CSR: Core Surprise Removal in Commodity Operating Systems

Noam Shalev[†] Eran Harpaz[†] Hagar Porat[†] Idit Keidar[†] Yaron Weinsberg[*]

Technion, Israel Institute of Technology[†]
IBM Research, Israel[*]

{noams,seharpaz,hagarp}@campus.technion.ac.il idish@ee.technion.ac.il yaron@il.ibm.com

Abstract

One of the adverse effects of shrinking transistor sizes is that processors have become increasingly prone to hardware faults. At the same time, the number of cores per die rises. Consequently, core failures can no longer be ruled out, and future operating systems for many-core machines will have to incorporate fault tolerance mechanisms.

We present CSR, a strategy for recovery from unexpected permanent processor faults in commodity operating systems. Our approach overcomes surprise removal of faulty cores, and also tolerates cascading core failures. When a core fails in user mode, CSR terminates the process executing on that core and migrates the remaining processes in its run-queue to other cores. We further show how hardware transactional memory may be used to overcome failures in critical kernel code. Our solution is scalable, incurs low overhead, and is designed to integrate into modern operating systems. We have implemented it in the Linux kernel, using Haswell's Transactional Synchronization Extension, and tested it on a real system.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design

General Terms Design, Reliability

Keywords Operating Systems, Transactional Memory, Reliability, Hotplug, Core Surprise Removal, CSR.

1. Introduction

In today's economies of scale, which rely upon adding more and more servers with cheaper hardware, the observation that failures are the norm rather than the exception [11, 24] has become a guiding light for system designers. Nowadays, constant monitoring, fault tolerance, and automatic recovery are an integral part of any large cloud or data-center system. However, these are mostly focused on machine failures rather than low-level hardware failures such as processor faults.

At the same time, the number of cores per chip constantly increases. Modern consumer machines already contain 32 cores, data center servers include more than 60 cores per machine, while tera-devices with hundreds or even a thousand fault-prone cores are the subject of active research [25, 70]. This was made possible due to constant shrinking in transistor size and voltage supply. However, with hardware shrinking, the probability of physical flaws on the chip significantly rises [8, 16], and chances for processor faults become substantial [66]. With many tens or even hundreds of cores per chip, core failures can no longer be ruled out. Recent studies show that, even on consumer machines, hardware faults are not rare [52], and memory errors are currently dominated by hard errors, rather than soft-errors [63]. We therefore believe that, with technology advances, tolerating failures of individual cores shall become inevitable.

Current operating systems, including Linux and Windows, crash in the face of any permanent core fault and most chip-originated soft errors. As we explain later, the reason for the crash lies in the various kernel mechanisms that require the collaboration of the faulty core. Thus, the failure of a single core, out of hundreds in the foreseeable future, brings down the entire system. Cloud systems, for example, usually consolidate multiple virtual machines on a single server in order to improve its utilization [7, 39, 43, 61, 65]. In such settings, the failure of a single core crashes all the VMs running on the server.

Our goal in this work is to fortify existing operating systems against unexpected core failures, and in particular, allow processes – and VMs – on other cores to operate without interruption. We further elaborate on our goals and design considerations in Section 2.

In Section 3, we present *CSR*, a strategy for overcoming *Core Surprise Removal* in commodity operating systems. CSR is designed for multi-core architectures with reliable shared memory, and incurs virtually no overhead when the

ASPLOS '16, April 2–6, 2016, Atlanta, Georgia, USA.
Copyright © 2016 ACM 978-1-4503-4091-5/16/04...$15.00.
http://dx.doi.org/10.1145/2872362.2872369

system is correct. Its primary objective is to keep the system alive and running after a core crash, while terminating at most the one user program that had been running on the faulty core. Recovery from that user program's failure can be handled, e.g., using checkpointing mechanisms [20], by application developers, and is out of this work's scope.

CSR's basic fault recovery function overcomes all core failures that happen when the faulty core is either in user-mode or executing non-critical kernel code. In order to cope with failures inside kernel critical sections, we propose in Section 4 a complementary technique based on *lock elision* using *Hardware Transactional Memory (HTM)* [31, 57], which is already incorporated in modern commodity processors [15, 77] and HPC systems [71]. In addition, CSR has a modular design, and it employs deferred execution of the recovery process in order to minimize the vulnerability to cascading failure scenarios, as discussed in Section 5.

In Section 6 we provide a proof-of-concept implementation of CSR in version 3.14.1 of the Linux kernel. We further exemplify the use of Haswell's HTM support[1] for recovery from failures in critical kernel code. HTM prevents critical sections' intermediate values from being written to shared memory, thus avoiding inconsistent states following a crash.

In Section 7, we evaluate CSR via three different experiments. First, we use a virtualized environment to inject thousands of permanent failures at random times. Since our emulated VM does not support HTM, we run CSR in this experiment without using HTM protection for critical kernel sections. We show that, while an unmodified kernel persistently fails, CSR successfully recovers from all failures occurring in user or idle mode, and in roughly 70% of the faults injected during execution of kernel code.

Next, we inject permanent core failures on real systems, during the execution of a hundred different critical and non-critical kernel functions. In these experiments, we use CSR with lock elision, and it successfully recovers from 100% of the faults. Finally, we build a fault-resistant cloud setting by installing CSR on the host OS of a 64-core server over which multiple virtual machines run unmodified Ubuntu. We show that following a core fault, only one VM crashes, while the others resume normal execution.

To conclude, our contributions in this work are:

1. We present CSR, a strategy for overcoming permanent core failures.
2. We propose using HTM for tolerating processor faults that happen during execution of critical kernel code.
3. We implement CSR in Linux, as well as a proof of concept for using HTM for recovery.
4. We evaluate CSR by injecting numerous faults on real and virtualized environments.

[1] The recently found erratum in Intel-TSX instructions did not affect our experiments. Our work comprises a prototype, and the concept of using HTM for our purposes shall remain applicable in future technologies.

2. Design Considerations

We design CSR to overcome core failures, whereby a core stops working without advance notice. Our goal is to take a system from an arbitrary state induced by a failure to a *recovered* state, where the following holds: (1) Interrupts are routed to online cores only. (2) All threads, as well as all types of delayed kernel tasks, are affined to online cores only. (3) All kernel services and subsystems operate using only online cores.

In this section we first argue that core failures merit a different treatment from CPU Hot-Plugging (Section 2.1), and then detail our failure model (Section 2.2) and the extent to which we can overcome failures in kernel code (Section 2.3).

2.1 Why Not CPU Hotplug?

Modern commodity operating systems usually include a *CPU Hotplug* [50, 51] mechanism, which allows one to dynamically plug or unplug a core. Ideally, we would like to simply unplug a faulty core once a failure is detected. However, there is a profound difference between voluntary CPU unplugging and on-line removal of a faulty core: CPU unplugging can use the cooperation of the victim core, thus enabling a completely supervised removal process, including choosing the correct moment to commence the removal, asking the victim core to perform some local tasks, and reading the values stored in its registers. In contrast, a failure can happen any time, depriving the crashed core of the ability to communicate, and leaving its buffers' and registers' contents unknown. The latter suggests that it is impossible to know what the core was doing at the moment of the failure.

2.2 Fault Model

We consider cores prone to *permanent faults* [55] – irreversible physical faults that cause consistent failure. This captures cases such as permanently open or shortcut transistors, slow components that cause timing violations, or a partially burned-out chip [56].

CSR ensures recovery from failures that happen while the faulty core executes in user-mode; other cores can be either in kernel or user mode. Failures in kernel code are partially handled, as we discuss in Section 2.3. CSR is only concerned with the system's survival, whereas application-level recovery is out of scope.

When dealing with core failures, one has to define their scope, namely, which hardware components are affected by the failure. Some designs of future many-core machines [25, 72, 73] assume that the entire internal state of a faulty core (including registers and buffers) may be flushed upon failure. We do not require this in this paper, but rather allow a core's internal state (buffers and registers) to remain unavailable following a crash. We do, however, assume that core malfunction does not destroy its cache. Hence, following a core's failure, it is possible to flush its private cache.

We use such a flush in order to ensure kernel data consistency in the following scenario: a kernel thread updates some OS data structure; next, the core switches to user code, but the changes it had made to kernel data have propagated from its local cache to the shared memory only partially; then the core fails. If one wishes to forgo triggering a cache flush following failure without sacrificing consistency, other solutions are possible. For example, one can configure kernel pages as non-cacheable, or flush the cache in kernel-to-user mode transitions. Note that fault-tolerant cache technologies are currently emerging [33, 42], and it is not unlikely that future caches will provide fault-tolerance features.

