Towards Transparent Hardening of Distributed Systems

Diogo Behrens
Christof Fetzer
TU Dresden
Dresden, Germany
diogo.behrens@tu-dresden.de
christof.fetzer@tu-dresden.de

Flavio P. Junqueira
Microsoft Research
Cambridge, UK
fpj@apache.org

Marco Serafini
Qatar Computing Research Institute
Doha, Qatar
mserafini@qf.org.qa

ABSTRACT

In distributed systems, errors such as data corruption or arbitrary changes to the flow of programs might cause processes to propagate incorrect state across the system. To prevent error propagation in such systems, an efficient and effective technique is to harden processes against Arbitrary State Corruption (ASC) faults through local detection, without replication. For distributed systems designed from scratch, dealing with state corruption can be made fully transparent, but requires that developers follow a few concrete design patterns. In this paper, we discuss the problem of hardening existing code bases of distributed systems transparently. Existing systems have not been designed with ASC hardening in mind, so they do not necessarily follow required design patterns. For such systems, we focus here on both performance and number of changes to the existing code base. Using memcached as an example, we identify and discuss three areas of improvement: reducing the memory overhead, improving access to state variables, and supporting multi-threading. Our initial evaluation of memcached shows that our ASC-hardened version obtains a throughput that is roughly 76% of the throughput of stock memcached with 128-byte and 1k-byte messages.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; D.4.7 [**Operating Systems**]: *Distributed systems*

General Terms

Algorithms, Reliability, Performance

Keywords

distributed systems, fault-tolerance, data corruption

1. INTRODUCTION

Large-scale distributed systems are at the core of every successful online service on the Web. Operating such systems at scale comes with a number of challenges related to fault tolerance. Unforeseen failure scenarios, failure propagation, and amplification are not uncommon in such settings. At the same time, identifying faults in such large systems is really a non-trivial task; there are often multiple potential sources and even the most sophisticated tools rarely cover all corners of the system. Often due to lack of data or time to investigate production issues carefully, such issues are resolved by replacing hardware, upgrading software, or simply rebooting. Issues occurring repeatedly are investigated further, but transient errors may never be spotted again.

When designing distributed systems for large-scale infrastructures, the fault model typically adopted is the one of crash faults. Machine and process crashes are commonly observed and from a practical perspective it is a reasonable fault model to adopt. However, the likelihood of non-crash faults like data corruption is typically not negligible in large scale distributed systems. Data corruption due to faulty hardware is also considered rare because when it occurs, it can easily be confused with Heisenbugs or simply go unnoticed. The difficulty of reproducing and pinpointing data corruption does not imply that they can or should be ignored, quite the opposite. The Amazon S3 service is a good example of a service severely affected by such an error. The data corruption of the internal state of a single machine caused a massive outage in July 2008[1] as well as other lesser incidents[2].

In this work, we argue that it is possible to harden systems against both crashes and data corruption with an acceptable performance penalty and a small addition to the development effort. In particular, we target a technique that enables developers to use existing crash-tolerant designs, without requiring dedicated and sophisticated protocols for Byzantine fault tolerance [3].

Correia *et al.* have proposed a technique to harden distributed systems against Arbitrary State Corruption (ASC) faults in a systematic way to tolerate data corruption [4]. It assumes that data corruption manifests as ASC faults: the whole state of a process, including the program counter, can transition to an arbitrary state. The goal of ASC hardening is to transform data corruption, modeled as ASC faults, into crash or omission. The ASC-hardening algorithm of Correia *et al.* has three attractive properties: it is **local**, so it does not require replica processes; it is **generic**, since it provides formal guarantees without knowledge of the application being hardened; it is **untrusted**, since faults can occur even during the execution of the hardening algorithm itself. An ASC hardened version of Paxos was able to tolerate tens of thousands of injected faults in the most frequently accessed code and data segments.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
HotDep'13 November 03 - 06 2013, Farmington, PA, USA
Copyright 2013 ACM 978-1-4503-2457-1/13/11 ...$15.00.

