 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 preserved.

This paper supersedes our preliminary workshop publication [21]. The paper at hand describes the fully implemented *CrossCheck* library, refined concepts, and an extended recovery procedure. Based on our new prototype implementation, we present a detailed evaluation of *CrossCheck*.

III. System Model and Assumptions

With *CrossCheck* we aim at hardening typical distributed systems composed as a client-server architecture and deployed in a Cloud.

A. Service structure

Multiple clients may connect and issue requests via messages to one server, which in turn processes them. During the execution of a request, commands may perform any kind of access (i.e., read, write, create and delete) to the internal in-memory state. Afterwards, the server responds to the clients with a reply message. To address multi-core hardware and recent service implementations, we assume a multi-threaded service design. In this model multiple requests can be executed concurrently by different threads which may cooperate via shared data. Access to all critical sections needs to be race-free, which is achieved by atomic locks (i.e., mutex locks). Lock-free solutions are excluded. Without further measures, the execution order of multiple requests is non-deterministic, as the order depends on the internal scheduling policy of the operating system that is influenced by the workload at the time of execution. We expect the server application to be written in an object-oriented way and to be well structured. In conclusion, the in-memory state must be encapsulated in any number of *state-objects*. However, only a subset of the state-objects are considered critical for the service to operate correctly. Typically these *critical state-objects* store mission-critical data or messages which need to be externalized. According to the best practice in object-oriented code design, critical data should not be stored in public member variables.

B. Fault model

Any error may stop the system (i.e., an infinite loop), lead to a crash or may corrupt data. In critical state-objects any data corruption needs to be detected and repaired before data is externalized to the client. For any other object it is sufficient to ensure that all possible data corruptions that propagate to a critical state-object are handled there. Information exchanged via message passing may fail completely, corrupt or delay messages. Also, messages may arrive out of order. In analogy to [7], errors have an arbitrary character. They may consist of single and multiple bit flips and may even alter the control flow. We also make no assumptions on the number of state-objects that can be corrupted at the same time. However, we expect the behavior to be random and not coordinated malicious. In the remainder of the paper, we will refer to these faults as arbitrary state corruption (ASC). At most f out $2f+1$ replicas can be faulty at the same time. Of course the entire system (including the operating system and libraries) may be affected by ASC. As long as ASC in these parts do not propagate to the critical state-objects, protection of these parts is not needed for correct behavior. Furthermore, we do not consider clients to be faulty and assume that the server software itself does not include bugs which cause arbitrary faults.

C. Architectural measures

In order to tolerate crashes or a not responding service we expect that *SMR* [22] is applied. Thereby, liveness is ensured while tolerating f crash faults with $2f+1$ replicated instances. However, this requires that each replica receives the same ordered set of requests and behaves like a deterministic state machine: To implement the requirements of SMR, we need to enforce total ordering of messages and enforce deterministic execution in each replica. Total message order is achieved by an agreement protocol (e.g., Paxos [23], Raft [24]) or a group communication framework (e.g., Spread [25]) supporting total ordering of messages. To ensure determinism a deterministic threading library like Dthreads [26] or a framework for determinism (Tern [27], Coredet [28], Kendo [29]) can be applied. In the case of *CrossCheck* we used *Storyboard* [13] that follows the idea of scheduling memorization similar to Tern.

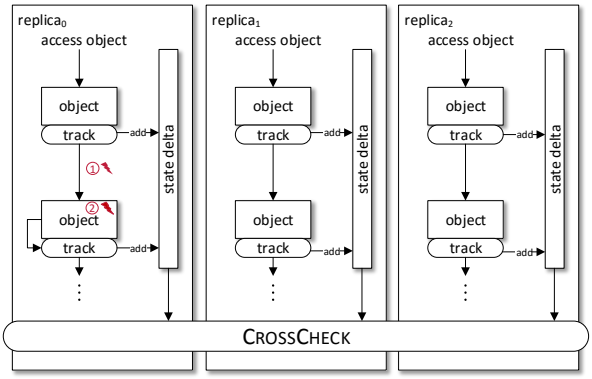


Fig. 1. The *CrossCheck* approach

To ensure that messages do not contain corrupted data, any information which leaves the protected application should be protected by a checksum, i.e., a 32-bit cyclic redundancy check (CRC32). Any other failures like missing or reordered messages need to be handled by the communication protocol (TCP/IP) or by a group communication layer. Protection of these software layers is beyond the scope of this paper.

Tolerating ASC in critical state-objects before data gets externalized to the client is the task of *CrossCheck*. *CrossCheck* further provides means to recover the state of corrupted objects via fault-free replicas. This way an arbitrary number of ASC can be tolerated over time.

IV. The CrossCheck Approach

The *CrossCheck* approach can be subdivided into three steps: First, during the processing of a request every change of the application state is tracked and aggregated as a *state delta*. This builds the basis for the second step: before a reply gets externalized to the client, the state delta is exchanged with other replicas and validated for correctness via majority voting. In case an error is detected, i.e., state deltas mismatch, a recovery process is initialized as a third step. This involves a synchronization of all replicas and a transmission of a recovery packet to the faulty replica. Using the recovery packet the faulty replica repairs its corrupted state. Normal execution continues after the recovery is finished. The following subsections explain each step in detail.

A. Tracking State Changes

Problem description: Tacking state changes in a concurrent environment: Tracking state changes can be done via regularly building checksums over parts of the in-memory service state, i.e., every time a reply is externalized to a client. However, state of the art services may possess an in-memory state of up to several GB and can easily serve thousands of requests per second. Under such conditions, tracking the entire state involves a substantial performance impact.

Fortunately, requests usually access only a very small portion of the service state, but tracking these state parts is nevertheless challenging. Naturally the state of an application is subdivided into pages at the machine level. A common way to track page access, i.e., during request processing, is to set all pages read-only via the `mprotect(2)` system call [8]. On a write access a page-fault is generated which can be caught by a special page-fault handler. The page-fault handler tracks the access and immediately grants write access to the affected page. At the end of the execution period all tracked pages are switched back to read-only access. The main advantage of this approach is its generality and its transparency to the application. However changing page access-rights implies flushing the respective TLB entry [30], which in turn impacts performance. More problematic, it is hard to apply this approach when requests are served by multiple threads because it is complicated to decide which changes belong to which request and thread. Furthermore, if not addressed properly, scheduler non-determinism might cause different checksums across replicas.