For detecting failures and triggering the recovery process, we assume the existence of a reliable *Failure Detection Unit (FDU)*. This can be implemented in hardware, using heartbeats and Machine Check Architecture [2, 32, 37], as suggested in previous works [21, 72, 73]. Upon fault detection, the FDU (1) halts the faulty core, thus preventing it from corrupting shared memory; (2) triggers a flush of its private cache; and (3) reports the error to the OS. Note that a hardware-based FDU implementation can easily provide these requirements. In this work, we simulate the FDU in software using heartbeats (see Section 6.1).

Given such an FDU, we design our core surprise removal strategy to cope with a *fail-stop* [62] model. Namely, a faulty core simply stops executing from some point onward.

CSR is designed for operating systems running on multicore architectures with reliable shared memory. Memory failures are typically addressed using other, complementary, mechanisms, such as error-correcting codes and relocating data [13, 60, 76]. By building upon reliable shared memory, we forgo the need for checkpointing.

2.3 Failures in Kernel Code

Unlike a failure in user-mode, certain failures in kernel-mode might be impossible to recover from by simply terminating the running thread. For example, failure during execution of a critical section might lead to data inconsistency. Failures in kernel code also include *cascading failure* scenarios, whereby the core that executes the recovery procedure crashes before completing the recovery process.

We address kernel-mode failures in two complementary ways: First, we design our algorithm to withstand cascading failures, while striving to minimize the time interval during which a cascading failure may interfere with an ongoing core removal. To this end, we use a strategy of queueing work for later execution. This allows us to migrate the recovery work to other cores in case the core that executes the recovery also fails.

Second, we propose the use of HTM for executing kernel critical sections. This prevents propagation of partial updates resulting from unfinished execution of critical sections to the shared memory. Recall that upon failure detection, the FDU triggers a flush of the faulty core's cache; here, we assume that the system does not flush uncommitted values.

We note that our proposed solution to failures in kernel code is a best-effort one. It guarantees recovery from failures during execution of code that does not access hardware different from RAM, and therefore can be protected with a transaction. Moreover, there are instructions that cannot be executed transactionally, such as a TLB flush, accessing a control register, an IO operation, etc. [36]. We cannot guarantee recovery from failures during execution of code sections containing such instructions.

3. Core Surprise Removal

The current section presents the heart of CSR's recovery approach, and focuses on the common case – failure during execution of user-code or non-critical kernel code. The next section addresses failures during critical kernel code, and in Section 5 we discuss cascading failures.

3.1 Background: OS Mechanisms Used

Our solution utilizes a few common facilities that exist in modern operating systems. We begin with some background and discuss the relevant aspects of these kernel mechanisms.

3.1.1 Deferrable Functions

Operating systems usually support multiple types of delayed tasks for performing work with different urgencies and execution contexts. For readability, we use the Linux terminology for the different delayed tasks types. CSR employs two types of delayed kernel tasks:

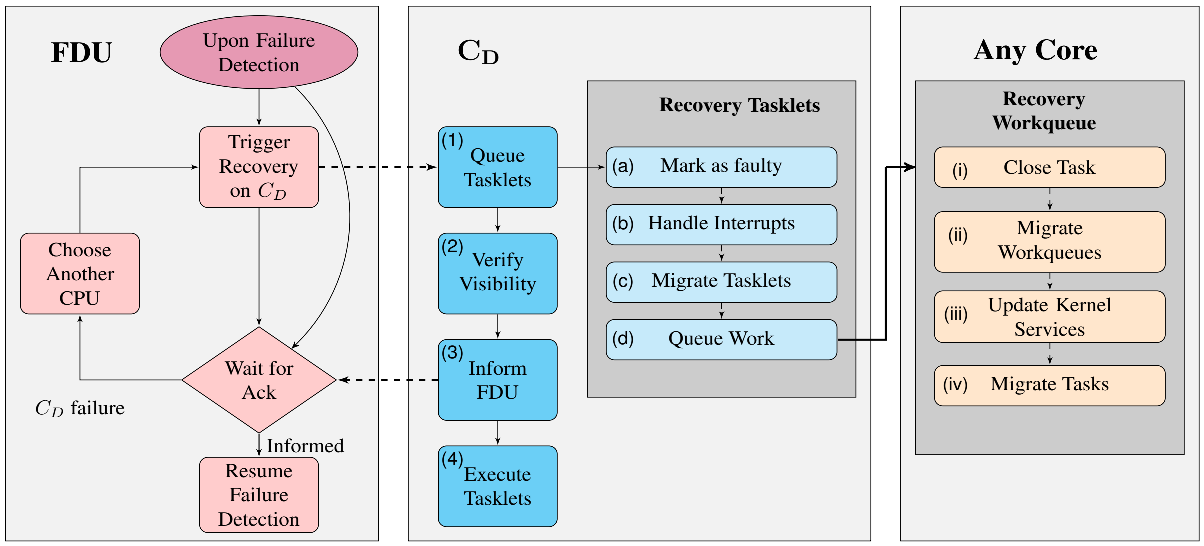
Tasklets [47] (Deferred Procedure Calls in Windows), which have three important properties: (1) They always run in interrupt context, and as a consequence, are unable to sleep or block. (2) A tasklet always executes on the core that schedules it. (3) The kernel services all pending tasklets immediately after handling all pending hardware interrupts.

Work-queues [47] (Asynchronous Procedure Calls in Windows), which, in contrast to tasklets, are executed by kernel threads in user context, and therefore are allowed to sleep or block. Furthermore, a work-queue may be bound to a specific processor, or may be unbound, in which case it can be executed by any core.

3.1.2 CPU States and Hotplug

Operating systems manage the state of all cores. For example, Linux uses bitmaps, dubbed *CPU masks*, for each possible state. Each CPU mask represents one state, and contains one bit per core. These masks are accessed globally and used by the kernel in various cases, such as for iterating over per-core data structures.

Linux, for example, defines, among others, an *online* state and an *offline* state. An offline core is not available for scheduling and does not receive or handle interrupts; it is in deep sleep mode, and consumes low power. A core's state can be dynamically changed between online and offline by using the CPU Hotplug mechanism. The CPU Hotplug



(a) Failure Detection Unit flow. (b) Recovery procedure on recovery core C_D. (c) Delayed tasks, execute on any core.

Figure 1: Recovery flow chart. Dashed lines represent message passing; solid lines represent flow.

feature has various uses [4, 51]. However, as explained in Section 2.1, CPU Hotplug cannot be used for disabling a faulty core, as it does not consider any reliability issues.

3.2 CSR Data Structures

We augment the kernel with two new data structures. First, we define a new processor state, *faulty*, and a corresponding global CPU mask, *faulty_mask*. This mask is the first to be updated upon processor failure, and, as its name suggests, indicates which cores in the system are in a faulty state. The faulty state serves two purposes at two different time intervals:

- While the system is not in a recovered state, namely, following a failure and before CSR's removal procedure has completed, the corresponding bit in the *faulty_mask* indicates to kernel code that hasn't yet adjusted to the failure to treat this core as unavailable for scheduling, migration, updating its data structures, etc.
- After the removal procedure finishes, a core marked as faulty is treated by the kernel as offline, with the exception that it is not permitted to come back online.

Second, we add a new globally accessed work-queue, *recovery-workqueue (RWQ)*. This work-queue is not bound to any processor, namely, different processors can pull its work items and execute them. We use RWQ to defer execution of certain recovery functions to a later time and a different execution context, in order to shorten the time interval during which a cascading failure may require re-execution of the recovery process. Functions queued to this work-queue are less urgent or might need to sleep, and so they are not queued as tasklets.

3.3 Failure Detection Unit

The FDU's operation is shown in Figure 1(a). When a failure is detected, the FDU invokes CSR's *recovery procedure* on a designated non-faulty online core, denoted C_D, which in turn handles the crash. The selection of C_D prefers cores that reside in a different socket in order to reduce the probability for a cascading failure. The ensuing flow in C_D, depicted in Figure 1(b), is described is Section 3.4 below.

After invoking the recovery process on the designated core, the FDU waits for an ack. Meanwhile, if the FDU detects a cascading failure of C_D, it invokes the recovery algorithm again, on another processor, for handling both the first failure and the new one. Once the FDU receives an ack from the designated core, it considers the recovery process done, and resumes normal operation with the faulty core, C_F, removed from the available processors list.

