Fault Tolerance Through Redundant Execution on COTS Multicores: Exploring Trade-offs

Yanyan Shen* Gernot Heiser† Kevin Elphinstone‡
UNSW Sydney and Data61, CSIRO, Australia
Email: *yanyan.shen@unsw.edu.au, †gernot@unsw.edu.au, ‡k.elphinstone@unsw.edu.au

***Abstract*—High availability and integrity are paramount in systems deployed in life- and mission-critical scenarios. Such fault-tolerance can be achieved through *redundant co-execution* (RCoE) on replicated hardware, now cheaply available with multicore processors. RCoE replicates almost all software, including OS kernel, drivers, and applications, achieving a sphere of replication that covers everything except the minimal interfaces to non-replicated peripherals. We complement our original, loosely-coupled RCoE with a closely-coupled version that improves transparency of replication to application code, and investigate the functionality, performance and vulnerability trade-offs.**

***Index Terms*—seL4; microkernel; SEU; replication; fault tolerance;**

I. Introduction

Computer systems in control of life- and mission-critical functions require high levels of integrity and availability, even in the case of component failure. The standard approach to achieving the required fault tolerance is to use *dual* or *triple modular redundancy* (DMR or TMR, respectively), where all critical functions (hardware as well as software) are replicated [1]. Such redundant hardware architectures are traditionally employed in scenarios where the cost of failure is unbearably high.

Traditional redundant designs are expensive, in terms of capital cost and often also in performance, they also tend to be robustly engineered and correspondingly bulky and heavy [1]–[4]. This creates *space, weight and power* (SWaP) problems that limit the use of such systems.

Computer control of critical systems is rapidly becoming more widespread, especially with the move towards autonomous land and aerial vehicles, and the explosive growth of small-satellite launches [5]. Many of those systems are too cost- and SWaP-sensitive for traditional fault-tolerance approaches. At the same time, on-going miniaturisation of commercial off-the-shelf (COTS) processors is increasing their vulnerability to transient faults, such as *single-event upsets* (SEUs) caused by ambient ionising radiation [6]–[10]. In other words, there is an increased risk of failure at the same time as critical systems are becoming more widespread.

Recent progress in formal verification has now made it possible to achieve 100% reliability in critical software components, such as the OS [11], file system [12], and security protocol implementations [13]. However, such verification inevitably assumes perfect hardware that always operates according to its specification. A single, transient bit flip can invalidate verification assumptions, and can lead to security violations, just as in unverified systems [14]–[16].

COTS hardware is far from perfect, and reliability issues are well established [17]–[22], making hardware redundancy particularly important. However, the abundance of multicore processors, especially high-performance, energy-efficient systems-on-chip designed for phone use, makes processor redundancy relatively cheap in terms of capital cost as well as SWaP. We have previously shown that on a multicore, a redundant OS can run a redundant software stack, where the application software is unaware of replication [23], with a *sphere of replication* (SoR) [24] covering almost all software.

This earlier work on *redundant co-execution* (RCoE) used loosely synchronised replication that is advantageous for performance but cannot support applications that contain data races, such as concurrency control using lock-free algorithms or atomic instructions. This also rules out supporting virtual machines (VMs), as we cannot assume that applications inside a VM are free from data races. Here we generalise the RCoE approach to support such use cases. Specifically:

- We introduce *closely-coupled redundant co-execution* (CC-RCoE), which makes fewer assumptions about application behaviour, and present its design and implementation on x86 and Arm multicore processors (Section III).
- We introduce an error-masking approach for RCoE that allows a TMR configuration to downgrade to DMR operation (Section IV).
- We perform an extensive performance comparison of CC-RCoE against the original, loosely-coupled variant (LC-RCoE), using microbenchmarks (Section V-A) and system benchmarks (Section V-B);
- We evaluate the ability of the schemes to detect errors in memory or CPU registers (Section V-C), and to mask errors (Section V-D).

II. Background

A. Soft Errors

A single-event upset is a non-destructive (transient) change of state in a storage element, affecting single or multiple bits, usually caused by high-energy particles originating in cosmic radiation or ambient natural radioactivity [6], [25].