Thus, instead of tracking the access of pages, *CrossCheck* tracks the access of coherent blocks of data. This partitioning is usually provided by an object-oriented code design, which is widely used in distributed systems. A block of data in this case is simply an object. As mentioned in our system model we assume all data inside critical state-objects is only accessible via method calls. According to this assumption, accessing and leaving an object is simply represented by calling and returning from methods.

Separation of critical from non-critical data: While protecting all data is in principle possible, this comes at a price and is in many situations not necessary. Objects usually offer an insight into the application semantics which can be utilized to further divide the state into critical and non-critical state-objects. For example statistical and logging informations do not affect the main task and temporary data structures can also be left unprotected. Usually connection management and intermediate buffers fall into this category. E.g., the loss of a connection is a common event in a distributed system that is handled by existing mechanisms. Errors in intermediate buffers are also not critical because the corrupted data will eventually propagate to critical state-objects and be handled there. As a rule of thumb all state objects that do not influence the output behavior of a service are not critical and can be ignored when hardening a service against state corruptions. An additional way to find critical state-objects is by conducting fault injection experiments, e.g., by using FAIL* [31].

Tracking state changes and calculating checksums: To track object modifications, we need to calculate a checksum from object members whenever an object is accessed at runtime. This approach is depicted in Figure 1. Over the course of a request-processing, every checksum generated for an object is captured inside a state delta. For each accessed object, the state delta contains a pair of the latest checksum and a unique *object id*. To be able to compare the same object across replica boundaries it is important that the object ids are deterministically assigned and equal across all replicas.

Tracking the state of critical objects by checksum computation of its members can be challenging depending on the types of the member variables. While checksumming primitive data types like char, int or long can be implemented straight forward, references and members of a class type require special treatment. As all critical-objects have their own checksum, critical state-objects nested in other critical state-objects should not contribute to a checksum to avoid overhead. Non-critical-objects nested in critical objects should also not contribute to the checksum because this data is not critical by definition. References either need to be deterministic across replica boundaries for comparison or should be avoided for checksum creation. The former solution can be achieved with deactivation of Address Space Layout Randomization (ASLR), when memory allocation is deterministic. This comes at the cost of reduced security and performance. In the latter case references need to be protected with another technique like [32], [33]. Implementing a tracking solution for every object access cannot be solved without access to the source code and code instrumentation. Fortunately, this can be done automatically and transparently to the source code via aspect-oriented programming as explained in Section V.

Over the course of a request-processing only the latest checksums for each accessed object are stored inside the state delta. Old checksums are not relevant because even if they have a hint for a transient error this may be resolved by application semantics. Of course discarded checksums imply an overhead which seems unnecessary at first. In order to prevent unnecessary checksumming a solution would be to track object accesses and compute the checksums once, right before broadcasting the checksums to other replicas. However, under concurrent request processing and without any additional means the timing of checksum computation is not deterministic with respect to object accesses from other threads. To overcome this issue every access to a critical object needs to become deterministic, including checksum generation. With weak determinism this could be achieved when a lock is acquired for the entire object whenever it is accessed. This dramatically increases the synchronization overhead because even data which is not shared between threads but placed in one object needs to be synchronized. Depending on the scenario this impacts the performance for concurrent request processing much more than redundant checksum computations.

Optimizing for concurrency: Even though checksums are computed right after an object is accessed, concurrent access to objects can still occur. For example two threads may access

different member variables of the same object concurrently. As a performance optimization, tracking capabilities can be relaxed. Whenever an object is accessed concurrently only the last thread leaving the object computes the checksum and thus tracks the state changes. This behavior can be realized with a wait-free algorithm [34] and involves minimal overhead. With this optimization not every modification is tracked immediately, meaning a subsequent comparison with other replicas is not always possible. Nonetheless, this solution is feasible under two conditions: First, we need to ensure that every externalized message is validated. Second, we need to guarantee that every object modification is tracked eventually. The first requirement ensures that even when an error in a critical state-object occurs which was not tracked and validated, no corrupted reply is externalized to the client or any other application. The second requirement ensures that no critical state-object stays untracked for an infinite period of time. This can be easily achieved by blocking the access to the object until its state change is tracked whenever too much accesses where not tracked consecutively. Further details on the implementation can be found in Sections V and VI.

B. CrossCheck state validation

Figure 1 gives an overview of the data flow during request-processing. Critical state-objects may be affected directly ② either when they are stored in the main-memory *(passive-state)* or during the access *(active-state)*. Furthermore, errors may occur somewhere else in the remaining service state ① and might propagate to the critical state-objects. In any case, every change of a critical state-object is tracked after the object is accessed.

When the processing of a request is finished, we start the *CrossCheck* state validation procedure directly before a reply is externalized to the client. At this point the **state delta** D contains one **set of pairs** $P = \{p_1, ...p_n, p_{reply}\}$ with the latest checksums for every critical state object accessed during request-processing. Each **pair** p comprises a **checksum** C and its corresponding **object id** I. Additionally P contains p_{reply}, a special pair for the reply message. For a setup with three replicas, the state validation follows the algorithm depicted in Fig. 2. Initially the own $P \in D$ is broadcasted to all replicas in form of a <CHECK> message as shown in TABLE II. For each replica and and each request a status is maintained that represents the knowledge about the state of the replica. An explanation for this status is given in TABLE I.

