Transparent Fault-Tolerance using Intra-Machine Full-Software-Stack Replication on Commodity Multicore Hardware

Giuliano Losa[ϕ], Antonio Barbalace[ρ], Yuzhong Wen[γ], Marina Sadini[β], Ho-Ren Chuang, Binoy Ravindran
ECE, Virginia Tech
{*giuliano.losa, antoniob, wyz2014, sadini, horenc, binoy*}*@vt.edu*

***Abstract*—As the number of processors and the size of the memory of computing systems keep increasing, the likelihood of CPU core failures, memory errors, and bus failures increases and can threaten system availability. Software components can be hardened against such failures by running several replicas of a component on hardware replicas that fail independently and that are coordinated by a State-Machine Replication protocol. One common solution is to replicate the physical machine to provide redundancy, and to rewrite the software to address coordination. However, a CPU core failure, a memory error, or a bus error is unlikely to always crash an entire machine. Thus, full machine replication may sometimes be an overkill, increasing resource costs.**

In this paper, we introduce full software stack replication within a single commodity machine. Our approach runs replicas on fault-independent hardware partitions (e.g., NUMA nodes), wherein each partition is software-isolated from the others and has its own CPU cores, memory, and full software stack. A hardware failure in one partition can be recovered by another partition taking over its functionality. We have realized this vision by implementing FT-Linux, a Linux-based operating system that transparently replicates race-free, multi-threaded POSIX applications on different hardware partitions of a single machine. Our evaluations of FT-Linux on several popular Linux applications show a worst case slowdown (due to replication) by ≈20%.

1. Introduction

Application demands have broadly resulted in increased processor parallelism (with the so-called end of Moore's law) and larger memory sizes. However, these changes in chip designs have required higher densities of circuits, causing manufacturing errors, heating issues, etc, and have resulted in soft and hard faults [1]. Since hardware faults are largely uncorrelated among components, fault-resilient software must provide availability in the presence of component failures. This problem has already been recognized in domains including data-centers [1], [2] and embedded systems [3]. Recent solutions for intra-machine fault-tolerance exclusively focus on application-level fault-tolerance [3]. Nevertheless, faults can happen even during OS kernel/monitor execution [4], hampering the stability of all applications running atop.

Recently, Shalev et al. [5] have addressed this problem for CPU core failures in commodity multicore hardware. However, their solution does not mask the failure from an application standpoint and sustain (application) functionality i.e., they only ensure that the OS remains alive after a core failure. In addition, they exclude memory and bus errors. In the distant past, full software stack intra-machine fault tolerance has been proposed by replicating hardware components, like in the case of Tandem computers [6] and lock-step processors, which require costly, special-purpose hardware. On the other end of the spectrum, software-only full software-stack fault tolerance, using commodity hardware, has been exclusively developed for the inter-machine case, cf. State-Machine Replication (SMR) [2]. We explore the intersection of these two solution spaces and propose a novel software design.

State-Machine Replication. SMR is a widely used technique to provide fault-tolerance of computing services. The technique involves running multiple replicas of a service on different machines that are assumed to fail independently. The execution of the replicas are coordinated to provide the abstraction of a single service to clients. A popular flavor of SMR is Primary-Backup, in which a unique replica, the primary, serves all client requests. Before executing an operation, the primary typically waits for a majority of the replicas so that, would replicas crash, at least one of them would still be available to provide the service. Hence, each operation costs at least a round-trip exchange with a majority of replicas.

In replication solutions, replicas are often physically separated, e.g., housed in different racks in a data center, or in different data-centers. However, physically separating the replicas is not always possible, such as in an embedded system where space constraints usually exist.

Additionally, physical separation usually requires duplication of the entire physical machine and its components, thus doubling the volume, and power consumption – thereby, doubling monetary costs. As a result, being able to tolerate core and memory failures within a machine is a cost-efficient solution.

Moreover, from a performance perspective, physical sep-

[ϕ] *The author is now with CS, UCLA.*
[ρ] *The author is now with Euler, Huawei German Research Center.*
[γ] *The author is now with OS Kernel Lab, Huawei.*
[β] *The author is not anymore with ECE, Virginia Tech.*

aration comes at a price: farther apart the replicas, longer the round-trip communication between replicas, and higher the overhead of replication. For example, Guerraoui et al. [7] measured a propagation delay of $0.55\mu s$, on average, when repeatedly sending messages from one core to another in a multi-core machine. In contrast, they measured a propagation delay of $135\mu s$ in a LAN, almost three orders of magnitude more. Newer networking infrastructures such as RDMA can reduce the latency of inter-machine communication and thereby reduce the gap to only one order of magnitude [8], but that is still a significant difference.

Intra-Machine Replication. When replicas are in close proximity, e.g., on the same electronic board in a dual-core lock-step processor, the performance of the replicated system can be close to that of the same system running within a single processor. However, dual-core lock-step processors are nowadays not an attractive proposition because of their lower performance when compared to recent multi-cores, higher acquisition costs, common mode failures, and lack of flexibility (all computation is replicated). Moreover, high-performance cores are non-deterministic, which makes it difficult to synchronize their state. Therefore, it is not clear whether high-performance lock-step configurations are likely to appear in the future, especially in application settings where COTS components are used.

Contributions. We investigate whether a variant of software Primary-Backup replication, wherein replicas run on different hardware partitions of the same electronic board of a commodity machine, is an interesting middle-ground between Primary-Backup replication on physically separated machines and lock-step processors. We target modern highly parallel multi-core machines that are built using multiple copies of the same circuit, each of which can be considered as an independent failure unit. We exploit hardware error monitoring mechanisms that are available in commodity multi-core machines to detect hardware failures. We focus on transparent Primary-Backup replication, which enables legacy applications to take advantage of our proposed fault tolerance mechanism without making any modifications, and also for applications that are developed without replication in mind.

We have developed FT-Linux, a prototype operating system built on Popcorn Linux [9]. FT-Linux's design borrows elements from FT-TCP [10], TFT [11], and Rex [12]:

- To maximize resilience to memory errors, FT-Linux has no software component that is a single point of failure: all Linux replicas run on bare metal, communicate via a "mail box" area in shared memory, and otherwise run on disjoint partitions of the hardware resources of the machine.
- To maximize performance, FT-Linux implements a hybrid replication strategy inspired by FT-TCP [10], [13]: the Linux TCP stack is replicated using incremental checkpointing; user-space applications are replicated using active replication, where the primary replica logs its non-deterministic execution steps and distributes the log to the replicas, which replay it.
- From the perspective of POSIX applications, replication in FT-Linux is completely transparent. FT-Linux replicates applications using a method inspired by Rex [12]: a primary FT-Linux replica records its execution and distributes a partially ordered log of the non-deterministic events to the replicas, which replay unordered events in parallel and thereby retain most of the parallelism and performance of non-replicated execution.