[1] http://status.aws.amazon.com/s3-20080720.html
[2] https://forums.aws.amazon.com/thread.jspa?threadID=22709
https://forums.aws.amazon.com/thread.jspa?messageID=86214

To assess the difficulties of hardening real systems, we endeavored into ASC-hardening memcached[3], an in-memory distributed cache system. We chose memcached because it is representative of a large class of systems, including other distributed in-memory caches, load balancers, Web servers, and applications servers, where integrity is critical and liveness is not strictly necessary, although certainly desirable. Processes in these systems need to provide correct results (safety) but do not need to remain available (liveness) since they can crash, lose all their state, and be replaced. For this class of systems, local detection is a much more convenient choice than fault tolerance since it does not require any replication if ASC hardening is used.

In this context, we found that PASC has three major limitations that we need to overcome to make it more appealing to developers and suitable for production settings:

Memory footprint: PASC essentially doubles the memory footprint of the hardened application because it keeps two copies of the state. For systems like memcached that stores its state primarily in memory, this limitation constitutes a major one;

Flat state: Hardening using the PASC library requires containing all application state in a single object, which is unfeasible for large code bases.

Concurrency: PASC does not allow multiple threads accessing a shared state, which in a multicore world limits the opportunities for concurrent processing.

In this paper, we discuss our progress towards achieving these goals. In particular, we discuss the design of an ASC hardening algorithm that can leverage hardware-level protection across the memory hierarchy to guarantee data integrity; hardening can then focus on detecting data corruptions during computation, which are much harder to detect. Alternatively, the user can use software-level codes that use less space than full replicas. We also argue that it is viable to overcome the current limitation of using a single state object, but dealing with concurrent threads to leverage modern multicore processors is still a challenge. We finally report our progress with hardening existing systems, discussing our memcached prototype. Our initial evaluation of memcached shows that our ASC-hardened version obtains a throughput that is roughly 76% of the throughput of stock memcached with 128-byte and 1k-byte messages.

2. OPTIONS FOR HARDENING

Each individual component in a computer is a potential source of errors. Controllers, processor, main memory, disk, are all prone to errors. DRAM, commonly used for the main memory of commodity servers, has been identified in various studies as an important source of errors [7, 11]. Hardware manufacturers have proposed a number of mechanisms to protect systems against hardware errors. DRAM often includes ECC codes like SECDED codes or codes in the chipkill family [5]; the extent of these techniques, however, is limited to main memory.

Processors also have included error detection and correction at various levels of the memory hierarchy and interconnects. To our knowledge, however, commodity processors do not detect general computation errors nor provide strict end-to-end error detection guarantees at the application level. Developers have also reported that computation errors occur in modern processors, sometimes due to bugs in the manufacturing process or in the design.[4]

The ASC work of Correia *et al.* offers an option to make systems resilient to such corruption errors [4]. Their work includes a model, a hardening algorithm, and a library to harden code bases in an automated fashion called PASC. The ASC model is comprehensive: it captures the notion of intermittent errors and that whenever an error occurs across a computer system, the whole state of a process can transition to an arbitrary value. The ASC-hardening algorithm is local (only redundancy inside a single process is used) and untrusted (it assumes that faults while executing the hardening code may occur). Finally, the PASC library mitigates the burden of hardening systems by automating it when applications follow an assumed structure.

ASC-hardening is not the only software-implemented approach that has been proposed in the literature and here are a couple of relevant alternatives:

Hypervisor-based fault tolerance.

Some approaches use multiple local replicas and voting to prevent error propagation, as for example, the work of Bressoud and Schneider [2]. Preserving fault isolation in the presence of hardware errors using a single server, however, is challenging due to shared hardware resources. On the other hand, if we assume that a fault might corrupt multiple local replicas, it is not necessarily the case that these faults will produce different outputs. ASC hardening executes redundant computations, but does not need to assume that one of them is faulty, or that redundant faulty executions produce different outputs. It does perform output comparisons, but instead of assuming that the outputs must be different in the presence of a fault, it guarantees that they are different.

Byzantine fault tolerance.

Given the body of work on Byzantine Fault Tolerance (BFT), it is natural to consider such a model. The Byzantine model assumes a powerful adversary and consequently a Byzantine-tolerant system is also able to cope with data corruption. Byzantine protocols, however, incorporate features orthogonal to data corruption resilience, such as intrusion or bugs. Intrusion falls into the domain of security and for data center applications, security is often an orthogonal concern (an important one, though). BFT protocols also tolerate software bugs under the premise that a quorum is not affected by such a bug; however, if all replicas of the state machine are identical and deterministic, they will activate the same bugs.