As long as the own replica status is UNKNOWN or SUSPECT the algorithm waits for new <CHECK> messages from other replicas (Line 3). On arrival of a <CHECK> message all pairs from the message are compared with the pairs stored in D (Line 5). Depending on the result of the comparison the replica states are updated (Lines 6 to 18). In case all pairs from <CHECK> match with the pairs stored in D for a majority of replicas, the successfully compared replicas can be set directly into the VALID state. For a setup with three replicas this can be reached after comparing the pairs of two replicas. If the first comparison fails, the state of the involved replicas changes from

```
Data: state delta D containing for each replica R_i a set
      of pairs P_i with pairs p=(checksum C, object id I);
1  broadcast own <CHECK> message;
2  while own status is UNKNOWN or SUSPECT do
3      wait until one <CHECK> message arrives;
4      foreach P_i ∈ D and P_i ∉ ∅ do
5          compare all P_j ∈ <CHECK> of replica R_j with P_i;
6          if comparison succesful then
7              foreach replica R ∈ {R_i, R_j} do
8                  set state of R to VALID;
9              end
10         else
11             foreach replica R ∈ {R_i, R_j} with state
                UNKNOWN do
12                 set state of R to SUSPECT;
13             end
14             foreach replica R ∈ {R_i, R_j} with state
                SUSPECT do
15                 set state of R to FAILED;
16                 initiate recovery for R;
17             end
18         end
19     end
20     add all P_j ∈<CHECK> to D;
21 end
22 if replica needs to reply then
23     if local p_reply is corrupt then
24         wait until recovery is finished;
25     end
26     externalize <REPLY> to client;
27 end
```

Fig. 2. *CrossCheck* state validation algorithm for three replicas

UNKNOWN to SUSPECT. At this point we know that an error exists but not which replica is affected. Another comparison is needed to identify the corrupt replica. In that case an already SUSPECT replica transits into the FAILED state when it is involved again in a failed comparison. A replica will also change into the FAILED state when it is SUSPECT and not part of a successful second comparison. Therefore two comparisons and pairs of all three replicas are needed to reach the FAILED state. When a FAILED replica is identified a recovery procedure is started (Line 16). Eventually for every request the state of the own replica is set either to VALID or FAILED. Finally, normal execution is continued even in case of an error. If the replica needs to externalize a reply its correctness must be ensured (Lines 22 to 27). Otherwise the reply must be hold back until it is repaired during recovery (Line 24).

A reply is send to the client as soon as a quorum of replicas reaches the VALID state. At that point not every replica state is known. However, for recovery we need to obtain the knowledge about the state of each replica. For that we need to continue validation until all replicas are validated. This can be offloaded to a separated, deferred validation step.

TABLE I
REPLICA STATUS FOR EACH REQUEST

Name	Meanining
UNKNOWN	Replica not verified
SUSPECT	Replica may have errors
VALID	Replica has no errors
FAILED	Replica has error

TABLE II
STRUCTURE OF A <CHECK> MESSAGE

CHECK header	p_0	p_1	...	p_n	p_{reply}

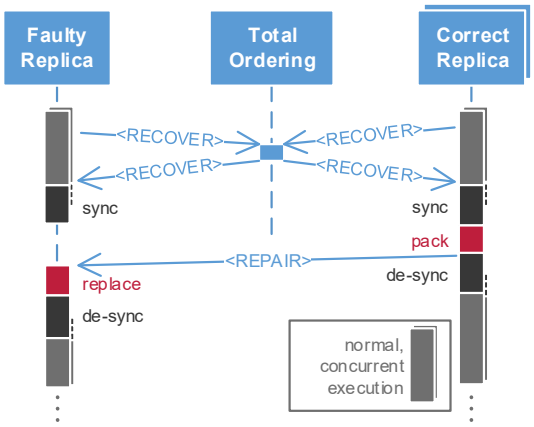


Fig. 3. Recovery procedure

C. Recovering from faults

In case of an error detected by any replica, *CrossCheck* invokes the recovery procedure through broadcasting a <RECOVER> message as shown in Fig. 3. On arrival of the first <RECOVER> message the whole recovery process is carried out in two steps: synchronization and repair. Because all replicas operate on their own speed, we first need to synchronize all replicas to bring them into an equal and quiescent state. Therefore, it is crucial to integrate the <RECOVER> message into the total order of all requests. All requests that arrive prior to <RECOVER> are executed in a normal way while any request that arrives afterwards is put on hold until the recovery procedure is finished.

Even though we risk error propagation inside the faulty replica, normal operation should be continued until synchronization is finished. However, by cross-checking every request, we ensure that every critical state object and each reply is validated before a request is finished. Any corrupted object we detect in that period is added to a set which will be repaired during the next step of recovery. This approach has performance advantages when errors do not propagate to subsequent requests. It also does not prolong the synchronization procedure because the fault free replicas need to continue cross-checking messages anyway. In contrast, stopping the faulty replica immediately would require additional state comparisons after synchronization and additional state updates would need to be transmitted.

When synchronization is initialized all threads processing requests need to stop in a deterministic way. Additionally it is necessary that every replica has the same global knowledge about the state of every other replica. To achieve this, every request that was executed on the replica needs to be compared regarding its state changes. Therefore each replica needs to wait until all <CHECK> messages have arrived. Of course a corrupted replica may crash or run into a deadlock during synchronization before it can transmit every <CHECK> message. Thus every synchronization needs to get finished in a given time or the corrupted replica must be considered unrecoverable and replaced by a new one.

After synchronization is finished each replica has the same global knowledge about the faulty replica and its corrupted objects. In order to repair the corrupted objects, each fault-free replica needs to pack and transmit the corresponding fault-free objects as a <REPAIR> message to the faulty replica. The structure of a <REPAIR> message is shown TABLE III. Each packed data set K_i inside <REPAIR> is identified by the unique object id I_i which is equal across all replicas. The faulty replica in turn accepts the first incoming <REPAIR> message and overwrites the corrupted objects with the packed data.

This procedure requires a special fine-grained packing technique, since we can not treat objects as simple data blobs or utilize existing serialization procedures. References inside objects for instance are not invariant between replica boundaries and nested objects are tracked and transmitted on their own. Our solution requires specialized *pack* and *replace* methods for each critical class which covers only the tracked parts of the objects. This way pack generates a coherent data set K_i consisting of all tracked object parts for object i. Calling the replace method overwrites all tracked parts inside the object i with the contents of K_i. Although this approach requires a tailored solution for each protected class, it can be implemented in a generic way in form of an aspect as explained in Section VI.