Using FT-Linux, we measured the performance of popular Linux applications including the Mongoose web server and the PBZIP2 compression utility, in various configurations, and compared against an Ubuntu-packaged Linux kernel. The results of our measurements, show that FT-Linux has a moderate performance overhead, achieving 80% of the throughput of an unmodified Linux kernel in practical situations.

The rest of the paper is organized as follows. Section 2 discusses FT-Linux's failure model and Section 3 discusses FT-Linux's design and implementation. Section 4 presents the experimental setup and evaluation results. Section 5 discusses past work, Section 6 identifies possible future extensions, and Section 7 draws the work's conclusions.

2. Failure Model

2.1. Fault Types

We say that FT-Linux implements a type of fault tolerance that we call intra-machine fault-tolerant replication (or intra-machine fault tolerance for conciseness). We argue that intra-machine fault tolerance is no more restrictive than fault-tolerance that is provided using replication on physically separate machines. In fact, a modern highly parallel multi-core machine already replicates various hardware components, including cores, memory controllers, I/O controllers, bus links, etc. (Note that redundant power supply is used in commodity server-grade machines, and also in life/safety-critical embedded systems.) Thus, a CPU socket or a NUMA node can be considered as an independent failure unit.

FT-Linux tolerates core, memory, and bus failures that affect a single partition, and that can be detected before they cause cross-replica contamination. These include fail-stop faults and data-corruption faults. The latter type of faults include memory and bus faults that are detected but not corrected by error detection and correction codes. Examples include ECC detection and reporting hardware, which cause hardware exceptions to be reported to the operating system, such as Intel Machine Check Architecture (MCA) and Advanced Error Reporting (AER) [14]. Our work presumes the existence of such hardware error detection technologies.

Intra-machine fault tolerance could also cover data-corruption faults that are not detected in hardware, but that would require a voting scheme involving at least three replicas, so that a majority vote can be used to filter out corrupted data. Even then, faults that corrupt in-kernel data structures

that are used to enforce memory protection, such as the page table, may cause one replica to make arbitrary writes in the memory of another replica, causing cascading failures. However, we expect faults that cause memory partitioning violations to be rare, given the low memory footprint of the data-structures involved. Currently, the FT-Linux prototype supports only two replicas; extensions to more than two are future work.

2.2. Likelihood of Tolerated Faults

A recent study on memory errors [15], which collected memory error data on the entire fleet of servers at Facebook during a full year, observed that 0.03% of the servers were affected by detected but uncorrected memory errors each month, and that 2% of the servers suffered correctable memory errors. Among those 2% of servers suffering correctable memory errors, 10% became unresponsive, which we consider a failure, because they were bombarded by hardware exceptions caused by the errors. Overall, 0.05% of the servers failed each month due to memory errors that can be prevented by intra-machine fault tolerance, and 0.02% of the servers failed each month due to memory errors that FT-Linux would have tolerated. These numbers hardly seem to justify the use of replication. However, it is likely that software would increasingly be able to handle faulty situations; memory in future server fleets may not be as reliable as today's, and even low fault rates can significantly impact long running jobs, as in scientific computing applications.

Finally, some embedded systems run in harsh environments where temperature, radiation, vibration, and other factors increase the likelihood of the type of hardware faults that are tolerated by FT-Linux. It is becoming increasingly common that embedded systems are not built using special-purpose hardware, but using COTS hardware, increasing their fault tolerance requirements, fostering software solutions.

2.3. Likelihood of Faults in Kernel

As reported by Shalev et al. [5], a CPU core fail-stop takes down the entire machine. However, there is no similar work that reports memory (or bus) errors affecting applications running on a monolithic kernel (Linux includes a memory fault handling framework cf. `mm/memory-failure.c`). To estimate the impact of a memory error on Linux, we ran the memcached server [16] with a load generator from CloudSuite [17] as a representative data center application. We interrupted the execution and simply dumped the physical memory state. We repeated this experiment for several loads, thereby measuring the probability of the error hitting different parts of the kernel.

The results, shown in Figure 1, reveal that, on a 64 core machine with $96GB$ of RAM, up to 15% of the RAM contains kernel data and is not recoverable with Linux-implemented memory fault-tolerance (labeled as Ignored). For the same dataset ("180x"), another 20% of the memory

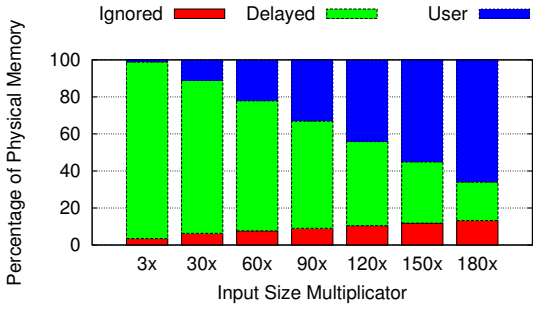

Figure 1. Memory dump of a Linux system running the memcached application under varying cached data size. The user-space memory consumed by memcached is "User." The kernel-space memory is labeled "Ignored" and "Delayed" [18], respectively, indicating an unrecoverable and recoverable memory area when hit by a memory fault.

is used by the kernel, but Linux is able to continue operation without immediate failure (Delayed). Note that if the memory error hits the application (User), the application will likely be killed. Since memcached does not extensively use the page cache, we expect that the percentage of hitting a sensible kernel area that is not recoverable will only increase for other benchmarks from the reported 35%.

3. FT-Linux Design and Implementation

FT-Linux is based on the following design principles:

1) the hardware should be strictly divided among software replicas;
2) not all the applications running on a system require fault tolerance;
3) multithreaded applications and (multithreaded) kernel services must be efficiently replicated;
4) when the primary replica fails, the backup replica (secondary) should quickly take over I/O seamlessly.

Hardware Partitioning. As shown in Figure 2, FT-Linux partitions the cores, memory, and I/O devices of a machine into isolated partitions, and runs one Linux kernel on each partition. Each I/O device is owned exclusively by one kernel. One of the partitions, called the primary, loads a full Linux distribution, such as Ubuntu. The other partition, called the secondary, loads a minimal user-space environment. Initially, both kernels execute completely independently. FT-Linux inherits the ability to boot multiple kernels on separate resource partitions from Popcorn Linux [9] (Section 3.1).