Note that this ack does not function as a keep-alive message. Its purpose is to notify the FDU that the recovery process has reached a point from which its completion is guaranteed; in case of a cascading failure before that point, the FDU cannot know what stages C_D completed before the failure.

3.4 Recovery Procedure

3.4.1 High Level Operation

C_D begins the recovery procedure, as presented in Figure 1(b). C_D's role consists of four primary steps: (1) First, it enqueues four new tasklets for repairing the system's state. These include Hotplug-like operations of modifying CPU masks, resetting interrupt affinities, and migrating tasklets from the removed core, as well as recovery-specific actions that address the surprise nature of the removal. The tasklets are only queued at this point, and are not executed yet. Queuing the tasklets rather than executing them reduces the time interval during which a cascading failure of C_D causes re-execution of the recovery process. (2) Next, C_D takes actions that assure the visibility of the tasklets to the rest of the cores (e.g., flush its write buffer). Once they are visible, other cores can steal them, and so the recovery operation is guaranteed to complete. (3) At this point, C_D sends an ack to the FDU. (4) Subsequently, C_D naturally turns to execute the tasklets found in its tasklets queue in FIFO order, as part of the kernel's normal behavior.

3.4.2 Recovery Work

Recovery Tasklets. The actual core removal process begins with the execution of four tasklets (depicted in the grey box in Figure 1(b)), which we subsequently call *recovery tasklets*. They perform the following:

(a) The first tasklet marks C_F's state as faulty in the corresponding CPU mask. Once a core is marked as faulty, kernel services will treat it as if it is out of the online map. Namely, no tasks will be scheduled or migrated to its run queue, iterations on various CPU masks will skip it, etc. The longer this step's execution gets postponed, the further the system might diverge from a recovered state, e.g., new tasks might get attached to C_F's run-queue.

(b) The next tasklet deals with interrupts. Modern operating systems use SMP IRQ affinity to assign interrupts to specific cores – this means that interrupts that were affined to the faulty core might have been lost. Therefore, to minimize the time interval during which interrupts can be lost, we reset their affinities early in the recovery process.

Next, we deal with interrupts that may have been lost. The loss of an interrupt can cause errors in software components that depend on its arrival. For example, a lost interrupt originating in the NIC may prevent network packets from being delivered to an application. Even worse, a lost Inter-Processor Interrupt (IPI) might prevent imperative kernel procedures from being executed and possibly crash the system. To recover from possible interrupt loss, we send spurious interrupts for all interrupt types that were previously routed to C_F (a technique that was employed in previous Linux versions). In addition, we check if any of the pending functions in C_F's IPI queue need to be migrated, and migrate those that do.

(c) The third tasklet migrates tasklets that are attached to C_F. This has to be performed urgently for two main reasons: first, tasklets embody high priority kernel work that is important to system maintenance and functionality. The second reason lies in our support for cascading failures, as we explain in Section 5 below.

(d) The last tasklet queues additional work that is essential to the recovery process but cannot run in interrupt context, or can endure a short delay in its execution. We divide this work into a number of functions, which we queue to our recovery-workqueue. This work will be executed later by dedicated kernel threads. Most of the work items queued at this step are needed in every operating system, but some of them are OS-dependent.

Queued Work. The following work-items are queued:

(i) Close the running task. We close the task that was running on C_F at the moment of the failure and free its resources. Since we are unable to communicate with C_F, significant data such as the instruction pointer and the register file content is lost. Therefore, we cannot recover the running application's state. It is possible to recover from such failures using a checkpointing mechanism, which could be done at application level.

(ii) Migrate work-queues affined to C_F. As in the tasklets case, we migrate work-items from the work-queues affined to C_F for later execution on other cores.

(iii) Update kernel services. This step is OS-dependent. Here, we update kernel services that might be affected by the sudden departure of C_F from the online map, such as performance events, synchronization services, and memory allocation.

(iv) Migrate C_F's tasks. User processes and kernel threads that were attached to the run-queue of C_F at the moment of the failure will usually still be there (unless the load balancer moved some of them); these must be migrated to other run-queues. The target run-queues for migrated tasks can be chosen in a variety of ways, but this choice is inconsequential as it only has a short-term effect – until the load-balancer kicks in. For the sake of simplicity, CSR moves these threads to the run-queue of the lowest-id correct core, leaving it up to the load-balancing mechanism to correct the overload that might be temporarily formed at the target core.

As in a regular CPU-unplug process, the migration of the run-queue is performed at the end of the removal process, as it does not affect system correctness. Its delay only blocks the execution of the tasks in C_F's run-queue until the end of the recovery process.

It is worth pointing out that CSR's recovery process subsumes the set of recovery actions performed by CPU-Hotplug. Specifically, the following steps are common, at least partially, to both CSR and CPU-Hotplug: (b), (c), (ii), (iii), and (iv). Nevertheless, tasks that perform analogous logical roles in both mechanisms differ in their implementation and order of execution, as CPU Hotplug exploits the cooperation of the unplugged core, and CSR cannot. After the execution of all the above, the system is again in a correct state, as all actions that would have been needed by a hot-unplug operation have been performed.

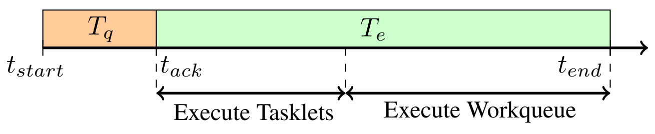


Figure 2: Recovery Periods.

Domain	T_q	T_e	
		Tasklets	Workqueue
In-Chip	0.9ms/1.1ms	1.8ms	1.5ms/1ms
Inter-Chip	0.9ms/1.1ms	1.8ms	2.3ms/2.1ms

Table 1: T_q and T_e measurements (Busy/Idle).

4. Failures in Kernel Code

Kernel code execution usually comprises a small fraction of the total system's runtime. However, a failure in kernel code might have severe consequences. For example, a failure during a migration operation may cause a task to be dequeued from the source queue without being enqueued at the target one. Moreover, a crash of a core holding a kernel lock may leave the system in an inconsistent, or even unrecoverable, state.

In order to prevent the implications such crashes may inflict, we propose to elide the locks [57] protecting kernel critical sections using HTM; HTM mechanisms cause changes to multiple memory locations to appear to be atomic, while providing isolation between parallel executions [31]. We assume that these properties still hold in the presence of failures, namely, the HTM prevents uncommitted changes from propagating, even when a failure occurs during a transaction. Thus, in case of a failure inside a critical section, no changes are written to shared memory, and the system remains in a consistent and recoverable state. Rossbach et al. [59] have proposed TxLinux, an operating system that exploits hardware transactional memory in kernel code for handling concurrency and improving performance. We, on the other hand, exploit HTM abilities in a similar context, but for the purpose of reliability.

Kernel developers devote effort in order to reduce contention among cores. To this end, each core has its own kernel data structures, and accessing another core's data is relatively rare. Therefore, transaction aborts due to data conflicts are likely to be infrequent. Nevertheless, since Intel Transactional Synchronization Extension (TSX) [36] is a *best effort* HTM, (namely, transactions are not guaranteed to commit because of various architectural reasons), and since kernel code executes many sensitive instructions, such as interrupt masking, interrupt sending, TLB and cache operations, which cause aborts, we expect transactions to abort occasionally. Furthermore, due to architectural reasons, some critical sections cannot be executed transactionally, e.g., a context-switch always causes a TLB flush – an operation that leads to abort. In such cases, our solution reverts to using locks. Using HTM in all remaining critical sections keeps the system fault-resistant a majority of the time. Though reliability is not assured while executing uncommon abort-prone sections as discussed above, the simplicity of this solution, and the very low overhead (or even performance gain) it incurs makes it appealing and easily applicable. We prototype this approach in Section 6, for a subset of OS critical sections, to illustrate its feasibility.

An alternative approach to handling failures during critical sections is by using Recovery Domains [44], an organizing principle presented by Lenharth et al., which uses logging and rollback for restoring system state upon failure. Unlike transactional memory systems, the Recovery Domains principle only focuses on recovery from failures and not on isolation between parallel executions. However, this does not constitute a problem, since concurrency in kernel code is already managed by locks. The most prominent advantage of this approach is the ability to cope with critical sections that cannot commit transactionally. However, this approach is likely to incur much higher overhead and complexity, a cost that might not justify the benefits. We do not further explore this path in this paper.