As already mentioned, memcached is just an instance of a large class of systems in which integrity (safety) in the presence of data corruption is sufficient, and availability is not strictly necessary. In the Byzantine model, providing just safety does not significantly change the cost of protocols, see for example the Nysiad system, which achieves safety through replication and agreement [6]. The Thema system also shows the additional complexity of using a fault-tolerance-only approach in Byzantine-tolerant three-tiered Web systems [13].

Software-level error detection.

Software-level error detection has been subject of a large body of research (*e.g.*, SWIFT [14]), but it typically does not provide end-to-end guarantees in distributed systems. Recent work has proposed using encoded execution to harden distributed systems like

[3] `http://memcached.org`

[4] Data corruption with Opteron CPUs and Nvidia chipsets: `https://bugzilla.kernel.org/show_bug.cgi?id=7768`

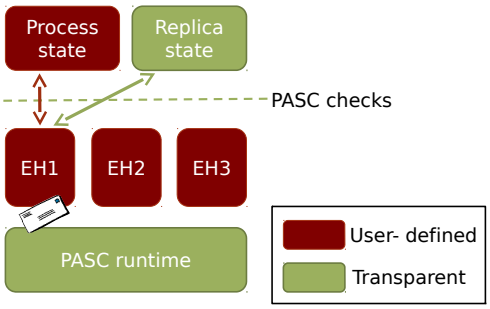

Figure 1: Structure of a PASC-hardened program.

Paxos and provide end-to-end guarantees [1]. Encoded programs can detect faults not covered by the ASC model, *e.g.*, permanent faults such as stuck-at bits. Encoding comes, however, with two main drawbacks. First, it incurs significant overhead, *e.g.*, a 4 to 20 times increased response time. Second, encoding lacks a formal description. In contrast, ASC hardening has been formally proven correct and presents moderate overhead.

3. ASC-HARDENING COMPLEX CODE

The PASC library imposes a particular structure to the (Java) code of hardened processes. A user implementing a PASC-hardened algorithm must define it in terms of three types of objects: one or more *messages*, one or more *event handlers*, and a single flat *process state* object (see Figure 1). PASC provides a generic runtime that transparently coordinates message processing in an event-based fashion and executes the necessary checks. The user must also implement the message passing layer, which feeds the PASC runtime with messages. When the runtime receives a message, it forwards it to the correct even handler. Event handlers can only access local state and the process state; the latter includes all the state that persists across the processing of different messages. PASC transparently replicates the process state and uses the replica state to verify the correctness of the process state whenever event handlers reads from it.

PASC requires that the hardened algorithm use the process state object as an access gateway to any object referenced after handling a message; all reads and writes to any persistent object must take place by calling a method of the process state. Having one process state object serves a couple of important purposes. PASC intercepts calls to process state methods in order to executes state integrity checks and buffers state modifications.

PASC has been used to harden a crash-tolerant version of Paxos and make it tolerant to data corruption. Consider the example of Figure 2, which is a code snippet of the PASC-Paxos implementation. All the objects constituting the persistent Paxos state, like the maximum observed ballot number or the log of currently accepted proposals, must be declared as fields of the `PaxosState` class, which inherits from the PASC-internal `State` class. Except the constructor, all methods of a state class must be either getters or setters. The signature of the methods must follow a particular syntax: getters must be of the form *getX(param)*, where the concatenation of *X* and the optional parameter *param* constitute a unique name for the variable being accessed. Similarly, setters must be of the form *setX(param, newVal)*. PASC uses this convention to determine if the method is reading or writing variables and to uniquely identify the variables being accessed by each method.

Having a single access gateway to the process state has two important limitations. First, it flattens the state onto a single object.

```
public class PaxosState extends State {
  long maxBallot;
  Proposal [] log;

  long getMaxBallot() {
    return maxBallot;
  }
  Proposal getAcceptedProp(long id) {
      return log[id];
  }
  void setAcceptedProp(long id, Proposal p){
      log[id] = p;
  }
}
```

Figure 2: Example of PASC process state

In large code bases, the size and complexity of this gateway can easily become unmanageable. For example, in the PASC-Paxos implementation by Correia *et al.*, the state class has more than 70 methods.

The second limitation of a single access gateway is that it impairs modularity. For correctness, modifications to the process state must occur in the event handlers only. As discussed, the access gateway can only give basic read/write access to event handlers. This is not the way large software systems are typically built; modules often encompass both state and all the necessary logic to process the state.