FT-Namespace. FT-Linux implements a new Linux namespace, FT-Namespace, in order to isolate applications that require fault tolerance from the one that do not require it. Applications created in a FT-Namespace will utilize full software-stack replication. Once an user enters the FT-Namespace, all the applications that run inside it will be replicated in the secondary kernel. The launching procedure inside FT-Namespace includes replicating all the environment variables to the corresponding replicated process on the secondary, so that both sides are launching the application with the same initial environment (Section 3.3).

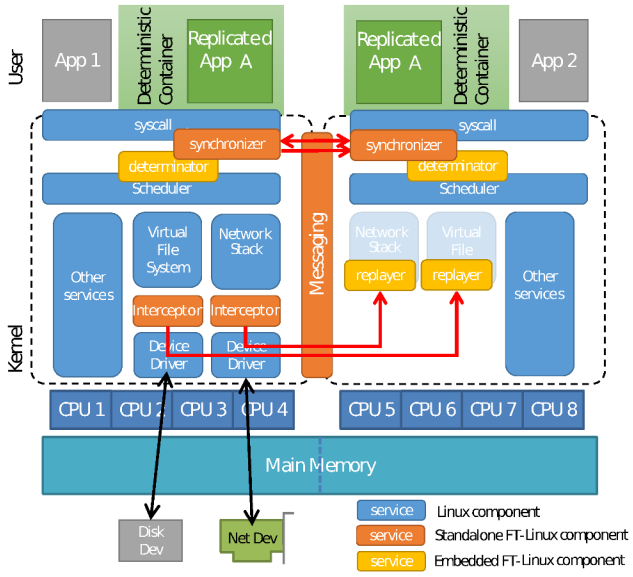

Figure 2. FT-Linux software architecture atop a multi-CPU platform in normal operation (i.e., before any failure).

Replicating the Application. For multithreaded applications, the non-deterministic thread interleaving leads to non-deterministic states and outputs. This becomes an issue when trying to replicate a multithreaded application because the output of the replicas must be an exact (deterministic) copy of the output of the primary. To tame the non-determinism, we implemented active replication (Section 3.2) to synchronize the thread-interleaving for multithreaded applications. We call this FT-Linux's *system-call synchronization component*. Non-deterministic system calls are transparently intercepted by this component that logs primary I/O and replays it on the secondary replica.

Replicating Operating System Services. In order to correctly replicate system services beyond system call replication, I/O is replicated above the driver interface in order to maintain a consistent kernel service state, which allows fast failover. To support network activities, we replicate the TCP stack; its state is kept synchronized using an ad-hoc technique inspired by FT-TCP [10] (Section 3.4). The technique consists of keeping the send and receive queues, sequence number, and a few other TCP parameters synchronized between primary and secondary. In effect, FT-Linux synchronizes checkpoints of the *logical state* of the TCP stack through incremental updates.

Failover. The primary and backup keep sending heart beat messages to each other periodically. When the secondary does not hear from the primary after a timeout, it will send an interrupt to it to forcefully halt. Additionally, modern hardware error monitoring is used to detect that a hardware partition is malfunctioning and to eventually restart the software on it. Subsequently, the secondary initiates device-ownership transfer by reloading the drivers of any device used by the primary, including the NIC. Once the NIC that is used to communicate with application clients becomes available on the secondary, it uses the logical TCP state to restore the primary's network sockets, including send and receive queues, sequence numbers, and other parameters. Finally, the application replicas switch from live replay mode to live execution mode (Section 3.7).

3.1. Popcorn Linux Extensions

Popcorn Linux [9] is a replicated kernel operating system that strives to improve the scalability of the Linux operating system on multi-core architectures. It provides a traditional Linux system image to applications above multiple independent kernels. Although we do not make use of the ability of Popcorn Linux to provide a single-system image, FT-Linux reuses the ability of Popcorn Linux to partition the hardware resources of a shared-memory multi-core machine, to boot one independent kernel on each partition, and its inter-kernel shared-memory messaging layer.

Popcorn Linux provides a messaging layer to allow kernel instances to communicate through message passing. In FT-Linux's setting, since all kernel instances reside on the same machine, it is possible to deploy Popcorn Linux's messaging layer over shared memory, and thereby minimize as much communication overhead as possible. Each kernel owns a receiving queue of fixed-size packets, which are consumed circularly by the kernel itself, thereby implementing a ring buffer for incoming messages. The memory reserved for each buffer is local to the partition assigned to each kernel, and the address of the buffer is shared at boot time by writing it on a small table accessible by all kernels.

Senders can directly write on the ring buffer of the receiving kernel, and message notifications are delivered through a combination of inter-processor interrupts (IPIs) combined with polling to reduce overhead. Since the buffer's memory is local to each kernel, once a message has been written on the receiver's ring buffer, the message will be always delivered unless the receiver itself fails. This allows FT-Linux to avoid using acknowledgements between replicas while providing reliable messaging with high probability: once a replica completes its shared memory write, the message will be delivered if the recipient does not fail, unless the sender experiences a failure while its write has not been propagated out of its local CPU cache, and the failure disrupts cache coherency. This is further discussed in Section 3.5.

3.2. Active Replication Above the System-Call Interface

In FT-Linux, both replicas actively execute the application above the system-call interface using a model similar to the execute-agree-follow model of Rex [12]. FT-Linux interposes its runtime system at the system call interface, and at the POSIX Pthreads interface, both on the primary and the secondary. The FT-Linux runtime manages the execution of the system calls, including `gettimeofday`, `time`, `accept`, `accept4`, `rcv`, `rcv_from`, `send`, `sendto`, `epoll`, `epoll_wait`,

and poll, and the Pthreads functions, exclusively cond_signal, cond_wait, cond_timedwait, mutex_lock, mutex_trylock, rwlock_wrlock, rwlock_trywrlock, rwlock_tryrdlock, and rwlock_rdlock. Interposition at the system call interface is performed in the kernel by modifying the system call handler. Interposition at the Pthreads interface is performed by linking applications with a modified version of the Pthreads library.

The primary executes a replicated application as on Linux, except that:

- for each thread, the primary streams the sequence of executed system calls to the replica, but does not otherwise interfere with system call execution;
- FT-Linux enforces a total order on all Pthreads operations (using a global lock), and streams the observed sequence of Pthreads operations to the replica.

The replica executes the application normally only above the system call and Pthreads interfaces. Below those interfaces, execution is managed by the FT-Linux runtime:

- FT-Linux intercepts all Pthreads calls and forwards them to the original POSIX Pthreads implementation so as to produce the same outcomes as on the primary. For example in a race on a lock, the lock call of the thread that won the race on the primary will be forwarded first, and the lock call of the other thread will be forwarded only after the lock is acquired;
- FT-Linux intercepts all calls to the poll and epoll I/O event notification APIs and returns the same values as on the primary. FT-Linux also maintains the kernel objects related to epoll (e.g., interest sets) so as to allow transition to unmanaged execution after failover.
- FT-Linux intercepts TCP socket system calls, does not forward them to the TCP stack, and instead copies and returns the same values as on the primary. The state of the TCP stack is managed internally by the TCP-Stack replication component, described in Section 3.4, so as to enable transition to unmanaged execution upon failover.