To continue normal execution no further communication is needed. Immediately after a fault-free replica finishes transmission of the <REPAIR> message it can carry on with normal request processing. The faulty replica has to finish replacement for all corrupted objects, but afterwards it may also continue serving requests.

Control-flow faults like deadlocks or infinite loops, which prevent a thread to ever finish its task, require a special treatment. We recommend to pair *CrossCheck* with a traditional watchdog approach in order to ensure each thread is running. Additionally errors may manifest outside the protected critical objects. By propagating to the critical state frequent repairs that will highly impact the performance may follow. Therefore, when a replica encounters too many recoveries in a given time period or when the watchdog is triggered, the affected replica should be replaced.

D. Dealing with false positives during error detection

Weak determinism does not ensure that every object access happens in the same order on every replica, because we do not

TABLE III
STRUCTURE OF A <REPAIR> MESSAGE

REPAIR header	I_0	K_0	I_1	K_1	...	I_n	K_n

```
1  class OperationsCounter {
2    int writes;
3    int reads;
4    void incrementReads();
5    void incrementWrites();
6  };
7  OperationCounter opCnt;
8
9  void writerThread() {
10   opCnt.incrementWrites();
11   [...] //do stuff
12 }
13
14 void readerThread() {
15   opCnt.incrementReads();
16   [...] //do stuff
17 }
```

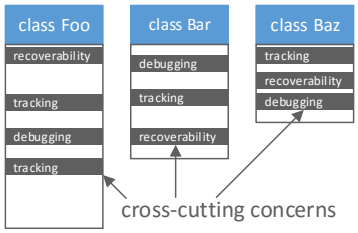
Fig. 4. Example of an asynchronous object access that may lead to false positives during error detection

acquire a lock whenever an object is accessed. Although not very common, situations may occur where several threads modify different members of the same object in different order without using a lock. An example for such a situation is depicted in Fig. 4. Here two threads, the *writerThread* and the *readerThread*, access the object *opCnt*. Because both threads access different member variables no synchronization is needed and therefore no determinism is enforced. Therefore no guarantees can be given in which order or at which point in time the threads modify the shared object and divergent checksums may turn up across replicas. In a simple case only F out of $2F+1$ replicas exhibit a false positive at the same time. Thus an unnecessary recovery will be initiated that affects only performance. However, the checksums may also be different at every replica. This would break our assumptions about faults. In this situation it is necessary to initiate a synchronization followed by a re-computation and exchange of all checksums that were divergent across replicas before. If a false positive error detection was the cause, all exchanged checksums must be equal after synchronization and normal operation can continue.

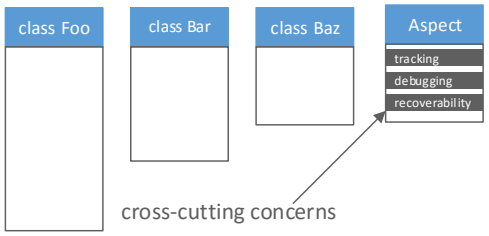
As shown in our evaluation Section VII-B, recovery is fast in *CrossCheck* and does not affect the performance significantly. Furthermore, false positives occurred only seldom in our example application. Nevertheless, if the false positive rate would be high for a given use case, a lock could be added to the affected objects thereby eliminating the race condition.

V. GENERIC OBJECT PROTECTION

We use aspect-oriented programming (AOP) [35] and generic object protection (GOP) [34] as base technologies for *CrossCheck*. This section gives a brief introduction to both technologies.



(a) Traditional implementation



(b) Implementation using AOP

Fig. 5. Implementation of cross-cutting concerns

```
1 criticalObj.check(); //introduced by Aspect
2 criticalObj.accessMembers(); //original call
3 criticalObj.update(); //introduced by Aspect
```

Fig. 6. New methods are woven around every original method call

A. Aspect-Oriented Programming (AOP)

AOP is a programming paradigm that focuses on the separation of *cross-cutting concerns* at the implementation level. A cross-cutting concern is a particular feature that is usually spread across the source code as shown in Fig. 5a. In the presented example, three cross-cutting concerns of *CrossCheck* are shown: tracking, recoverability and debugging. A traditional implementation of these concerns would be spread over many classes. With AOP, cross-cutting concerns can be implemented by an aspect as depicted in Fig. 5b.

We used *AspectC++* [36], an AOP extension for C++, in the course of implementing *CrossCheck*. AspectC++ allows code transformations (weaving) in classes and functions at any point that can be described by match expressions. Thereby, classes can be extended by new members, methods, or parent classes. In addition, methods and functions can be extended by new code. Furthermore AspectC++ provides a compile-time introspection API that is comparable to the Java Reflection API available at runtime. AspectC++ offers information on the class hierarchy and the call graph. Together with C++ templates this provides a very powerful tool to build generic solutions.

That way, our solution for tracking state changes and recovery can be applied transparently to any class.

B. Generic Object Protection

GOP is an AOP-based dependability mechanism that we exploit for tracking state changes. The main goal of GOP is to protect critical state-objects from transient faults during the passive state. Applying GOP only requires the developer to specify a set of classes by a match expression, and the resulting code transformation is automatically carried out by the AspectC++ compiler.

For example, all classes considered as critical can be extended by a checksum. Integrity verification is carried out whenever a critical object (that means an object of a critical class) is accessed through a call of its methods. As shown in Fig. 6, new `check()` and `update()` methods are woven around every original method call. `check()` validates the *critical object* by comparing the introduced checksum with the object's real data. If both diverge an error has been detected and further measures can be applied. In case no error is detected, the original method call is performed. Afterwards `update()` is invoked. It recomputes the critical object's checksum and stores it inside the object itself.