Shrinking feature sizes, and reductions of supply voltage, noise margins, and node capacitance increase sensitivity to soft errors and lower-energy particles [26]–[28]. Increased device density also increases the likelihood of a single strike affecting multiple components, resulting in SEC-DED ECC memory providing insufficient protection for modern memory systems, with up to 20 *undetected* failures in time (FIT) per DRAM device reported [22]. Failure rates of CPU and DRAM are significant even under terrestrial conditions: 1 in 190 for CPU subsystems and 1 in 1700 for DRAM (one bit-flip) during a period of 30-day total accumulated CPU time [20].

B. Redundant Co-Execution

A standard approach to redundancy is using lock-stepping [29] or loosely-synchronised [1] processors with hardware-supported voting. The purchase and maintenance costs of such commercial systems are significant, and their sizes and power requirements make them unsuitable for embedded systems. There is also growing demand for performance in embedded systems, for instance satellites [30], which is at odds with the performance characteristics of radiation-hardened processors [31], [32].

COTS multicore processors are not sufficiently synchronised to support lock-step execution of replicas, and even if they were, such an approach would be prohibitively expensive without hardware-supported voting. Instead, *redundant co-execution* (RCoE) runs multiple replicas of a software system concurrently and independently on different CPU cores, until they reach an explicit synchronisation point [23].

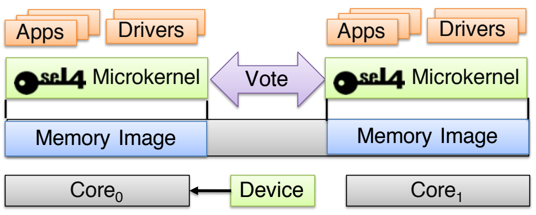

Fig. 1. Redundant co-execution (DMR configuration).

Fig. 1 shows an example DMR system based on RCoE. The whole software system running on Core_0 is replicated onto Core_1, with memory partitioned between replicas. RCoE redundantly executes most components of the software system, including the OS kernel and device drivers, and all management of replication and its synchronisation is done by the kernel. We can consider each replica running on a physical core a state machine [33], where state transitions result from device inputs. As devices are not physically replicated, only one replica, the *primary* on Core_0, can perform low-level device access and interrupt handling. All replicas must synchronise at the boundary of this non-replicated device code, so the state machines can process the same events. The non-replicated code at the primary is minimal, essentially device-register reads and writes. All the device driver *logic* is replicated.

III. Closely-Coupled RCoE

A. Nondeterminism

To ensure all replica state machines perform the same state transitions, it is sufficient to replicate the events that trigger transitions, i.e. input data and interrupts. LC-RCoE assumes that the system executes deterministically between events, and therefore can synchronise at any time between state transitions. It uses a logical time that counts deterministic events, i.e. system calls and application-triggered exceptions.

```
preempt R_a ⇒ ...
preempt R_b ⇒ x++;
              if (x == 1)
              else { ... }
```

Fig. 2. Data race creating divergence between replicas R_a and R_b.

Data races introduce internal non-determinism that will cause replicas to diverge. To understand this, assume an application, running on a single logical core, consisting of two threads T_0 and T_1. Both threads execute the same code segment shown in Fig. 2, which has a shared counter `x` initialised to 0. Assume at the previous time slice, T_0 of two replicas, R_a and R_b, was preempted at the points shown. Assume now that on the present timer tick, the replicas of T_1 start and keep running to the `if` statement. The replicas will diverge since they observe different values: `x` is 1 on R_a, but 2 on R_b.

Divergence will also happen with any multitasking workload running on a guest OS when using hardware-supported virtualisation (where the kernel coordinating the replication is now the hypervisor), not only with races in the application, but also lock-free synchronisation in the guest OS. This generally rules out supporting virtual machines with LC-RCoE.

B. Precise Logical Clock

CC-RCoE avoids this divergence by instruction-accurate synchronisation. We define the CC-RCoE logical time by adding the exact point in execution since the last deterministic event (i.e. the LC-RCoE clock tick). To obtain this clock, we make use of the fact that the number of backward branches taken, together with the current instruction pointer (IP), identify a unique point in the instruction stream [34]. A CC-RCoE replica's logical clock thus consists of the triple *(LC-RCoE_time, user_branches, user_IP)*. Note that we do not include instructions executed in the kernel, as kernel replicas inherently execute somewhat different instruction streams upon non-deterministic events (the primary differs from other replicas).