3.3. Replicating Multithreaded Applications

As mentioned above, we are targeting to synchronize the execution order of Pthreads primitives. FT-Linux provides two system calls called __det_start and __det_end . Any execution between a __det_start __det_end pair will be logged on the primary and be replayed on the secondary. We denote a code section which is surrounded by __det_start and __det_end a "determinisitic section". Figure 3 shows a simplified implementation of it.

__det_start and __det_end will serialize all the deterministic sections by using a global mutex to control the mutual exclusion. Every thread in the FT-Namespace maintains a sequence number Seq_{thread} and the entire namespace maintains a sequence number Seq_{global}. On the primary, __det_start simply locks the global mutex. __det_end unlocks the global mutex, sends a tuple of

```
1  /*
2   * ns: current Popcorn Linux namespace
3   * ns->global_mutex: global_mutex in current
        namespace
4   * ns->seq: global sequence number
      Seq_global
5   * current->seq: task sequence number
      Seq_thread
6   * current->ft_pid: replicated task unique
      identifier
7   */
8  void __det_start()
9  {
10     if (is_secondary(current))
11         wait_for_sync(current->seq,
12             ns->seq, current->ft_pid);
13     lock(ns->global_mutex);
14 }
15 void __det_end()
16 {
17     if (is_primary(current))
18         send_sync(current->seq,
19             ns->seq, current->ft_pid);
20     current->seq++;
21     ns->seq++;
22     unlock(ns->global_mutex);
23 }
```

Figure 3. Simplified implementation of __det_start and __det_end syscalls.

$< Seq_{thread}, Seq_{global}, ft\_pid >$ to the secondary and then increases the value of Seq_{global} and Seq_{thread}. On the secondary, __det_start blocks until it receives a $< Seq_{thread}, Seq_{global}, ft\_pid >$ tuple corresponding to its caller thread, then holds the global mutex, and __det_end increases Seq_{global} and Seq_{thread}, then the global mutex is released.

In order to provide a transparent interface to applications, we have re-implemented a set of Pthreads' synchronization primitives instrumented with our __det_start and __det_end syscalls. The re-implemented functions are built into a shared library that can be loaded via the LD_PRELOAD environment variable, so that applications can call our implementations instead the original ones in Glibc.

For pthread_mutex_lock (and other lock primitives), we simply put __det_start and __det_end around it to enforce the lock acquisition order. Moreover, we have modified Linux futex implementation to enforce a FIFO behavior on the futex queue, so that the order of possessing a futex will lead to a deterministic order of releasing it. With the above being implemented, the order of acquiring pthread_mutex_lock can be synchronized between the primary and the secondary.

Here, pthread_cond_wait needs to be treated specially. The timing of getting into futex_wait and waking up from it is non-deterministic. According to Glibc's implementation, the futex_wait inside pthread_cond_wait only gets woken up when

the condition variable differs from what it was when `pthread_cond_wait` was called previously. This can be caused by calling `pthread_cond_signal`, or calling another `pthread_cond_wait` with a specific interleaving. As a result, by synchronizing the access sequence of condition variables inside the implementation of `pthread_cond_wait` and `pthread_cond_signal`, we are able to synchronize the wakeup sequence of `pthread_cond_wait`. This is done by providing a new implementation of `pthread_cond_wait` and of `pthread_cond_signal`, with `__det_start` and `__det_end` protecting the access to the internal condition variables. The re-implemented `pthread_cond_wait` is in library that we load with LD_PRELOAD. `pthread_cond_timedwait` is treated in the same way. In addition, for the timeout case in `pthread_cond_timedwait`, since we synchronized the result of `gettimeofday` between primary and secondary, the wake up time is inherently synchronized.

3.4. TCP-Stack Replication

To provide transparent failover, it is not sufficient to replicate user-space applications because application-visible state might also be held in the kernel, such as in the TCP stack. A possibility would be to apply live replication to the TCP stack, as is done for the applications. While it is possible to replicate applications using live replay, because non-deterministic actions can be trapped and controlled by an underlying software layer, it is more difficult to do so for the OS kernel. The Linux kernel is highly non-deterministic and replaying its execution faithfully on the replica would require controlling hardware non-determinism, possibly using a hypervisor. The work of Bressoud and Schneider [19] uses an hypervisor, but that would introduce a single point of failure and a likely performance bottleneck.

Instead, we maintain a synchronized copy of the logical state of the primary's TCP stack on the secondary. The logical state of the TCP stack is designed so that, upon failover, the new primary can bring its original TCP stack in a state that is indistinguishable to the application from the last externally visible state of the primary's TCP stack. TCP stack replication in FT-Linux implements the design of Alvisi et al. [20].

FT-Linux intercepts calls below Linux's TCP stack and above the TCP stack, at the system call interface. We use Netfilter hooks to intercept packets coming from the network just before they enter the TCP layer of the Linux TCP/IP stack, and from the TCP stack just before they reach the IP layer of the Linux TCP/IP stack.

Non-determinism in the original TCP stack is resolved when its effect on outputs (both network output, up calls, and return values to the user-space layer) are known. For example, the TCP stack may return a non-predictable number of bytes to an application reading from a socket. However, before returning to the application, the primary will update the logical state on the replica to reflect the number of bytes returned. In this way, we resolve non-determinism without requiring the exposure of internal non-determinism at the interface of the TCP stack. Similarly, the size of a network packet sent by the primary TCP stack may not be predictable. Therefore, before sending a packet, the primary will inform the replicas of the size of the packet. In turn, the logical state of the TCP stack on the replica is adjusted according to the information received from the primary, resolving non-determinism.

Special care and engineering effort is needed to handle the many possible states of a TCP connection. We modified the Linux network stack in order to handle each TCP state.

3.5. Ensuring Output Stability

To ensure that transparent failover is possible at any time, the primary replica must wait for an output packet to be stable before sending it through the network, i.e., it waits for all TCP-stack state updates and all user-space non-deterministic actions that caused the output to be acknowledged by the secondary replica. This is the traditional "output commit" problem [21].