5. Cascading Failures

In this section we elaborate on CSR's tolerance to cascading failures. This relies on the principle that each recovery task is eventually executed by some core exactly once.

We use HTM to avoid inconsistent states resulting from failures during recovery tasks execution. To this end, we have divided the recovery procedure into four tasklets and four work items, each of which is small enough to run as a transaction. We execute each individual tasklet or work item as a separate atomic transaction. Next, we take steps to ensure that recovery items are executed exactly once.

To analyze different cascading failure scenarios, we examine a timeline of the recovery process, as shown in Figure 2. The time at which the failure of C_F is detected is denoted t_{start}, and t_{ack} denotes the time at which the FDU receives the ack from C_D. We denote by T_q the interval between t_{start} and t_{ack}, and by T_e the time between t_{ack} and the end of the recovery process.

Measurements of our implementation on a 64-core machine, presented in Table 1, show that T_q lasts about 1ms, in both busy and idle systems. T_e, can last a bit longer, depending on the system load as well as the chips on which C_D and C_F reside.

In case of a cascading failure, the FDU detects and triggers a recovery procedure on another processor, C'_D. We first discuss the case where C_D fails during T_e. Here, the FDU knows that the recovery tasklets are accessible by the rest of the cores, and if necessary, other cores can read, migrate, and execute those tasklets. T_e itself can be divided into two subperiods: one that consists of the execution of the recovery tasklets and a second where the work-items queued to RWQ are executed. In the latter case – since RWQ is not bound to any core, failure of C_D during that time does not affect the recovery process. In the former case, as part of step (c) of the recovery procedure, C'_D will take over and execute the recovery tasklets among the rest of C_D's tasklets. Thus, in either case, the recovery tasklets related to C_F's failure are executed, either by C_D before it fails, or by C'_D.

If the failure occurs during T_q, it is impossible to know which stages of CSR C_D has completed. Therefore, in case the FDU detects a failure of C_D without having received an ack from it, the FDU invokes a double recovery procedure on C'_D, to handle the failures of both C_F and C_D.

A failure during T_q can cause the recovery functions to appear more than once on the same queue. We give an example in Figure 3, where C_D completes stage (2) and crashes just before informing the FDU. C'_D then invokes a double recovery procedure (Figure 3(b)), during which it migrates C_D's tasklets. As a result, C'_D has the tasklets for handling C_F's failure twice. To prevent duplicate execution, we give the tasklets unique IDs, and mark completed tasklets in a globally accessed bitmap, as part of the transaction executing them. This bitmap is checked before each tasklet execution to ensure that it has not been executed before.

6. Implementation Issues

As a proof of concept, we implemented CSR in version 3.14.1 of the Linux kernel. Our implementation uses high-priority tasklets to queue the recovery tasklets, and allocates a new unbound workqueue to serve as the recovery-workqueue. We exploit, with some modifications, most of the functions that are used by CPU Hotplug. Eighty-one files of the Linux source were changed, for a total of about 4000 changed and new lines. We next explore some issues that arose during the work.

6.1 FDU and C_D

We implement the FDU using a periodic timer, which times out in predefined constant-length intervals and checks for heartbeats from all online cores (see Figure 4). Online cores send heartbeats to the FDU each timer interrupt. When the

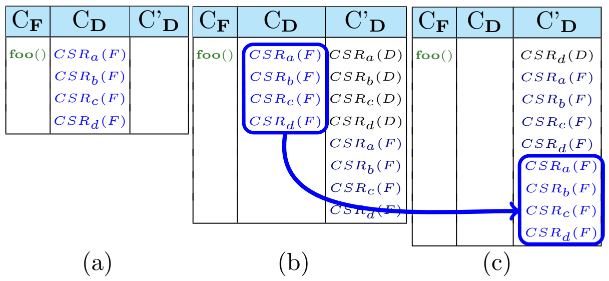


Figure 3: Duplicate queuing of recovery tasklets due to failure during T_q. Columns represent tasklet queues. $CSR_X(Y)$ represents recovery tasklet X for core Y.
(a) C_D queues recovery tasklets for C_F's failure.
(b) C_D fails, C'_D queues recovery tasklets for C_D and C_F.
(c) After executing $CSR_c(D)$, C'_D has duplicates in its queue.

FDU detects a core that has stopped sending heartbeats, it considers that core to be faulty, and initiates CSR on another processor, C_D, by sending it an IPI. To simulate the reliability of the FDU, we affine the FDU to core 0 solely and do not crash it during the experiments. This is done only for simplicity, as a reliable FDU can be implemented in hardware, even in a fault-prone environment [21, 72].

Upon receiving the IPI, C_D queues the recovery tasklets and verifies their visibility by using the WBINVD [34] instruction, which flushes its write buffer and private caches by invalidating them and performing a write back.

6.2 Setting a Faulty State

Linux's supervised core-unplug uses the *stop_machine()* mechanism for setting a core offline. The purpose of this mechanism is to prevent a core from going offline during execution of non-preemptible code on another core. This prevents, for example, changes to the online mask while another core iterates on it, and thus avoids inconsistent updates of the removed core's data. The *stop_machine()* function halts the execution of all online cores before the actual removal of the outgoing core from the online mask; thus, it postpones the actual removal of the victim core to a later time, at which no non-preemptible code is executed.

However, in crash failure cases, the use of *stop_machine()* is inadvisable. This is because the update of *faulty_mask* about the failure, as well as interrupt resetting, should happen fast. Deferring or preventing a core from crashing while another core executes non-preemptible code is, of course, impossible. Moreover, since the core will not get back online, consequences of inconsistent updates of its data are limited. We therefore refrain from using *stop_machine()*.

Note that previous works [26, 67] have also proposed to decouple *stop_machine()* from the CPU Hotplug path, for performance and design considerations.

6.3 Handling Lost Interrupts

For handling possible interrupt loss, following resetting the interrupt affinities, we send spurious interrupts for all the interrupt types that were previously routed to the faulty core. However, this does not cover cases of lost Inter-Processor Interrupts: The IPI queue on the faulty core might contain *migratable* callbacks, namely, callbacks that can execute on any core, and their execution is necessary. For example, in order to change the frequency of a chip, an IPI is sent to one

```
for_each_possible_cpu(cpu){
  if (!heartbeats_received(cpu) &&
          !cpu_is_faulty(cpu)){
    mark_cpu_as_faulty(cpu);
    Cd = choose_recovery_cpu();
    send_IPI(Cd,CSR,cpu);
  }
}
```

Figure 4: FDU periodic callback.

of the cores in that chip; if that core fails, the callback needs to be migrated. On the other hand, some functions passed by IPIs should execute exclusively on the core they were sent to (e.g., read MSRs or flush TLB). Only the code that sends the IPI can determine whether the callback should be migrated when its recipient fails.

We therefore provide each callback with a flag indicating whether it is migratable. The flag is set when the IPI is generated and checked during recovery. Linux provides four primitives for broadcasting IPIs, depending on whether the destination is (1) a particular core, (2) a subset of the cores, (3) all of the cores, (4) any of the cores in a subset (*anycast*). By examining the Linux source, we found that only functions that are sent by the anycast primitive are migratable. Thus, we changed the anycast implementation to queue these functions with flag set to true. By default, the other primitives queue functions as not migratable.

6.4 RCU Implications

Read-Copy Update (RCU) [48, 49] is a synchronization mechanism that allows low overhead wait-free reads at the cost of potentially expensive updates – each update must wait for a grace period to elapse before it completes. Specifically, an update thread must wait for a quiescent state to occur on each of the online cores in order to complete its update. As a result, an update operation that begins prior to C_F's crash and does not complete before the crash might wait forever for a quiescent state to occur on the faulty, non-responsive, core. In order to prevent such deadlocks, following a core crash, we explicitly allow all RCU updaters waiting for a quiescent state to occur on C_F to proceed. Likewise, future updates must avoid the waiting for C_F, and stop tracking it, as its future quiescent states will not happen. We therefore iterate over all RCU data-structures and delete all wait list entries corresponding to C_F, thus preventing future RCU updaters from waiting in vain. Figure 5 shows the high level operation of the corresponding code, which is executed as part of stage (iii).

6.5 Using HTM

We gathered statistics about lock usage in kernel code, and found that the run-queue locks are among the most commonly acquired for the workloads we study and the most commonly occur in the kernel source. We therefore chose to elide these locks as a case study for using transactions for recovery. We replaced all kernel critical sections protected by run-queue locks with the lock elision code in Figure 6, using Intel TSX. Here, we assume a convention in the Linux kernel, whereby all accesses to the run-queues are protected by these locks.