Solutions.

The use cases described in [4] were implemented from scratch, so it was possible (although presumably cumbersome) to design them according to the strict structure imposed by PASC. However, retrofitting this structure onto existing code bases was unmanageable. We consequently have been seeking more flexible approaches to apply ASC-hardening without imposing a strict structure.

Different from PASC, we have chosen to use C instead of Java. In our experience with Java, it is expensive to transparently manipulate state. Cloning and reflection are techniques that one often has to resort to when manipulating state, which often induce a high performance overhead. In Java, to transparently determine if any application state is being changed, we have needed to intercept method calls (*e.g.*, using aspect-oriented programming). Language extensions like AspectJ[5] do not allow intercepting calls to library methods, which limits substantially the applicability of the approach. Although libraries could be hardened independently, as of today, we do not have such hardened versions. We had to wrap function calls with C too, but they were only a handful of low-level operations. Consequently, given its flexibility and our experience so far, C is a good option to demonstrate the benefits of our techniques.

The direction we are pursuing provides a special interface to memory management instead of using an access gateway, and assumes the whole state is stored in the heap. We are considering two alternative designs. The first is page-level protection. We consider a memory page as the atomic storage unit we protect. The user accesses the heap through an API that wraps the POSIX memory management API: for example, it calls a `malloc_asc()` function to allocate memory, and this function relies in turn on `malloc()` for actually allocating memory. The hardening library detects modifications to the state by initially setting memory pages as read-only and intercepting kernel interrupts arising from write

[5] `http://eclipse.org/aspectj/`

attempts. Whenever a page is modified, it is cloned and later recovered to guarantee that the two executions of the event handler are identical.

The second alternative is word-level protection. Reducing the granularity of our memory management mechanism allows reducing the cost involved in cloning and recovering modified variables. We use the compiler support (namely, the transactional memory support of `gcc`[6]) to intercept all load and store operations reading or modifying the heap. Event handlers are marked with special macro functions as in the following example:

```
__asc_begin();
size_t len = strlen(buf);
process_buffer(buf, len);
__asc_end();
```

The compiler instruments the code between the markers redirecting heap access to our hardening library. This approach copies less data but intercepts calls more often. In Section 7 we evaluate this approach with memcached.

4. REDUCING MEMORY OVERHEAD

PASC requires a full replica of the persistent state of a hardened process. When a process receives a message, it executes event handling twice: the first execution reads from the original state (called S), the second from the replica state (called R). In both executions, the first access to every variable is preceded by a *state integrity check* verifying that the variable is consistent with its replica.

Solutions.

Our new ASC-hardening library also uses state integrity checks, but instead of using a full replica state R, it leverages error detection codes present in modern hardware. These can be either hardware-level codes like ECC, with zero memory and CPU overhead, or user-defined software-level codes, with flexible trade offs between error coverage, CPU overhead, and memory overhead. It is worth noting that hardware-level mechanisms alone do not solve the hardening problem, since they detect data corruption while data is stored but not during computation. The data could be read, corrupted arbitrarily, and written back. Memory protection mechanisms do not catch such cases.

Using only one full copy of the state implies that the hardening algorithm must use S for both executions of the event handler. This change requires modifying the hardening algorithm in a few key aspects, *e.g.*, to make sure that errors do not propagate the same way in both executions. PASC uses software-level replication and assumes that R is under control: variables in R and in S are updated in different phases. By contrast, if R is handled transparently at the hardware level, for example in the form of an ECC, this is no longer possible: every time a variable in S is modified, its corresponding replica variable in R is set to (an encoding of) the same value. To enable leveraging hardware-level memory protection, our new hardening algorithm uses exactly this access pattern, updating the replica of a variable immediately after updating the original variable. The same pattern is used with software-level codes. This scheme is correct because we treat the hardware-generated code for memory words as the copy R of the state. The argument of correctness for our new algorithm consequently builds upon the one for PASC with this observation.

5. CONCURRENCY

Given the current trend of multicore processors, it is not realistic to assume or constrain systems to be single-threaded. It is quite critical for performance to enable the use of multiple threads. ASC hardening, however, works for processes that can be abstracted as deterministic state machines and the presence of multiple threads with shared state introduces non-determinism. Determinism is needed because event handlers are executed twice and both executions are expected to produce the same results. This is a common assumption in all state machine replication algorithms; nonetheless, existing distributed systems are rarely designed as state machines.