If the primary replica waits for its output to be known to the secondary before releasing it (by waiting for an acknowledgement), then, should the primary crash after its output, the secondary replica will be able to reach the state the primary was in just before the output, produce the same output, and continue execution transparently to the application. Note that the primary does not need to wait for the secondary's live copy of the application to have reached the same execution point, but only to have received it for subsequent live replay. As much as possible, we execute the primary without waiting for the replica to have received synchronization information.

Due to our single-machine setting, this constraint can be relaxed at a small cost. If the primary does not wait for an acknowledgement from the secondary, there is a small chance that some of its messages will not have left its local memory caches when it is hit by a failure. If the failure is a memory failure, then the cache coherency protocol will continue functioning and the messages will be promptly received by the replica. Only if the failure disrupts cache coherency, will the messages be lost. We conjecture that this case will be rare.

3.6. Failure Detection

Failure detection in FT-Linux relies on a heart-beat mechanism: each replica periodically sends a heart-beat message to the other replica. If no heart-beat is received after a configurable timeout, the sender replica is considered to have failed by the other replica. In order to prevent a situation in which a replica that is deemed to have failed is just too slow, the other replica also sends an inter-processor interrupt that will forcibly halt it, if it has not already done so.

Depending on the hardware architecture, a range of additional mechanisms can be used to detect hardware failures. For example, Intel's Machine Check Architecture reports

hardware errors to the operating system. In the case of FT-Linux, those reports can be used to determine whether to halt the primary replica and trigger failover.

3.7. Failover

When the failure of the primary replica is detected, the system switches to a failover phase in which the secondary replica prepares to enter primary execution. All necessary kernel objects are created or brought to a state consistent with what the application's clients observed so far. Moreover, the primary's exclusive ownership of the hardware devices needed by replicated applications must be revoked and device ownership is transferred to the secondary replica. FT-Linux transfers devices ownership by re-loading the necessary drivers on the secondary replica. For drivers that do not hold application-visible state, such as Ethernet drivers, this works well. For other cases, the techniques of Swift et al. [22] could be used to replicate device-driver state similarly to how FT-Linux replicates the state of the TCP stack.

4. Experimental Evaluation

Software. We evaluated the performance of several popular Linux applications when running on FT-Linux. We compared the applications' performance on FT-Linux with that on unmodified Ubuntu Linux 12.04. Both OSes are based on Linux kernel 3.2.14 – the Linux version on which FT-Linux has been developed. We measured the overheads introduced by FT-Linux by means of characteristic parameters of the tested applications, which include PBZIP2, Mongoose (plus ApacheBench), an in-house developed HTTP-based file server, and wget.

Hardware. All the experiments were performed on a machine with four AMD Opteron 6376 processors with 16 cores each (64 cores in total) and 128 GB of RAM, split in 8 equally sized NUMA nodes. Unless otherwise stated, FT-Linux is configured to divide the machine into two symmetrical partitions of two processors (32 cores, or 4 NUMA nodes) and 64 GB of RAM each. When comparing with Ubuntu, Ubuntu is allocated the same resources as a single FT-Linux partition (e.g., 32 cores, 4 NUMAs, 64 GB).

4.1. Compute Performance

To measure the overhead caused by the inter-replica synchronization of Pthreads operations, we measured the performance of FT-Linux when compressing a file using the PBZIP2 file compressor. PBZIP2 is a multi-threaded program that spawns three types of threads. First, a producer thread reads the file to be compressed from the file system, divides it into blocks of equal size, and inserts them one by one into a shared queue. Second, a configurable number of worker threads repeatedly dequeue a block from the queue, compress it, and insert the compressed block into another shared (output) queue. Finally, a writer thread reads blocks from the output queue, puts them in order, and writes the resulting compressed file to the file system. The queues are protected by Pthreads locks, and the producer threads notify the consumer threads using a condition variable.

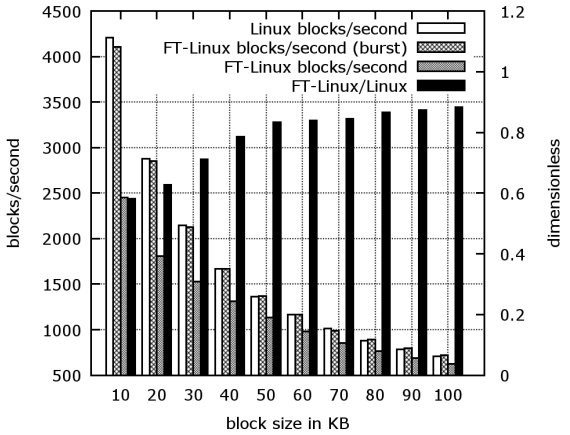

Figure 4. Compressing a file with PBZIP2, varying the block size.

We compressed a 1GB file using PBZIP2, using 32 worker threads on FT-Linux and on Ubuntu. Figure 4 shows the number of blocks compressed per second on Ubuntu versus on FT-Linux as a function of the block size used (the scale is on the left vertical axis). For FT-Linux, two quantities are shown: the peak throughput attainable in a short burst, and the throughput sustainable over a long period. The two are different because during a short burst, the primary replica only sends data to the secondary replica and does not have to wait for the secondary replica to process it. Over a long period of time, the primary replica must slow down to the pace of the secondary replica, which is slower; otherwise, data accumulates "in flight" and eventually exhausts the available buffers on the secondary replica. Figure 4 also shows the percentage of the performance of Ubuntu achieved by FT-Linux (scale on the right axis).

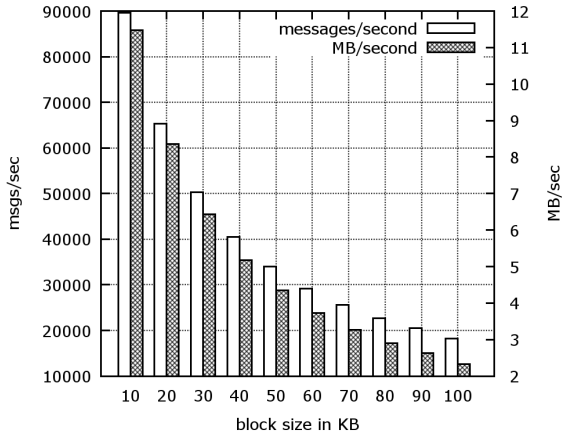

Figure 5. Inter-replication messages per second and bytes per second observed when running PBZIP2, as a function of the block size.

Figure 5 shows the traffic, in terms of the number of

inter-replica messages and the number of bytes, flowing through FT-Linux's shared-memory messaging layer as a function of the block size used. The inter-replica traffic increases super-linearly as the block size decreases because worker threads contend on the input and output queues and increasingly have to retry their dequeue and enqueue operations.