GOP is highly configurable and error detecting and also correcting variants are available. Redundancy can be applied in various forms of checksums as well as full-fledged object copies. For *CrossCheck* we use a CRC32-based error-detecting variant that uses hardware instructions provided by `SSE 4.2`. According to Koopman et. al. [37] CRC32 achieves for reasonable data sizes a hamming distance of at least 4. For the simplicity we will keep using the term "checksum" for referring to the CRC32 code, even though it is not mathematically accurate. Recovery capabilities are not required as they are provided by *CrossCheck*.

GOP performance optimizations: Several optimizations are applied to improve performance. Typically GOP requires two checksum computations for each access of a critical object, one during `check()` and one during `update()`. However read-only methods (in C++ these are qualified as *const*) do not modify the object, thus the `update()` can be omitted for performance reasons.

Furthermore short running methods like getters and setters typically appear in sequences of calls. In these cases the critical object stays only very short in the passive state while frequent checksum computations involve high computational overhead. An optimal solution would invoke `check()` at the beginning of such call sequence and `update()` when the last short method was called. The AspectC++ provides a *project model* that contains information from a static control- and data-flow analysis during compilation. This project model can used to automatically optimize out unneeded `check()` and `update()` operations. Further details on this topic are given in [38].

As mentioned in Section IV-A, computing the checksums can be implemented with a wait-free synchronization algorithm without the need for locks. Every critical object is extended with an atomic counter that is incremented for every thread concurrently accessing the object. Whenever `check()` or `update()` are called, the counter has to be examined. If it is not zero, another thread is accessing the same object concurrently, thus, `check()` should be omitted. Likewise, `update()` should be left out whenever the counter is not set to one. Additionally an atomic *dirty flag* is needed to mark an ongoing checksum computation. Altogether this algorithm ensures that `check()` is called by the first thread entering the object and `update()` is called by the last thread leaving the object. Further details on the algorithm, including a formal model and verification are explained in [34].

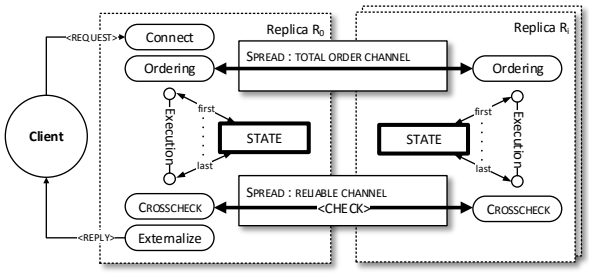


Fig. 7. Prototype implementation

VI. Implementation

A. Example service: Memcached++

We implemented *CrossCheck* as a framework based on a small library that is augmented by aspects. As a case study for *CrossCheck* we used *Memcached++* (Figure 7), a C++ version of the well-known key-value store *Memcached*. Since all relevant components of *Memcached++* are represented by classes we can apply the whole spectrum of AOP as provided by AspectC++. *Memcached++* features the same API and threading model as the original version (v 1.4.10) of *Memcached*. Additionally, *Memcached++* supports active replication and deterministic execution. Both features are required for *CrossCheck*.

Communication between the *Memcached++* replicas is carried out via the *Spread Toolkit* [25] (v 4.04). Spread provides group communication channels with configurable guarantees. In each replica every request is broadcasted over a totally ordered channel to bring all incoming requests in total order. Only requests that arrive over this channel are processed subsequently. During the execution of requests weak determinism is enforced by our *Storyboard* [13] framework.

B. State tracking

State changes are tracked during execution by GOP that is configured with a CRC32-based error detection. We extended the GOP implementation so that checksums are gathered in a set of checksums for every request separately. This set represents the state delta as detailed in Fig. 1. Additionally wildcard checksums are generated whenever multiple threads access the same object concurrently. With this approach we protect the individual key-value pairs, the central hash table and a number of other management classes. The state machine, which controls the execution, and the client connection handler on the other hand were the most prominent parts which stayed unprotected. We have

chosen this partitioning because any error will either stop the application or propagate to at least one of the protected objects.

C. Cross-checking state changes

State validation by *CrossCheck* is conducted in the last state of execution, before reply externalization. As explained in Section IV-B, the *CrossCheck* algorithm of Fig. 2 is executed at this point in time and <CHECK> messages are broadcasted. The <CHECK> messages do not need to be totally ordered and are exchanged over a reliable channel. To improve performance we send out <CHECK> in batches of 5 messages with a dedicated thread. Since waiting for other <CHECK> messages is very time consuming, *CrossCheck* stores its state and immediately returns. This allows to continue execution by processing other client requests. When <CHECK> messages eventually arrive, we return to corresponding request execution, reinvoke *CrossCheck* and continue with state validation.

D. Recovering from faults

For recovery, the *CrossCheck* library provides an API that supports synchronization of threads, in addition to aspects to make critical classes *recoverable*. Since many different threading models are exhibited by distributed applications only very basic operations like stopping a thread or notifications are provided by the *CrossCheck* library. For the synchronization of the worker threads in *Memcached++* we create specific requests that are dispatched deterministically to every worker on arrival of a <RECOVER> message. These requests use the API calls of *CrossCheck* to stop the workers until recovery is finished.

The recovery itself is implemented in an aspect-oriented way. No manual changes need to be applied to the critical classes. Fig. 8 shows a class diagram which explains the extensions we weave into every critical class. First, we derive every critical class from a recoverable abstract class, which declares the pack and replace functions. This serves as a common interface for every critical object. Then, with the help of AOP, we extend the derived classes with a tailored implementation of these functions. The introspection API of AspectC++ and template meta programming together provide the necessary tools to iterate over all class members.

Unique object ids that allow object identification across replica boundaries are also implemented as an aspect. The *ObjectIds* aspect extends the constructor of each critical class by a `getObjectId`() method call. This method acquires an object id and stores the reference to the object inside the *objectIdStore*. All ids are created from an *textitidGeneratorCounter* that is deterministic across replica boundaries via storyboard. To obtain the reference of an object by its id, a `getReference`() method is provided.

E. Portability

In Summary only minor changes are needed in order to use the *CrossCheck* framework for any other application that fits our requirements from Section III. The most important are SMR that also implies determinism and a well structured C++ code basis.