Each critical section is executed transactionally, and, as befits a best effort TM, is provided with a fallback path (line 9) [35]. The fallback path retries to execute the transaction, up to a predefined number of times. If the allowed number of retries has been exhausted, the implementation resorts to regular lock acquisition (line 13). To ensure correctness and reciprocity of a transaction with the fallback path, hardware transactions must read the lock as free (line 4), thus inserting the lock value into their read sets. The transactional memory semantics then guarantee that the transaction commits only when there is no ongoing fallback execution.

Though resorting to regular locking compromises our ability to recover from core failures in critical code, it is necessary to prevent livelock. The retries limit value determines a tradeoff – higher values favor reliability, whereas lower ones favor performance, as they shorten the maximum possible time to spend on retrying. Also, too high values may cause abort-prone sections to retry numerous times, thus harming the commit rates of other sections, causing them to resort to locking. We examined different limit values, and found the sweet spot to be at 10000 retries. Our results in the next section show that more than 99% of the transactions commit successfully in a lock-free manner, making failures inside a lock-protected critical section extremely improbable.

Among the 47 kernel critical sections protected by the run-queue locks, only three could not commit transactionally. Not surprisingly, one of them contains the context-

```
rcu_report_crash(cpu){
  // Update all RCU trees
  for_each_rcu_tree{
    struct rcu_node* rnp;
    rnp = rcu_tree_leaf_node(cpu);
    while (rnp != NULL){
      // For future grace periods -
      // Exclude cpu from the initial mask
      remove_from_mask(rnp->qsmaskinit,cpu);
      rnp=rnp->parent;
    }
  }
  // For the current grace period -
  // Artificially report a quiescent state
  rcu_report_qs(cpu);
}
```

Figure 5: RCU recovery.

```
1  RetryTxn:
2  // start the transaction
3  if (_xbegin() == _XBEGIN_STARTED){
4      if (raw_spin_is_locked(&rq->lock)){
5          _xabort(1);
6      }
7      /*** Critical Section Code Here ***/
8      _xend(); // finish the transaction
9  }else{ //Tx failed - fallback:
10     if(retries++ < MAX_RETRIES){
11         goto RetryTxn;
12     }
13     raw_spin_lock(&rq->lock);
14     /*** Critical Section Code Here ***/
15     raw_spin_unlock(&rq->lock);
16 }
```

Figure 6: Lock elision using HTM [35].

Benchmark	Workload Properties			Success Rate	
	user	*system*	*iowait/idle*	*non-system*	*system*
K-means	99%	1%	0%	100%	86%
401.bzip2	99%	1%	0%	100%	72%
410.bwaves	99%	1%	0%	100%	88%
429.mcf	22%	14%	64%	100%	68%
Postmark	5%	21%	74%	100%	70%

Table 2: Recovery rates, without HTM, under random fault injections in user and kernel code.

switch function, which issues a TLB flush – an operation that cannot commit transactionally. The second creates a new kernel thread, and is called only a handful of times during the system's lifetime. We left the first two critical sections with a surrounding lock, compromising our ability to recover from failures in these critical sections. The third critical section has a large data set and could successfully commit after we split it into three smaller sections, after ensuring that such chopping is safe.

Given the above, we conclude that eliding the run-queue locks constitutes a good case study. While the most contended lock in the kernel will be workload-dependent, run-queue locks are commonly used and some of the critical sections they protect have complex behaviors that are challenging for HTM mechanisms. Our work is a proof of concept for using HTM for recovery on a real system. A complete conversion of kernel locking to HTM usage has already been presented on a simulator in TxLinux [59], and in that aspect, TxLinux is complementary to our work.

7. Evaluation

This section presents our evaluation of CSR. We begin in Section 7.1 with a massive random fault injection campaign on a virtualized environment protected by the basic CSR algorithm, without using HTM. Next, in Section 7.2, we evaluate CSR with the lock elision code on real systems. Finally, in Sec. 7.3, we quantify the effects of using HTM on energy and performance using a dedicated scheduler benchmark.

In all experiments, except noted otherwise, the timer interrupt is set to the default period of 4ms and the FDU wakeup period, which can be set to any value higher than the timer interrupt period, is set to 10ms.

7.1 Massive Virtualized Fault Injection Without HTM

We install our CSR-enhanced Linux on a 4-core VM emulated by QEMU [6]. As we employ QEMU without HTM emulation, we run a kernel with only the basic CSR functionality and no lock elision. Hence, kernel critical sections are not protected. We run three sets of experiments, with random fault injections during three different execution modes: (1) user mode, (2) kernel mode, and (3) idle/IO-wait mode.

Emulating Failures. We change the QEMU source, causing it to shutdown a random virtual core at a random time when this core is executing in the desired mode (user, kernel

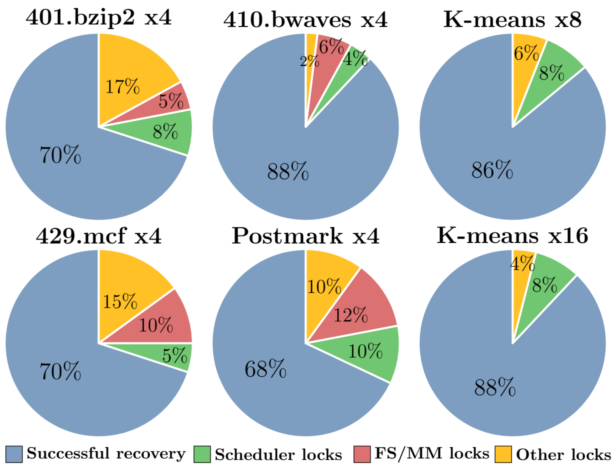


Figure 7: Recovery rates and failure locations under random fault injection in kernel code, no HTM in use.

or idle), thus simulating a permanent core fault. This is done by abruptly stopping one of the QEMU threads that simulate the virtual cores. We determine the execution mode by examining the CPL and EIP registers of the victim core. Note that an unmodified kernel crashes following a fault injection during any execution mode.

In each experiment, we start a VM running our kernel, run a given workload, wait for a virtual core to crash during the required execution mode, and allow the system time to recover. In order to verify that the system has indeed recovered, we create a new file, write a timestamp into it, and flush it to disk using `sync`. Thus, a file is created per each successful recovery. We repeat the experiments thousands of times per workload and per execution mode.

The benchmarks we use are K-means of the Metis [46] in-memory MapReduce library; SPEC-CPU™2006 [29] benchmarks, out of which we choose one data-intensive (429.mcf) and two CPU-intensive (401.bzip2, 410.bwaves) applications; and Postmark [40], an IO-intensive filesystem benchmark. Since Postmark and SPEC benchmarks are single threaded, we run four instances of each application, in order to keep all the cores occupied. In order to exercise the scheduling system, we run K-means with 8 and 16 threads on the four cores. We measure the induced workload properties of each benchmark using SYSSTAT [27].

Results. Our results, which appear in Table 2, show that our system recovered in all cases wherein fault injections were performed in user or idle mode, as expected. For fault injections during kernel code execution, the system recovered in about 70% (or more) of the experiments. For each experiment that resulted in a system crash, we extracted the exact code line before which the fault was injected (using the EIP and CPL registers) and analyzed the reason that prevented the system from recovering. We present in Figure 7 a breakdown of the failure reasons into three main categories: (1) holding a *scheduler lock*, such as the run-queue locks; (2) holding a *filesystem or a memory lock*; and (3) holding *other locks*, such as RCU and timer locks. As can be seen, IO-intensive workloads (mcf, Postmark) tend to crash during filesystem and memory operations more often than their computation-intensive counterparts. In addition, all workloads suffer considerably from crashes due to holding a scheduler lock. These failures occur since our system is tested here without eliding the scheduler locks (see Sec. 6.5), which we next evaluate on a real HTM-equipped system.

7.2 Experiments on a Real System

We next evaluate our implementation in two physical environments. The first is a PC running Ubuntu 14.04 on a 1x4x2 Intel Core i7-4770, TSX-equipped processor. The second is a server running Ubuntu Server 12.04 on 4x8x2 Intel Xeon E5-4650 processors.