FT-Linux compares well to Ubuntu for block sizes between 50 and 100 KB: FT-Linux achieves close to 80% of the throughput of Ubuntu. At a block size of 50 KB, FT-Linux compresses 1113 blocks/s, and the inter-replica message rates reach around 34,000 messages per second for a byte rate of 4.3 MB/s.

Below 50KB, the peak throughput of FT-Linux continues to follow the throughput of Ubuntu closely, but the sustainable throughput drops steadily as the block size decreases. The throughput drops because the secondary replica cannot replay the execution of the primary replica fast enough. This is due to the Linux implementation of the `wake_up_process()` function. We rely on this function to wake up the next process waiting for a message, which can be waiting on an idle processor – this function can take up to tens of ms. Since inter-replica Pthreads synchronization is done serially (all Pthreads operations are serialized with a global lock), `wake_up_process()` is a bottleneck that limits the performance of the entire application. The inter-replica bandwidth is not a bottleneck because the core to memory transfer rates can easily reach several GB/s when we only transfer at 4.3 MB/s.

4.2. Network I/O Performance

Figures 6 and 7 show performance measurement of a Mongoose web server running 32 worker threads under a client load generated by the ApacheBench utility. The ApacheBench utility repeatedly requests a 10KB static web page (over TCP) using 100 parallel connections per second. The client machine is connected through a 1Gb network link to the Mongoose server running on FT-Linux. In this experiment, we artificially inserted a CPU loop that runs upon each request in the Mongoose server code, simulating an application performing a computation on each request. The Mongoose web server uses one listening thread which accepts connections from clients, and delegates connection processing to worker threads using a shared queue protected by a Pthreads lock and a condition variable. In this experiment, FT-Linux must maintain the state of the primary's TCP stack on the replica, and must coordinate thread scheduling among the replicas.

Figure 6 presents the performance of the Mongoose server on FT-Linux and Ubuntu as a function of the number of iterations that each CPU loop involves; for each increment of the value on the horizontal axis, the number of iterations is multiplied by two. We show the number of requests per second processed by Ubuntu, by FT-Linux in short bursts, and by FT-Linux at a sustained rate. Figure 7 presents the inter-replica traffic, in messages per second and bytes

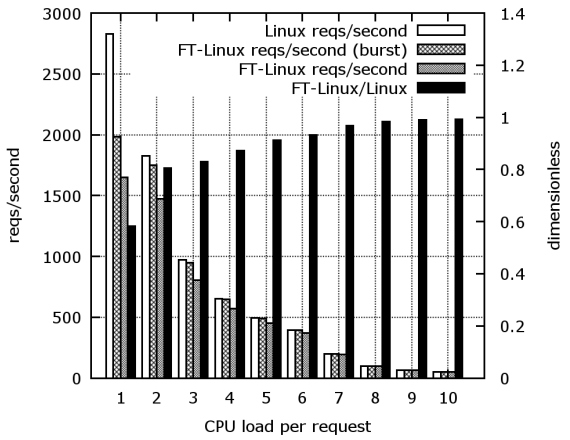

Figure 6. Performance serving a 10KB static web page with the Mongoose web server, with a varying CPU load per request (incrementing the CPU-load value by one corresponds to doubling the time spent by the CPU to process one request).

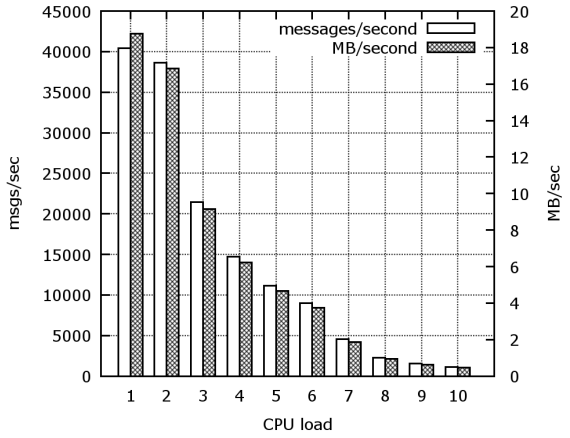

Figure 7. Inter-replica messages serving a 10KB static web page with the Mongoose web server, with a varying CPU load per request (incrementing the CPU-load value by one corresponds to doubling the time spent by the CPU to process one request).

per second, flowing through the shared-memory messaging layer.

Figure 6 shows that FT-Linux stays within 20% of the throughput of Ubuntu when the throughput is lower than roughly 1500 requests per second. At a higher number of requests per second, the performance of FT-Linux, when compared to Ubuntu, drops sharply. Unlike PBZIP2, the burst rate of FT-Linux is also affected by the increasing synchronization load, showing that network I/O synchronization is more costly than Pthreads schedule replication. Note that for all values of CPU load, the bandwidth remains lower than the network capacity (for a maximum of 900Kb/s).

The benchmark shows that FT-Linux performs well for CPU-bound applications, but suffers from a larger overhead, reaching only 60% of the performance of Ubuntu, under a high load of short requests.

4.3. Mixing Non-Replicated and Replicated Applications

FT-Linux allows replicated applications to run alongside non-replicated applications. To measure whether non-replicated applications affect the performance of replicated ones, we configured FT-Linux to create a 32-core primary partition alongside a single-core secondary partition. On the primary partition, we run a CPU-intensive non-replicated application that occupies all 32 CPUs at 100% when left running alone. Then, we started a Mongoose web server and used the ApacheBench utility to repeatedly request a 10KB static web page using 5 concurrent requests at a time. To compare with Ubuntu, we ran the same benchmark on 32 cores.

We observed that Ubuntu reaches a throughput of 760 requests per second while FT-Linux reached 700 requests per second, or 91% of the throughput of Ubuntu. Moreover, Ubuntu exhibits a latency of $1.3ms$ per request, while FT-Linux achieves a latency of $1.4ms$ per request, an 8% increase. This benchmark shows that the replicated Mongoose server on FT-Linux does not suffer significantly from the interference of non-replicated applications when compared to Ubuntu.

4.4. Failover

To evaluate the failover capabilities of FT-Linux, we measured the throughput obtained by downloading a 10GB file via a TCP connection, traversing a 1Gb/s Ethernet link using the wget utility. FT-Linux runs the in-house HTTP-based file server written in C. The server listens for incoming connections, and transfers the 10GB file when it receives one. We opted for a light-weight server to easily break down the overheads.