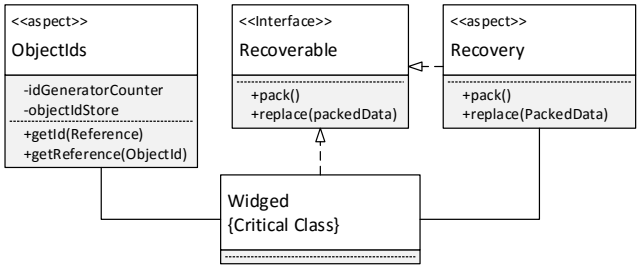


Fig. 8. Aspect-based class extensions for state validation and recovery

Our framework covers all necessary changes to existing data classes via the use of AOP. With this approach marking a class as critical is simply a matter of configuration and a recompilation of the code. This automatically enables state tracking and recovery capabilities. Message exchange, state validation, and synchronization are supported in form of a small library. It is the responsibility of the application developer to integrate the library functions at the right positions in the target application. As usual, this requires minor manual code modifications. For *Memcached++* this was in the range of 100 to 200 lines of code. However this effort depends highly on the architecture of the application.

VII. Evaluation

For the evaluation of our prototype, we set up a cluster of three *Memcached++* replicas, each configured with four worker threads. Each replica was running on a dedicated server machine equipped with two Intel Xeon E5645 CPUs (six cores at 2.40 GHz) and 24 GB RAM. An identical fourth machine was used to simulate clients. All machines were connected with two switched gigabit ethernet networks, one dedicated for to <CHECK> messages and the other for everything else. To simulate clients and generate load we used the benchmark tools *Memaslap* and *Memslap* from *Libmemcached* [2] (v1.0.16).

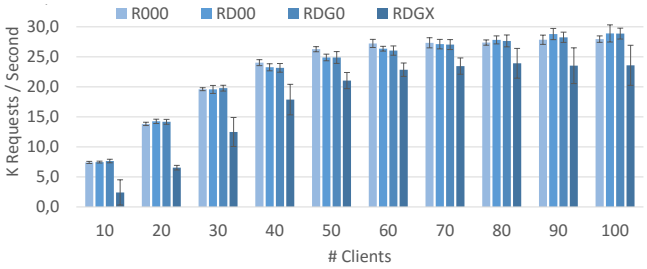
A. Overhead Analysis

To study the performance impact of *CrossCheck* we simulated multiple concurrent clients issuing requests with the Memaslap benchmark tool. We conducted measurements over 2 seconds of operation and repeated these experiments 10 times. All requests were configured with a fixed key-length of 100 B and a fixed value-length of 400 B. We varied the used

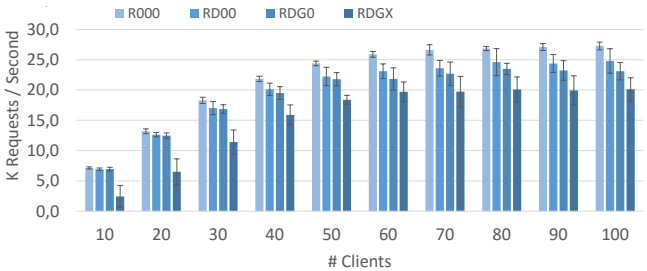
[2]http://libmemcached.org/

TABLE IV
Memcached++ CONFIGURATION CODE

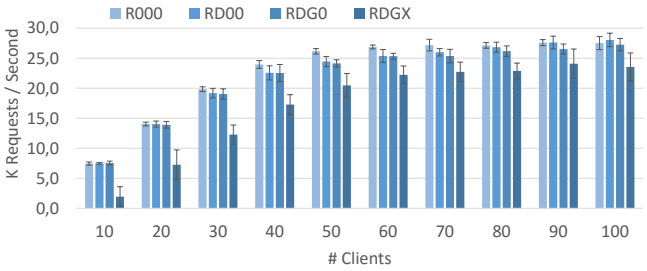
Code	Meaning
R	Replication enabled
D	Deterministic execution enabled
G	GOP enabled
X	*CrossCheck* enabled
0	Feature disabled



(a) 100% GET-Requests

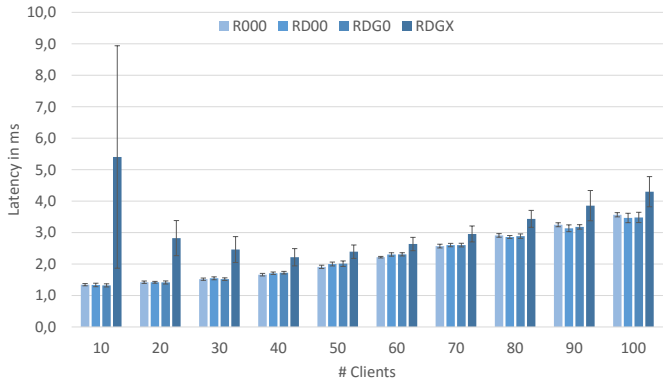


(b) 100% SET-Requests

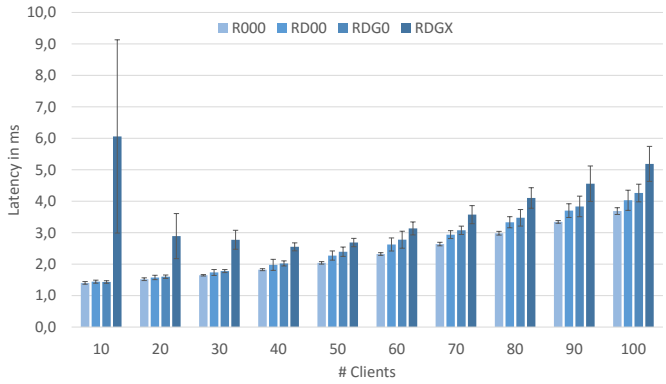


(c) 25% SET and 75% GET-Requests

Fig. 9. Throughput of *Memcached*++ in different configurations



(a) 100% GET-Requests



(b) 100% SET-Requests

Fig. 10. Latency of *Memcached*++ in different configurations

techniques and components in our *Memcached*++ prototype to identify their individual overhead proportion. In all figures the active components are identified by a code, which is explained in Table IV. For instance, a traditional active replication setup is represented by the RD00 configuration that means *Memcached*++ is replicated and requests are processed deterministically.