To illustrate the problem, consider the following execution of two ASC-hardened threads t_1 and t_2 that share a variable v. The execution presents the following interleaving of events. Thread t_1 reads $v = 10$ during the first execution of an event handler, and computes $u = v + 1 = 11$. Between the first and second execution of the handler of thread t_1, thread t_2 sets v to 50. The second execution of t_1 results now in $u = 51$, a divergence that can be incorrectly perceived as a data corruption.

The problem of non-determinism and the efficient support of multi-threading in state machines applies also to state machine replication and has been discussed in the literature [9, 15]. The additional challenge with ASC hardening is that the mechanisms to handle multi-threading need to be made ASC-tolerant too; for example, if locking is used, we must consider that the value of a lock may be corrupted.

Solutions.

Our current approach with memcached is to run several single-process instances instead of a single multi-threaded instance. Recent work on Dthreads has shown that implementing the `pthread` thread abstraction using multiple processes with no shared state can match and occasionally exceed the performance of a standard `pthread` library implementation [12]. An alternative is to execute hardened handlers atomically, providing isolation between concurrent handler executions. An option to provide atomicity and isolation is to leverage software transactional memory techniques [16]. Such techniques, however, do not provide to date the level of performance overhead we expect. Whether we can enable multi-threading with a lightweight mechanism, perhaps based on transactional memory, is an open question.

6. ASC-HARDENING MEMCACHED

Memcached is a popular in-memory key-value cache. It provides a server that exposes a simple `get/set` interface and it is implemented in C. Clients of memcached are implemented separately and we have used the memaslap client[7] version 1.2. We have used the version 1.4.15 of the server code. We now describe a few challenges of ASC-hardening memcached.

Identifying the event handlers.

ASC hardening assumes event handling starts with the receipt of a message. In practice, the reading of a message from the socket and its processing are often interleaved, for example, the *set* commands in memcached. Hence, one challenge when hardening memcached was choosing the right place where to start and finish the hardened event handlers. We start our handlers before the tokenization of the input buffer and before the processing of the payload.

Message integrity.

ASC hardening calculates 32-bit CRCs of messages upon receiving and sending. Efficient CRC computation was critical in our implementation to obtain a reasonable performance penalty. Our

[6] `http://gcc.gnu.org/wiki/TransactionalMemory`

[7] `http://libmemcached.org`

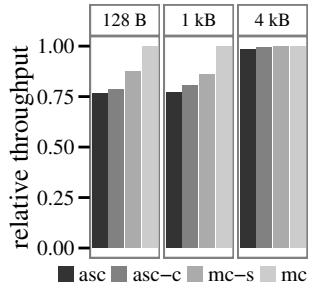

Figure 3: Throughput for different message sizes

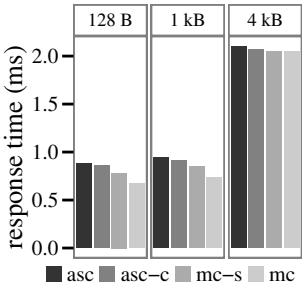

Figure 4: Response time for different message sizes

implementation calculates CRCs using the SSE4.2 hardware extensions [8] when available, otherwise it falls back to the efficient slicing-by-8 algorithm [10].

Prototype limitations.

Our prototype computes 32-bit CRCs, but it does not send them along with the messages. We have not implemented it for convenience, since it reduces our changes to memcached and we expect the performance impact to be minimal once we add it. We have also disabled hashtable rebalancing and slab allocation of memcached, since they require multiple threads and this is not yet supported by our ASC-hardening approach (Section 5). Finally, memcached periodically reads the clock and updates the `current_time` global variable. Since these updates are non-deterministic, we serialize the accesses to `current_time`.

7. PRELIMINARY RESULTS

We present in this section a preliminary evaluation of the overhead of our ASC-hardened memcached. Memcached runs with a single worker thread and 1 GB cache size on a 2.66 GHz Intel Xeon X5650 machine (Linux 3.8 kernel). Memaslap measures achievable throughput and response time. It runs on another machine with similar configuration; both machines are connected via Gigabit Ethernet. Each experiment consists of 60s runs of memaslap with 64 connections. Memaslap produces a workload of 10% sets and 90% gets. We experiment with message sizes varying from 128 bytes to 4 k bytes and with the following memcached variants: *mc* is the stock memcached; *mc-s* is a stripped down version of memcached without slab allocator; *asc* is memcached instrumented and running the new ASC-hardening algorithm with hardware memory protection; and *asc-c* is *asc* without CRC calculation.