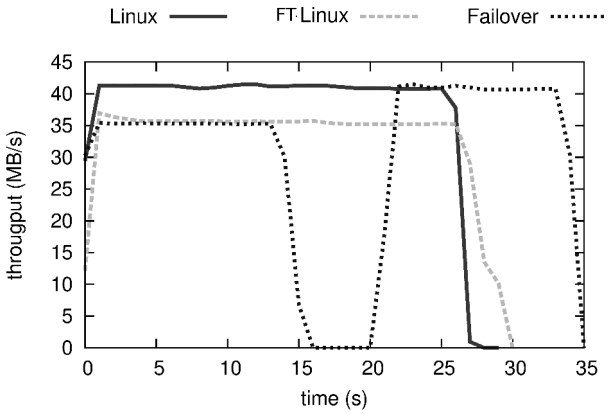

Figure 8. The throughput of FT-Linux in transferring a data file of 10GB between machines compared with Ubuntu (Linux) and FT-Linux with failures (Failover).

Figure 8 shows the throughput obtained in three scenarios on the server: using the stock Ubuntu installation, replicating the server with FT-Linux, and replicating the server with FT-Linux when the primary replica fails in the middle of the transfer. In this experiment, FT-Linux achieves a transfer rate up to 85% of the Ubuntu ones in the failure-free case. Although the TCP connection is not interrupted, the throughput drops to zero for roughly 5 seconds upon failure, before increasing again and reaching the rate of Ubuntu.

Our breakdown shows that 99% of this 5 seconds failover time is due to the time that the NIC driver takes to load on the secondary; this is consistent with what has been reported by previous work [22]. We left the optimization of this case as future work, as more engineering work on the driver itself can significantly reduce the latency.

5. Past and Related Work

Prior to FT-Linux, there exist several architectural solutions to mitigate faults on a multi-core system. Tandem Computer Systems [6] are multi-CPU computer systems designed with fault-tolerance capability. A process running on one CPU has an identical replica on another CPU. Such redundancy is called a "process pair." When one CPU fails, the paired process can take over and the system will automatically redirect I/O data to the replica. However, process pairs on Tandem Computer Systems need to be written with their checkpoint APIs to enable the fault-tolerance feature.

Another modern solution is the Dynamic Processor Sparing [1] technique, which can be seen on IBM Power 8-based servers. With this technique, running processors will automatically checkpoint at a fine granularity. As soon as an error is detected, the pre-configured spare processors can resume the system from the checkpoint.

While [1], [6] require either proprietary hardware or specially designed software or both, FT-Linux is a pure software solution that is able to run on commodity multicore systems, and supports existing POSIX applications without any code modifications.

Another pure software-based solution is Core Surprise Removal (or CSR) [5], which is also based on Linux. In CSR, when a core fails, tasks running on the failed core are migrated to other working ones. While CSR is targeted toward core failures, FT-Linux's full-stack replication approach is capable of mitigating other faults such as memory and device failures.

To the best of our knowledge, FT-Linux is the first system that utilizes a multi-kernel OS to achieve fault-tolerance and recovery. Hive [23], [24] and Cellular Disco [25] pioneered the field of multi-OS kernel setups in shared memory, partitioning or dynamically sharing hardware resources and confining hardware and software errors to the affected kernels.

The Barrelfish [26] operating system implements a new OS architecture in which independent kernels run on each CPU of a multi-core system and communicate by message passing. Popcorn Linux [9], FT-Linux's ancestor, was originally designed to enable unmodified Linux applications to transparently execute on heterogeneous architectures to optimize performance and energy consumption.

Replicating multi-threaded application is a major challenge in FT-Linux. Our approach is inspired by a number of existing solutions. Rex [12], Eve [27], and Crane [28] are recent systems that are designed for replicating multi-threaded applications on a multi-machine setup. They all utilize state machine replication to some degree, to maintain consistent states of multi-threaded applications across replicas. While Crane maintains a global deterministic execution order across all the replicas, Rex and Eve allow the applications to have arbitrary thread inter-leavings on the primary machine, and enforce the same execution order on other replicas.

For recovering the TCP stack for Linux, FT-TCP [10] proposes a TCP-stack replication scheme for two replicas, and demonstrates how to engineer fault-tolerant applications around it. Each application is replicated in an ad-hoc manner, and system calls are synchronized as needed. FT-Linux uses similar techniques to replicate the Linux TCP stack, but offers a more generic approach to user-space application replication.

6. Limitations and Future Work

Similarly to lock-step processors, a limitation of the proposed approach is that intra-machine replication on commodity hardware may rely on non redundant hardware components, such as the motherboard, a network card, and a disk drive – we assume they do not fail. However, we believe that fault tolerance support can be added cheaply to such components. Note that multiple network cards can be plugged in a single machine, but the system will require external hardware to replicate the I/O

FT-Linux offers fault-tolerance guarantees similar to dual lock-step processors. Another hardware replication technique is Triple Modular Redundancy, in which all hardware is triplicated and a voting mechanism is used to exclude inputs or outputs from faulty components. Currently, FT-Linux does not support more than two replicas; however, it could be extended to support a configurable number of replicas. FT-Linux's ancestor, Popcorn Linux and its messaging layer already support a configurable number of replicas, and replica synchronization could be achieved with few modifications of the current prototype by overlaying a consensus protocol over the inter-replica messaging layer: David et al. [7] describe how to implement high-performance consensus based on the Paxos algorithm in a shared memory setting.

Another limitation of FT-Linux's architecture is that it does not work with devices whose memory space is directly mapped into the application itself, such as kernel-bypass drivers. With FT-Linux, we target traditional monolithic operating systems in which all resources are centrally managed by the OS itself.

Some replication systems ensure Byzantine Fault Tolerance (BFT) [29] and can therefore continue operating even when some replicas are controlled by a malicious adversary. The design of FT-Linux precludes BFT because some hardware I/O components are not replicated and constitute single points of failure, like the NIC, and because a malicious kernel could make arbitrary writes in the address space of other kernels. However, new hardware mechanisms such as Intel's Software Guard Extension (SGX) may be used to enforce replica isolation. Leveraging SGX, FT-Linux could support BFT if each I/O device is also replicated.

FT-Linux's current implementation does not support applications with non-trivial use of a file system (e.g., a database backed by a file system). However, recent work on the specification of POSIX file-systems [30] presents strong evidence that POSIX file-systems are deterministic except for the number of bytes returned by a read. This indicates that file system replication could be done using SMR relatively easily even when the file system is implemented in kernel. Another approach would be to run a user-space file system as a replicated application.

Device drivers that hold application-visible state could be replicated by recording their execution and restarting them on the secondary replica (after the primary's failure) using techniques developed by Swift et al. [22].

7. Conclusions

We investigate whether fault-tolerant full software-stack transparent replication in a single machine is a practical middle ground between multi-machine live replication, as implemented in FT-TCP [10], Rex [12], or Crane [28], and hardware fault tolerance, as implemented in lock-step processors.