Throughput: Fig. 9 shows the average throughput for three different workloads consisting of pure GET-requests (Fig. 9a), pure SET-requests (Fig. 9b), and a 75:25 mix of GET- and SET-requests (Fig. 9c). The error bars represent the 95% confidence interval. At 100 concurrent clients, the throughput limit is reached in all configurations of *Memcached*++.

For GET-requests the enforcement of determinism and GOP involve only insignificant overhead. This can be explained by the simple control flow of GET-requests. Only two critical sections are passed and a minimum of 5 accesses to critical objects is needed during GET-requests. In contrast, SET-requests pass 7 critical sections and accesses at least 23 times a critical object. This affects the performance of all components of *Memcached*++ and leads to lower throughput. At 100 concurrent clients, determinism reduces the throughput of SET-requests by 9%, GOP adds another 6 percentage points and finally 11 percentage points are added by *CrossCheck* alone. Compared to a traditional active replication setup that is represented by the RD00 configuration, *CrossCheck* achieves 81% to 86% of the throughput.

Latency: Fig. 10 depicts the average latency together with the 95% confidence interval for GET- and SET-requests. We kept here the same setup as for the throughput measurements. *CrossCheck* adds roughly 1 ms to the latency of requests due to the exchange and comparison of checksums during request processing. For SET-Requests at high access rates, GOP adds also a significant amount of latency as it increases the execution time of requests by multiple checksum computations. The high latency peaks in the RDGX configuration for 10 and 20 clients are caused by message batching. If too few clients are connected, batches don't get filled timely and transmitted only after a timeout that was set to 3 ms in all our experiments. With 30 clients and more this timeout was only seldom triggered.

Execution time variation: Requests in *Memcached*++ are typically processed very quickly. On our machines it takes 15 us to process a GET-Request including a reply and 21 us are needed for SET-Requests on an average. With *CrossCheck* however, more than 500 us are needed to finish a request since <CHECK> messages need to be exchanged first. As explained in Section VI-C, other requests can be processed while waiting for

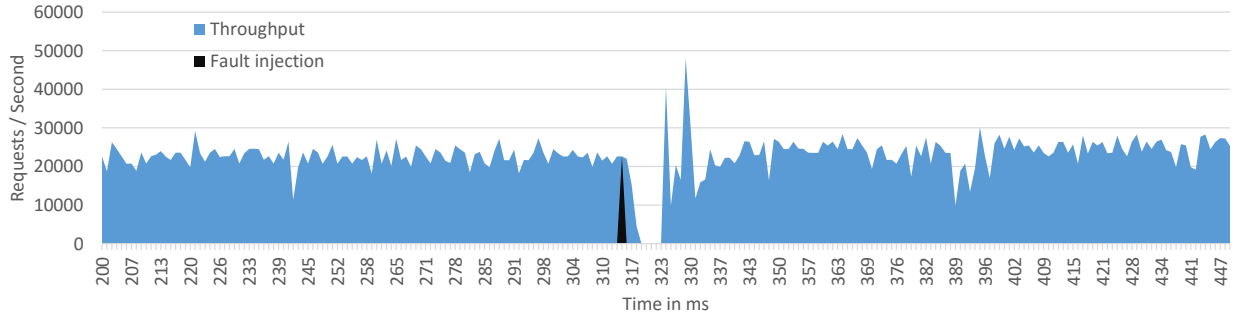


Fig. 12. Influence of a recovery on throughput

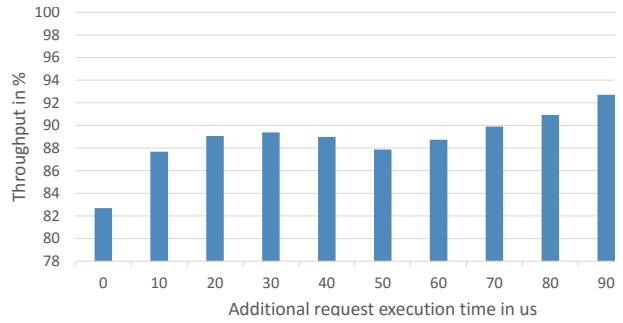


Fig. 11. Throughput of *Memcached*++ in RDGX configuration relative to plain active replication (RD00) for a variation of request execution times

<CHECK> messages. Judging by the execution times of requests, 25-35 requests need to be processed in parallel per worker thread to hide the effect of the <CHECK> messages. However, our system is not capable to serve this high concurrent rate of requests without a performance drop due to the overhead that is added by the total ordering of requests. To investigate the performance of *CrossCheck* when requests would require more processing time, we added artificial overhead into the execution of requests. For this we used the RDTSC instruction [39] in conjunction with an empty loop.

For the measurement campaign we used again the same setup as for the throughput measurement but with a fixed amount of 50 clients. The results of this experiment are presented in Fig. 11 as throughput normalized to the RD00 configuration. Although not linear, a trend can be seen for more complex requests having lower throughput penalty with a minimum of 93% reached at approximately 111 us of total execution time. It should be noted, that by increasing the request execution time by only 10 us (to a total of 31 us on an average) the relative throughput penalty is reduced by 5 percentage points to a total of 88%. We think services like databases, that exhibit longer request execution times, could be very efficiently hardened by *CrossCheck*.

B. Overhead of Recovery

Recovery influence on throughput: In order to analyze the overhead introduced by our recovery mechanism, we used the benchmark tool Memslap and set it up with 100 clients, each issuing 100 SET-requests. Every 2 ms we collected the amount of finished requests and computed the throughput. After counting 5000 requests inside *Memcached*++ we injected a fault in one replica by altering one byte in the request string. This fault propagated to an item object that is the biggest object in our system and eventually caused a recovery of 512 bytes of data. The results of this experiment are shown in Fig. 12. After the fault injection, that is represented as a black spike in the figure, the throughput drops not immediately as we continue execution until all replicas are synchronized. During synchronization we also need to wait until all requests are fully validated by *CrossCheck* and the repair package is transmitted. The throughput gap, that is approximately 10 ms long in this case, is mainly caused by this validation because we have to wait for the slowest replica. This time is highly influenced by the batch transmission timeout that is always triggered during synchronization. After recovery is finished we see a throughput peek since new requests are accepted during recovery and can be immediately processed afterwards. This mitigates the throughput drop during the whole recovery procedure.