7.2.1 Crash Simulation

For simulating a crash in a real system, we force a core to hang inside various kernel critical and non-critical sections in an un-interruptible state, thus causing it to stop responding. Figure 8 shows the code snippet we use for crashing an online core. The return value of *fault_injection()* is negative until set to be some core's id by a system call. Once the code is invoked on the victim core, it hangs in an infinite loop, with interrupts disabled, and therefore no possibility to be preempted. Note that this simulates our failure model, where the values in the crashed core's cache is accessible to other cores following a failure.

In order to hang a core on a real system, the code snippet (Figure 8) has to be hard coded into the kernel source. This does not allow us to perform automatic fault injection at random locations as in the virtualized case. Therefore, we manually perform fault injections in one hundred representative kernel functions (a partial list appears in Table 3). For functions that contain critical sections protected by a run-queue lock, we inject a crash in the critical section itself.

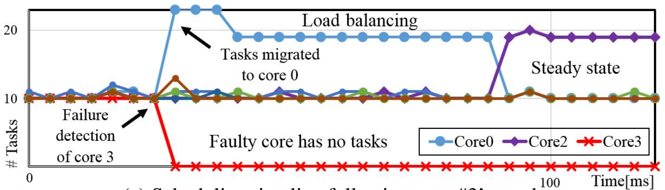
7.2.2 Crash and Recovery Timeline

In this set of experiments, we create and affine to each core 10 computation-heavy tasks. These tasks are always hungry for CPU time, and so, at any given moment during the experiment, each correct core should have at least 10 tasks in its run-queue. Next, we crash one of the cores and let CSR recover the system. We repeat the experiment on the PC by placing the code snippet of Figure 8 in the one hundred kernel critical and non-critical sections mentioned above.

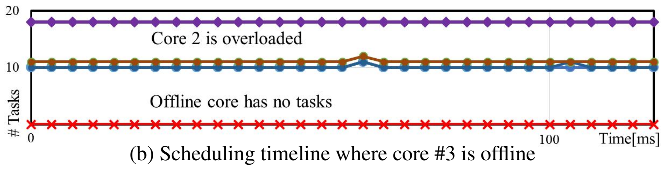
Traditional operating systems do not recover from such faults. Linux, for example, uses a watchdog [14] to detect

```
interrupts_disable();
if (fault_injection() == smp_processor_id())
    while(TRUE);
```

Figure 8: Crashing a core.



(a) Scheduling timeline following core #3's crash



(b) Scheduling timeline where core #3 is offline

Figure 9: Scheduling timelines for PC system with 8 cores.

lock-ups. Following a fault detection, Linux provides two possible actions (determined at system install time). The first is to reboot the system, and the second is to ignore the fault and resume normally. We configured our unmodified Linux system to the second option and used the method described in Figure 8 to hang a core. We repeated this experiment multiple times, and got a variety of behaviors, all of which led to a system freeze within a few seconds, possibly due to lost interrupts, unanswered IPIs, synchronization problems (held locks, stuck RCU operations), etc.

On the other hand, with CSR, the system successfully recovered in *all* the experiments. Exemplary results of experimenting on the PC and server are shown in Figures 9 and 10, respectively. To improve readability, we increase the FDU timeout to 100ms, show only cores 0-15, and disable hyper-threading on the server, just for the depicted experiments. The number of tasks in each run-queue is sampled every clock interrupt by a tool we implemented, and we plot the resulting scheduling timeline. Figure 9a shows the recovery timeline following a crash of core 3 (out of 8). We see that the system recovers, and the tasks that belong to the faulty core are migrated to the lowest id correct core. After less than 70ms, the load balancing mechanism kicks in and corrects the overload. Perhaps surprisingly, the tasks are migrated to core 2. The reason for this behavior lies in the scheduling-domains approach [41], according to which, in our case, each pair of logical cores constitute a scheduling domain. After the crash of core 3, core 2 constitutes a domain by itself. Therefore, the balancing among the domains causes core 2 to get a double amount of work. To verify this analysis, we examine the system with the same workload and take core 3 offline using CPU-Hotplug in an unmodified Linux kernel. As can be seen in Figure 9b, the load balancing behaves the same in both cases. We conclude that the Linux load balancer is not tuned for considering offline cores, and leave fixing this issue for future work.

Figure 10 shows the scheduling timeline of our server platform, after a crash of core 13 among 32. We see that the system recovers, and the overload formed on core 0 is spread among the rest of the cores.

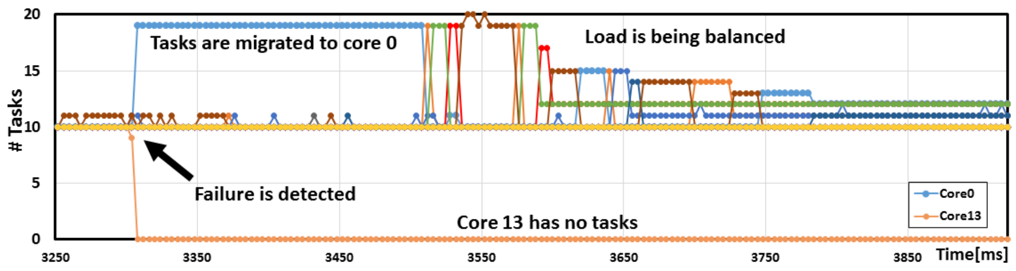


Figure 10: Scheduling timeline, on server with 32 cores (16 shown) following the crash of core 13.

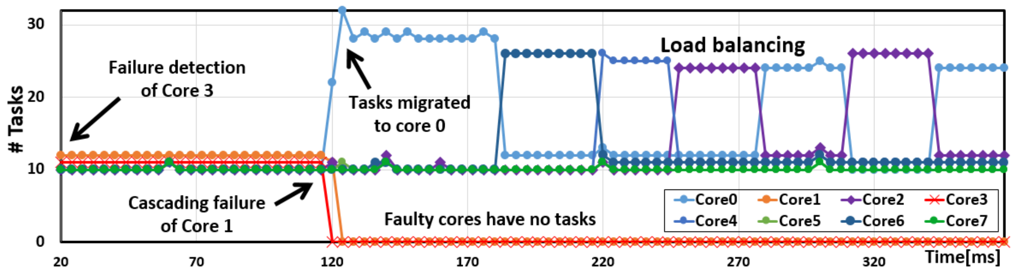


Figure 11: Scheduling timeline, with cascading failures on a PC.

We simulate cascading failures by crashing a core, and immediately afterwards, crashing the core that was nominated by the FDU to execute CSR. Figure 11 presents the resulting scheduling timeline on the PC. Here, the FDU detects core 3 as faulty and nominates core 1 to handle the fault. Core 1 crashes before it sends an ack to the FDU, and 100ms (the FDU wakeup period) later, the FDU considers core 1 to be faulty as well. It then nominates core 2 to perform CSR for both cores 1 and 3, and the system recovers.

7.2.3 Multiple Virtual Machines

Servers nowadays often run multiple VMs. In the absence of support for core surprise removal, a crash of a single core brings down the entire system, along with all running VMs. To demonstrate that CSR eliminates this problem, we conduct the following experiment: We install our CSR-protected Linux on our 64 hardware threads server. Using unmodified QEMU, we create and run four VMs; each is allocated 16 cores and runs an unmodified Ubuntu 14.04. We then set the affinity of VM i to the CPU in socket i and cause a permanent failure to one of the cores in the 4th socket. Thanks to CSR, only the fourth VM, (which has affinity to the fourth chip), suffers from the fault and crashes, while the remaining VMs continue normally. We revert the host OS to an unmodified kernel and repeat the experiment. Not surprisingly, without CSR, all the virtual machines, along with the server itself, crash.

7.3 Lock Elision

We now proceed to quantify the energy and performance implications of using HTM instead of locks in critical kernel code. We use the SysBench [1] benchmark tool, and gather performance, energy consumption (using the RAPL [38] interface), and abort statistics under various workloads on our PC system. Since we apply lock elision on the run-queue lock, we use the *threads* test mode of Sysbench, which is intended for measuring scheduler performance. We set the retries limit to 10000 for all critical sections. By examining the commit rates of each critical section, we found one exceptional critical section (*task_tick)*) that was able to commit, but caused, under some workloads, a significant increase to the total system abort rate. In principle, this code section updates only local data, and should abort rarely. However, the best-effort nature of Intel-TSX causes it to abort under certain workloads due to reasons like cache evictions. To avoid performance penalties, we set the retries limit for this critical section to 10, thus prevent it from contending excessive times with other sections and improve the overall commit rate.

Measurements of our reliable kernel over billions of transactions appear in Table 4. The use of lock elision eliminates expensive atomic instructions. This results in small improvements to the energy consumption and performance.