To understand this space, we built a prototype operating system based on Linux, called FT-Linux, which partitions the hardware resources of a single multi-core machine into two partitions and replicates unmodified Pthreads applications using Primary-Backup replication. To ensure fully transparent replication of network applications, FT-Linux also transparently replicates the Linux TCP stack. The architecture of FT-Linux is inspired by FT-TCP [10] and TFT [11]. Our performance evaluation shows that intra-machine replication incurs a moderate overhead on a selection of Linux applications. Compared to multi-machine replication, intra-machine replication can be applied to embedded systems where space constraints preclude multi-machine replication, and can improve the fault-tolerance of systems with spare capacity at a low cost.

Acknowledgments

This work was done as part of Yuzhong Wen's MS thesis at ECE, Virginia Tech. The authors thank the anonymous reviewers and Prof. Haibo Chen for comments on an early version of this paper. This work is supported in part by grants received by Virginia Tech including that from AFOSR (grants FA9550-14-1-0163 and FA9550-16-1-0371) and ONR (grants N00014-13-1-0317 and N00014-16-1-2711).

References

[1] J. Cahill, T. Nguyen, M. Vega, D. Baska, D. Szerdi, H. Pross, R. Arroyo, H. Nguyen, M. Mueller, D. Henderson *et al.*, "Ibm power systems built with the power8 architecture and processors," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 4–1, 2015.

[2] V. K. Garg, "Implementing fault-tolerant services using state machines: Beyond replication," in *International Symposium on Distributed Computing*. Springer, 2010, pp. 450–464.

[3] G. Gogniat, T. Wolf, W. Burleson, J.-P. Diguet, L. Bossuet, and R. Vaslin, "Reconfigurable hardware for high-security/high-performance embedded systems: the safes perspective," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 2, pp. 144–155, 2008.

[4] D. Jewett, "Integrity s2: A fault-tolerant unix platform," in *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium*. IEEE, 1991, pp. 512–519.

[5] N. Shalev, E. Harpaz, H. Porat, I. Keidar, and Y. Weinsberg, "Csr: Core surprise removal in commodity operating systems," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2016, pp. 773–787.

[6] J. Bartlett, J. Gray, and B. Horst, "Fault tolerance in tandem computer systems," in *The Evolution of Fault-Tolerant Computing*. Springer, 1987, pp. 55–76.

[7] T. David, R. Guerraoui, and M. Yabandeh, "Consensus inside," in *Middleware*, L. Réveillère, L. Cherkasova, and F. Taïani, Eds. ACM, 2014, pp. 145–156.

[8] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian, "The end of slow networks: it's time for a redesign," *Proceedings of the VLDB Endowment*, vol. 9, no. 7, pp. 528–539, 2016.

[9] A. Barbalace, B. Ravindran, and D. Katz, "Popcorn: a replicated-kernel os based on linux," in *Proceedings of the Linux Symposium, Ottawa, Canada*, 2014.

[10] D. Zagorodnov, K. Marzullo, L. Alvisi, and T. C. Bressoud, "Engineering fault-tolerant TCP/IP servers using FT-TCP," in *DSN*. IEEE Computer Society, 2003, pp. 393–402.

[11] T. C. Bressoud, "TFT: A software system for application-transparent fault tolerance," in *FTCS*. IEEE Computer Society, 1998, pp. 128–137.

[12] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang, "Rex: replication at the speed of multi-core," in *EuroSys*, D. C. A. Bulterman, H. Bos, A. I. T. Rowstron, and P. Druschel, Eds. ACM, 2014, pp. 11:1–11:14.

[13] L. Alvisi, T. C. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov, "Wrapping server-side tcp to mask connection failures," in *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 1. IEEE, 2001, pp. 329–337.

[14] A. B. Lim and E. D Heaton, "Platform-level error handling strategies for intel systems," http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/platform-level-error-strategies-paper.pdf.

[15] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field," in *DSN*. IEEE Computer Society, 2015, pp. 415–426.

[16] B. Fitzpatrick, "Memcached - a distributed memory object caching system," http://www.memcached.org.

[17] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proceedings of ASPLOS XVII*, pp. 37–48.

[18] A. Kleen, "Ongoing evolution of linux x86 machine check handling," http://www.halobates.de/mce-lc09-2.pdf.

[19] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," *TOCS*, vol. 14, no. 1, pp. 80–107, 1996.

[20] D. Zagorodnov, K. Marzullo, L. Alvisi, and T. C. Bressoud, "Practical and low-overhead masking of failures of TCP-based servers," *TOCS*, vol. 27, no. 2, 2009.

[21] E. N. Elnozahy and W. Zwaenepoel, "Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit," *IEEE Transactions on Computers*, vol. 100, no. LABOS-ARTICLE-2005-013, pp. 526–531, 1992.

[22] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, "Recovering device drivers," *ACM Transactions on Computer Systems (TOCS)*, vol. 24, no. 4, pp. 333–360, 2006.

[23] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta, "Hive: Fault containment for shared-memory multiprocessors," in *SOSP*, M. B. Jones, Ed. ACM, 1995, pp. 12–25.

[24] M. Rosenblum, J. Chapin, D. Teodosiu, S. Devine, T. Lahiri, and A. Gupta, "Implementing efficient fault containment for multiprocessors: Confining faults in a shared-memory multiprocessor environment," *CACM*, vol. 39, no. 9, pp. 52–61, 1996.

[25] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum, "Cellular disco: resource management using virtual clusters on shared-memory multiprocessors," *TOCS*, vol. 18, no. 3, pp. 229–262, 2000.

[26] A. Baumann, P. Barham, P. Dagand, T. L. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new OS architecture for scalable multicore systems," in *SOSP*, J. N. Matthews and T. E. Anderson, Eds. ACM, 2009, pp. 29–44.

[27] M. Kapritsos, Y. Wang, V. Quéma, A. Clement, L. Alvisi, and M. Dahlin, "All about eve: Execute-verify replication for multi-core servers," in *OSDI*, C. Thekkath and A. Vahdat, Eds. USENIX Association, 2012, pp. 237–250.

[28] H. Cui, R. Gu, C. Liu, T. Chen, and J. Yang, "Paxos made transparent," in *SOSP*, E. L. Miller and S. Hand, Eds. ACM, 2015, pp. 105–120.

[29] L. Lamport, R. E. Shostak, and M. C. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982.

[30] T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell, "Sibylfs: formal specification and oracle-based testing for POSIX and real-world file systems," in *SOSP*, E. L. Miller and S. Hand, Eds. ACM, 2015, pp. 38–53.