Recovery influence on throughput: As mentioned in Section IV-D we cannot prevent false error detections. Although we did not see any false positives during our evaluation, we cannot claim that this may be always the case. To study the throughput impact of unnecessary recoveries we periodically injected faults into the request string after every 5000 requests. We think this is a highly overestimated value that leads to roughly 4 recoveries per second. Like in the previous experiment, every injected fault propagated to an item object. For comparability, we used the same setup as during our throughput measurement in the previous Section VII-A but kept the number of clients fixed at 100. With multiple recoveries the throughput is not much affected and remains at 96% of the fault free case. In summary 86 recoveries were conducted over the course of all 10 experiment runs. On an average 6.4 ms were needed to finish the synchronization while 2.6 ms were needed for packet transmission and the replacement of corrupted data. Altogether the the mean time to complete a recovery was 9 ms

VIII. Conclusions

We presented *CrossCheck*, an approach to harden multi-threaded services against arbitrary state corruptions and crash-

stop failure. This is achieved by selectively protecting critical data objects using an AOP-based generic object protection combined with SMR replication and providing a fine-grained object-level recovery support. Our evaluation based on a multithreaded key-value store shows that *CrossCheck* can reach 86% performance of a classical SMR-based replication, which only tolerates crash stop faults.

ACKNOWLEDGMENT

This work was partly supported by the German Research Foundation (DFG) under priority program SPP1500 grant no. KA 3171/2-3 and SP 968/5-3.

REFERENCES

[1] "2013 cost of data center outages," http://www.emersonnetworkpower.com/documents/en-us/brands/liebert/documents/white%20papers/2013_emerson_data_center_cost_downtime_sl-24680.pdf, Ponemon Institute, 2013, [Online; accessed 10-May-2015].

[2] B. Schroeder, E. Pinheiro, and W.-D. Weber, "Dram errors in the wild: A large-scale field study," in *Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '09, 2009.

[3] 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.

[4] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory errors in modern systems: The good, the bad, and the ugly," in *ASPLOS*, 2015, pp. 297–310.

[5] T. J. Dell, "A white paper on the benefits of chipkill-correct ecc for pc server main memory," *IBM Microelectronics Division*, pp. 1–23, 1997.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] H. Cui, J. Wu, C.-C. Tsai, and J. Yang, "Stable deterministic multithreading through schedule memoization." in *OSDI*, 2010, pp. 207–221.

[15] C. Di Martino, Z. Kalbarczyk, R. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of blue waters," in *Dependable Systems and Networks (DSN)*, 2014.

[16] 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.

[17] R. Kotla and M. Dahlin, "High throughput Byzantine fault tolerance," in *Proc. of the 2004 Conf. on Dependable Systems and Networks*, 2004, pp. 575–584.

[18] A. Bessani, J. Sousa, and E. Alchieri, "State machine replication for the masses with bft-smart," in *Dependable Systems and Networks (DSN)*, 2014.

[19] U. Wappler and C. Fetzer, "Software encoded processing: Building dependable systems with commodity hardware," in *Computer Safety, Reliability, and Security*, 2007, pp. 356–369.

[20] 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.

[21] A. Martens, C. Borchert, T. O. Geißler, D. Lohman, O. Spinczyk, and R. Kapitza, "Crosscheck: Hardening replicated multithreaded services," in *Workshop on Dependability of Clouds, Data Centers and Virtual Machine Technology (DCDV 2014)*, 2014.

[22] 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.

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

[24] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX Annual Technical Conference*, 2014.

[25] 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.

[26] T. Liu, C. Curtsinger, and E. D. Berger, "Dthreads: efficient deterministic multithreading," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011.

[27] H. Cui, J. Wu, C.-C. Tsai, and J. Yang, "Stable deterministic multithreading through schedule memoization." in *OSDI*, 2010.

[28] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman, "Coredet: a compiler and runtime system for deterministic multithreaded execution," in *ACM SIGARCH Computer Architecture News*, 2010.

[29] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: efficient deterministic multithreading in software," *ACM Sigplan Notices*, 2009.

[30] M. Gorman, *Understanding the Linux Virtual Memory Manager*. Prentice Hall PTR, 2004.

[31] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann, and O. Spinczyk, "FAIL*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance," in *Proceedings of the 11th European Dependable Computing Conference (EDCC '15)*, 2015, pp. 245–255.

[32] S. McDermott, L. Jordán, and K. Anderson, "The use of overloaded software operators for error detection and correction," in *18th AIAA/USU Conference on Small Satellites, SSC04-IV-7*, 2004.

[33] I. Stilkerich, M. Strotz, C. Erhardt, M. Hoffmann, D. Lohmann, F. Scheler, and W. Schröder-Preikschat, "A jvm for soft-error-prone embedded systems," in *14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, ser. LCTES '13, 2013.

[34] C. Borchert, H. Schirmeier, and O. Spinczyk, "Generic soft-error detection and correction for concurrent data structures," *IEEE Transactions on Dependable and Secure Computing*, vol. PP, no. 99.

[35] 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.

[36] 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.

[37] P. Koopman, "32-bit cyclic redundancy codes for internet applications," in *Dependable Systems and Networks (DSN)*, 2002.

[38] C. Borchert and O. Spinczyk, "Hardening an L4 microkernel against soft errors by aspect-oriented programming and whole-program analysis," in *Proceedings of the 8th Workshop on Programming Languages and Operating Systems (PLOS '15)*, 2015, pp. 1–7.

[39] "Intel 64 and ia-32 architectures developer's manual, instruction set reference, n-z," vol. 2B, 2015.