File	Functions
kernel/sched/core.c	scheduler_tick(), schedule(), ttwu_queue()
kernel/watchdog.c	watchdog_timer_fn()
kernel/timer.c	__run_timers()
kernel/workqueue.c	__queue_work()
kernel/softirq.c	wakeup_softirqd(), __do_softirq()
kernel/events/core.c	update_event_times()
kernel/pid.c	alloc_pid()

Table 3: Functions injected with faults (partial list).

Workload	Commit Rate	Fall-Backs on Locking	Performance Gain	Energy Saving
Idle	61%	0%	-	4%
16-threads	93%	0.1%	0%	1%
32-threads	80%	0.1%	3%	3%
64-threads	42%	0.2%	2%	2%

Table 4: HTM implications on performance and energy.

As can be seen, commit rates are always higher than 40%, meaning that transactions require less than 3 retries on average to commit successfully. The second column shows that the percentage of critical sections that exhaust all retries and resort to locking is negligible, meaning that the vast majority of critical sections execute in a reliable manner.

8. Related Work

To the best of our knowledge, CSR is the first system to address unexpected permanent core failures in commodity architectures and OS.

Reliability Support in OS. A number of works have addressed permanent or intermittent hardware failures: Hive [12], is designed to cope with fail-stop failures. It is built of independent kernels and confines errors to kernels where they occur. However, it is intended for a special architecture, is incompatible with commodity OS, and was not tested on a real system. Dobel and Hartig [18] designed an OS that tolerates soft errors using redundant threads by transferring essential OS code to a dedicated reliable computing base. CSR, on the other hand, does not assume the existence of a reliable core. C^3 [64] provides predictable recovery from intermittent faults in embedded and real-time systems. However, it does not address commodity computers and OSes.

Dolev and Yagel [19] present two self-stabilizing OS principles, which allow an OS to start from any initial state. Unlike CSR, they do not deal with allowing a system to continue to run in case of hardware failures.

Other works have addressed software-induced failures. MINIX3 [30], for example, is a reliable micro-kernel that detects software failures such as deadlocks, and restarts the faulty software component for recovery. Microreboot [10] performs fine-grain rebooting of application components, but is not designed for kernel code. Nooks [68, 69] and SafeDrive [79] provide fault isolation and recovery for device drivers by inspecting their interaction with the kernel, and are thus able to recover from failures in drivers and other kernel extensions without rebooting the OS. Akeso [44] is a system based on the Recovery Domains principle, which uses logging and rollback to tolerate faults in the entire kernel. All efforts mentioned above tolerate software-induced failures only, and except for Akeso [44], none of them tolerates errors in core kernel code.

Transactional Memory. Rossbach et al. have proposed the use of TM in kernel code, and presented TxLinux [59], which exploits TM in order to improve performance. CSR, on the other hand, exploits HTM for reliability. FaulTM [75] utilizes a modified HTM for reducing the overhead of double execution; however, it does not address execution of OS code. Moreover, unlike both of these works, CSR was tested on a real system, taking into account the best-effort nature of real HTM implementations.

CPU-Hotplug. Various works have pointed out CPU Hotplug's shortcomings. Gleixner et al. [26] propose improvements to the CPU-Hotplug path for energy and real-time purposes. Panneerselvam and Swift [54] propose operating system support for dynamic processors, which can dynamically reconfigure the machines' cores at runtime. However, neither of these refers to hardware failures, and both require the cooperation of the victim core.

Many-Core OS. Many-core operating systems is an active area of research. The TeraFlux project [25] encompasses various aspects of teradevice computing, including reliability [21–23, 72]. These works mainly focus on designing and implementing hardware FDUs, but do not consider the OS implications upon failure detection, and are therefore complementary to our work. Other works on OS for systems with large processor counts mainly focus on improving performance and scalability [5, 9, 28, 45, 58, 78], rather than reliability. IBM BlueGene [53] features fault-aware job scheduling, which uses fault prediction to improve the supercomputer's performance, but without considering recovery methods within a faulty node.

9. Conclusions

Existing operating systems cannot recover from unexpected core failures. Thanks to technology scaling, many-core machines are going to be widely deployed in the foreseeable future. However, hardware shrinking introduces new reliability challenges which cannot be addressed by current software.

In this paper we have introduced CSR, a new approach for OS reliability in the face of permanent core faults. We further proposed the use of HTM to cope with faults in kernel critical sections. CSR can be integrated with kernels running on servers with multiple coherence domains [3, 25], preventing entire clusters from crashing following a single core fault. Moreover, a combination of CSR and application-level runtime solutions to re-issue work performed by crashing threads [17, 74] can conceal consequences of failures from user applications.

The CSR principle can be extended to cope with unrecoverable chip-originated soft-errors. Such errors, which are common even today, typically indicate that something is wrong with the chip and therefore it is considered good practice to take the malfunctioning core offline. Since the core is already known to be faulty, expecting it to perform the unplug steps is inadvisable. In addition, Machine Check Exceptions that catch such malfunctions today usually result in a kernel panic. Instead, one can catch the MCE, flush the cache, and revert to CSR upon such errors.

Acknowledgments

This research was funded in part by Hasso-Plattner Institue (HPI), the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI), and the Technion Funds for Security Research. We thank Chris Rossbach, Emmett Witchel and Mark Silberstein for many insightful suggestions that helped to improve the paper.

References

[1] Alexey Kopytov. SysBench - A Modular, Cross-Platform and Multi-Threaded Benchmark Tool, 2016.

[2] AMD®. Machine Check Architecture. In *AMD64 Architecture Programmer's Manual*, volume 2, chapter 9. May 2013.

[3] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. COTSon: Infrastructure for Full System Simulation. *SIGOPS Oper. Syst. Rev.*, 43(1):52–61, 2009.

[4] Ashok Raj. CPU Hotplug Support in Linux Kernel. In *Linux Documentation*.

[5] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *22nd Symposium on Operating Systems Principles*. Association for Computing Machinery, Inc., October 2009.

[6] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[7] N. Bobroff, A. Kochut, and K. Beaty. Dynamic Placement of Virtual Machines for Managing SLA Violations. In *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on*, pages 119–128, May 2007.

[8] S. Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6):10–16, Nov. 2005.

[9] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[10] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot: A Technique for Cheap Recovery. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 3–16, Berkeley, CA, USA, 2004. USENIX Association.

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[12] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-memory Multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 12–25, New York, NY, USA, 1995. ACM.

[13] C. Chen and M. Hsiao. Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review. *IBM Journal of Research and Development*, 28(2):124–134, March 1984.

[14] Christer Weingel. The Linux Watchdog API. In *Linux Documentation*.

[15] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Rivière. Evaluation of AMD's Advanced Synchronization Facility Within a Complete Transactional Memory Stack. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 27–40, New York, NY, USA, 2010. ACM.

[16] C. Constantinescu. Trends and Challenges in VLSI Circuit Reliability. *Micro, IEEE*, 23(4):14–19, July 2003.

[17] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[18] B. Döbel and H. Härtig. Who Watches the Watchmen? Protecting Operating System Reliability Mechanisms. In *The Eighth Workshop on Hot Topics in System Dependability*, Berkeley, CA, 2012. USENIX.

[19] S. Dolev and R. Yagel. Towards Self-Stabilizing Operating Systems. *Software Engineering, IEEE Transactions on*, 34(4):564–576, July 2008.

[20] I. Egwutuoha, D. Levy, B. Selic, and S. Chen. A Survey of Fault Tolerance Mechanisms and Checkpoint/Restart Implementations for High Performance Computing Systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013.

[21] B. Fechner, A. Garbade, S. Weis, and T. Ungerer. Fault Detection and Tolerance Mechanisms for Future 1000 Core Systems. In *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pages 552–554, July 2013.

[22] A. Garbade, S. Weis, S. Schlingmann, B. Fechner, and T. Ungerer. Fault Localization in NoCs Exploiting Periodic Heartbeat Messages in a Many-Core Environment. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 791–795, May 2013.

[23] A. Garbade, S. Weis, S. Schlingmann, B. Fechner, and T. Ungerer. Impact of Message Based Fault Detectors on Applications Messages in a Network on Chip. *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 0:470–477, 2013.