Figure 3 shows the measured throughput for the memcached variants relative to *mc*. With 128 bytes and 1 kB messages *asc* can provide 76% of *mc*'s throughput. Note that the difference between *asc-c* and *asc* is small, since the CRC is efficiently implemented using SSE4.2 extensions. With large 4 kB messages all variants are network bound. The remaining CPU resources can be used for hardening without affecting the throughput.

The response time given by memaslap for the same experiments is depicted in Figure 4. For smaller messages (128 B and 1 kB message), the overhead of *asc* is more pronounced because the system is CPU-bound. The response time difference between *asc* and *mc* is less than 200 μs. Note that latency is typically the main performance factor that memcached applications care about. Throughput is also important, but typically secondary.

The *mc-s* variant provides about 85% of *mc*'s throughput for 128 kB and 1 kB message sizes. Remember that *mc-s* is a stripped down version of *mc* without slab allocation – it contains no hardening. An open question of our experiments is whether the slab allocation feature, if present, reduces the overhead of *asc* further.

We added about 50 code lines to memcached, 8 of them event handler markers, 7 of them CRC related. More than 120 functions were automatically instrumented.

8. CONCLUSIONS

Although data corruption may seem very unlikely in the everyday experience of most practitioners, they are a statistical certainty in large-scale distributed systems. While adequate hardware memory protection represents a valid protection against errors occurring in memory, computational errors cannot be as easily detected. The overall goal of our research is to enhance the fault tolerance of existing distributed systems in a principled and automated manner, yet with minimal changes to the existing code base and minimal performance overhead. This paper considers memcached as use case and shows promising initial results: transparent and principled hardening based on the ASC model can be achieved with a moderate performance overhead and with minimal changes to the source code. Our current work is on consolidating these results and on expanding transparent hardening to concurrent systems.

9. ACKNOWLEDGMENTS

This work is partly supported by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden".

10. REFERENCES

[1] D. Behrens, S. Weigert, and C. Fetzer. Automatically tolerating arbitrary faults in non-malicious settings. In *Proceedings of the Sixth Latin-American Symposium on Dependable Computing (LADC)*, pages 114–123, April 2013.

[2] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. Comput. Syst.*, 14:80–107, February 1996.

[3] M. Castro and B. Liskov, Practical Byzantine fault tolerance. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[4] M. Correia, D. G. Ferro, F. Junqueira, and M. Serafini. Practical hardening of crash-tolerant systems. In *USENIX Annual Technical Conference*, 2012.

[5] T. J. Dell. A white paper on the benefits of chipkill- correct ECC for PC server main memory. Technical report, IBM Microelectronics Division, 1997.

[6] C. Ho, R. van Renesse, M. Bickford, and D. Dolev. Nysiad: Practical protocol transformation to tolerate byzantine failures. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[7] A. A. Hwang, I. Stefanovici, and B. Schroeder. Cosmic rays don't strike twice: Understanding the nature of DRAM errors and the implications for system design. In *ASPLOS*, 2012.

[8] Intel. *Intel SSE4 Programming Reference*, 2007.

[9] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about eve: execute-verify replication for multi-core servers. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 237–250, 2012.

[10] M. E. Kounavis and F. L. Berry. A systematic approach to building high performance software-based crc generators.

2012 IEEE Symposium on Computers and Communications (ISCC), 0:855–862, 2005.

[11] X. Li, M. C. Huang, K. Shen, and L. Chu. A realistic evaluation of memory hardware errors and software system susceptibility. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 6–16, Berkeley, CA, USA, 2010. USENIX Association.

[12] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 327–336. ACM, 2011.

[13] M. G. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan. Thema: Byzantine-fault-tolerant middleware forweb-service applications. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, SRDS '05, pages 131–142, Washington, DC, USA, 2005. IEEE Computer Society.

[14] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August. SWIFT: software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 243–254, Mar. 2005.

[15] R. Rodrigues, M. Castro, and B. Liskov. Base: using abstraction to improve fault tolerance. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 15–28, New York, NY, USA, 2001. ACM.

[16] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.