[24] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[25] R. Giorgi, R. M. Badia, F. Bodin, A. Cohen, P. Evripidou, P. Faraboschi, B. Fechner, G. R. Gao, A. Garbade, R. Gayatri, S. Girbal, D. Goodman, B. Khan, S. Koliaï, J. Landwehr, N. M. Lê, F. Li, M. Lujàn, A. Mendelson, L. Morin, N. Navarro, T. Patejko, A. Pop, P. Trancoso, T. Ungerer, I. Watson, S. Weis, S. Zuckerman, and M. Valero. TERAFLUX: Harnessing Dataflow in Next Generation Teradevices. *Microprocessors and Microsystems*, 38(8, Part B):976 – 990, 2014.

[26] T. Gleixner, P. E. McKenney, and V. Guittot. Cleaning Up Linux's CPU Hotplug for Real Time and Energy Management. *SIGBED Rev.*, 9(4):49–52, Nov. 2012.

[27] S. Godard. *SYSSTAT Utilities - System Performance Tools for the Linux Operating System*, 2016. Available at `http://sebastien.godard.pagesperso-orange.fr/`.

[28] G. Heiser. Many-Core Chips — A Case for Virtual Shared Memory. In *Workshop on Managed Many-Core Systems*, Washington DC, USA, Mar 2009.

[29] J. L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006.

[30] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. MINIX 3: A Highly Reliable, Self-Repairing Operating System. In *ACM SIGOPS Operating Systems Review*, 2006.

[31] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[32] Intel®. OS Machine Check Recovery on Itanium®-Based Systems. Aug. 2008.

[33] Intel®. Intel® Cache Safe Technology. In *The Intel® Itanium® Processor 9300 Series*. 2014.

[34] Intel®. Instruction Set Reference. In *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 2, chapter 4. Dec 2015.

[35] Intel®. Intel TSX Recommendations. In *Intel 64 and IA-32 Architectures Optimization Reference Manual*, chapter 12. Sep 2015.

[36] Intel®. Intel® Transactional Synchronization Extensions. In *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 1, chapter 15. Dec 2015.

[37] Intel®. Machine-Check Architecture. In *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3, chapter 15. Dec 2015.

[38] Intel®. RAPL Interface. In *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3, chapter 14. Dec 2015.

[39] R. Iyer, R. Illikkal, O. Tickoo, L. Zhao, P. Apparao, and D. Newell. VM3: Measuring, Modeling and Managing VM Shared Resources. *Comput. Netw.*, 53(17):2873–2887, Dec. 2009.

[40] Jeffrey Katcher. Postmark: a New File System Benchmark. Technical report, October 1997. TR3022, Network Appliance.

[41] Jonathan Corbet. *Scheduling Domains*, 2004. Available at `http://lwn.net/Articles/80911/`.

[42] C.-K. Koh, W.-F. Wong, Y. Chen, and H. Li. The Salvage Cache: A Fault-Tolerant Cache Architecture for Next-Generation Memory Technologies. In *Computer Design, 2009. ICCD 2009. IEEE International Conference on*, pages 268–274, Oct 2009.

[43] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu. An Analysis of Performance Interference Effects in Virtual Environments. In *In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2007.

[44] A. Lenharth, V. Adve, and S. King. Recovery Domains: An Organizing Principle for Recoverable Operating Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, pages 49–60, 12 2008.

[45] LSE. *Linux Scalability Effort Homepage*, 2004. Available at `https://lse.sourceforge.net/`.

[46] Y. Mao, R. Morris, and M. F. Kaashoek. Optimizing MapReduce for Multicore Architectures. *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep*, 2010.

[47] W. Maurer. *Professional Linux Kernel Architecture*. 2008.

[48] P. E. Mckenney, J. Appavoo, A. Kleen, O. Krieger, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-Copy Update. In *In Ottawa Linux Symposium*, pages 338–367, 2001.

[49] P. E. Mckenney and S. Boyd-wickizer. RCU Usage in the Linux Kernel: One Decade Later. Technical Report, sep 2012.

[50] Microsoft ®. Windows Hot Add CPU.

[51] Z. Mwaikambo, A. Raj, R. Russell, J. Schopp, and S. Vaddagiri. Linux Kernel Hotplug CPU Support. In *Linux Symposium*, 2004.

[52] E. B. Nightingale, J. R. Douceur, and V. Orgovan. Cycles, Cells and Platters: An Empirical Analysis of Hardware Failures on a Million Consumer PCs. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 343–356, New York, NY, USA, 2011. ACM.

[53] A. J. Oliner, R. K. Sahoo, J. E. Moreira, M. Gupta, and A. Sivasubramaniam. Fault-Aware Job Scheduling for BlueGene/L Systems. In *IPDPS*, 2004.

[54] S. Panneerselvam and M. M. Swift. Chameleon: Operating System Support for Dynamic Processors. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 99–110, New York, NY, USA, 2012. ACM.

[55] D. A. Patterson. An Introduction to Dependability. *login*, pages 61–65, 2002.

[56] M. Radetzki, C. Feng, X. Zhao, and A. Jantsch. Methods for Fault Tolerance in Networks-On-Chip. *ACM Comput. Surv.*, 46(1):8:1–8:38, July 2013.

[57] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

[58] B. Rhoden, K. Klues, D. Zhu, and E. Brewer. Improving Per-node Efficiency in the Datacenter with New OS Abstractions. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 25:1–25:8, New York, NY, USA, 2011. ACM.

[59] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, A. Bhandari, and E. Witchel. TxLinux: Using and Managing Hardware Transactional Memory in an Operating System. In *SOSP*, 2007.

[60] D. Rossi, N. Timoncini, M. Spica, and C. Metra. Error Correcting Code Analysis for Cache Memory High Reliability and Performance. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March 2011.

[61] A. Roytman, S. Govindan, J. Liu, A. Kansal, and S. Nath. Algorithm Design for Performance Aware VM Consolidation. Technical report, 2013.

[62] R. D. Schlichting and F. B. Schneider. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, Aug. 1983.

[63] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM Errors in the Wild: A Large-Scale Field Study. In *SIGMETRICS*, 2009.

[64] J. Song, J. Wittrock, and G. Parmer. Predictable, Efficient System-Level Fault Tolerance in C^3. *2013 IEEE 34th Real-Time Systems Symposium*, 0:21–32, 2013.

[65] S. Srikantaiah, A. Kansal, and F. Zhao. Energy Aware Consolidation for Cloud Computing. In *Proceedings of the 2008 Conference on Power Aware Computing and Systems*, HotPower'08, pages 10–10, Berkeley, CA, USA, 2008. USENIX Association.

[66] J. Srinivasan, S. Adve, P. Bose, and J. Rivers. The Impact of Technology Scaling on Lifetime Reliability. In *Dependable Systems and Networks, 2004 International Conference on*, pages 177–186, June 2004.

[67] Srivatsa S. Bhat. *CPU Hotplug: stop_machine()-Free CPU Hotplug*. Available at `http://lwn.net/Articles/533553/`.

[68] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering Device Drivers. *ACM Trans. Comput. Syst.*, 24(4):333–360, Nov. 2006.

[69] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 207–222, New York, NY, USA, 2003. ACM.

[70] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *Solid-State Circuits, IEEE Journal of*, 43(1):29–41, Jan. 2008.

[71] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 127–136, New York, NY, USA, 2012. ACM.

[72] S. Weis, A. Garbade, B. Fechner, A. Mendelson, R. Giorgi, and T. Ungerer. Architectural Support for Fault Tolerance in a Teradevice Dataflow System. *International Journal of Parallel Programming*, pages 1–25, 2014.

[73] S. Weis, A. Garbade, and T. Ungerer. Design Exploration of FDUs and Core-Internal Fault-Detection. *Exploiting Dataflow Parallelism in Tera-Device Computing*, 2010.

[74] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[75] G. Yalcin, O. Unsal, and A. Cristal. FaulTM: Error Detection and Recovery Using Hardware Transactional Memory. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 220–225, San Jose, CA, USA, 2013. EDA Consortium.

[76] G.-C. Yang. Reliability of Semiconductor RAMs with Soft-Error Scrubbing Techniques. *Computers and Digital Techniques, IEE Proceedings*, 142(5):337–344, Sep 1995.

[77] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel ® Transactional Synchronization Extensions for High-performance Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 19:1–19:11, New York, NY, USA, 2013. ACM.

[78] G. Zellweger, S. Gerber, K. Kourtis, and T. Roscoe. Decoupling Cores, Kernels, and Operating Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 17–31, Broomfield, CO, Oct. 2014. USENIX Association.

[79] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-based Techniques. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 45–60, Berkeley, CA, USA, 2006. USENIX Association.