C. Synchronisation and Voting

RCoE synchronises on kernel entries, which is straightforward for system calls and other exceptions. Interrupts are only received by the primary replica; to force synchronisation on receiving an interrupt, the primary sets a flag and sends inter-processor interrupts (IPIs) to the other replicas. When all replicas observe the event, they vote a leading replica by comparing logical times of all replicas. The leading replica waits for the others to catch up by spinning on a kernel barrier. While spinning, it monitors the per-core cycle counter. If the spinning time exceeds a system-defined timeout value, this is taken as an indication of divergence (hanging replica).

When a replica must catch up, it sets a global instruction breakpoint[1] at the address of the next user-level instruction of the leading replica, and resumes execution. When the breakpoint exception fires, and the replica is still behind the leader (which can happen if the breakpoint is inside a loop), it repeats the process, else it joins the wait if there are more followers outstanding.

Once synchronised, the replicas need to vote, i.e. compare their state. Comparing all state would be prohibitive. We make this tractable by reducing critical kernel state updates, driver-contributed state updates, and system-call parameters to a three-word signature, consisting of the present event count plus a checksum representing state changes. To maximise the signature's sensitivity to historical changes, we use a Fletcher checksum [35], which is dependent on the values forming the checksum as well as the order in which they are applied.

We justify not comparing all system state by the observation that not all user state is critical, e.g. bit flips in image data are usually harmless, and the application should determine its integrity needs. In contrast, any updates to kernel data structures, such as page tables or capability storage, are potentially critical. Furthermore, divergence of an application's execution path will almost certainly lead to changes in system-call parameters, resulting in observed divergence.

Detection latency can be reduced by configuring the kernel's timer tick. In addition we allow an application to add critical data to the signature at any time through a new system call, `FT_Add_Trace`. Device drivers in particular should use this system call to contribute output data into the signature and reduce detection latency. We will examine the reliability benefits in Section V-C1.

D. Implementation Challenges

One might think that the precise logical clock would be easy to obtain with the help of the performance monitoring unit (PMU). However, on the COTS x86 and Arm processors, many PMU events are imprecise, exhibiting over- or under-counting [36].

Fortunately, on Intel processors [37] the difference between the number of branch instructions retired and the number of far branches retired is deterministic and equal to the number of branches executed in user mode, if we program the PMU counters to count only user-mode events. We confirm this experimentally on a number of Intel processors of the Haswell (Core i7-4770 and i5-4590) and Skylake (i7-6700) microarchitectures. ReVirt [38] uses a similar approach to record and replay nondeterministic events. Note that when running a virtual machine (VM), we count branches in user code as well as the guest kernel.

The x86 architecture presents an additional challenge through string operations with rep-family prefixes (e.g., `rep movsb`), which logically execute in a loop but do not increment branch counters. This makes it impossible to determine a precise logical time if a breakpoint is set at such an instruction. To avoid this case, we need to examine the memory referenced by the instruction pointer. If the code is in a VM, this requires locating the instruction by a software walk of the guest page table and the extended page table, which significantly adds to the cost of supporting VMs.

[1]A global breakpoint triggers an exception when any thread's program counter matches the breakpoint.

```
ldr r0, [r3]

bl   _IO_getc
uxtb  r0, r0
cmp r0, #66

ldmeqfd sp!, {r3,
    pc}
cmp r0, #55
moveq r0, #56

ldmfd sp!, {r3, pc}
```

Listing 1. Original code.

```
ldr r0, [r3]
add r9, r9, #1
bl   _IO_getc
uxtb  r0, r0
cmp r0, #66
add r9, r9, #1
ldmeqfd sp!, {r3,
    pc}
cmp r0, #55
moveq r0, #56
add r9, r9, #1
ldmfd sp!, {r3, pc}
```

Listing 2. Modified code.

For Armv7-A processors, including the Cortex-A9 cores we are using, we find no PMU events that produce accurate branch counts. Instead we adopt a compiler-based branch-counting technique demonstrated by Slye and Elnozahy [39], who built a record-replay fault-tolerant solution on a DEC Alpha processor. We use their ideas to develop a plugin for GCC to count branches, by inserting a count instruction before each call or jump instruction.

To avoid the overhead of accessing memory and minimise the number of extra instructions required, we reserve a register for maintaining the counter (using the `--ffixed-r9` argument to GCC). The register can be incremented in a single cycle, at the cost of stealing a register from the compiler's optimiser. The plugin iterates lists of `insns`, which are GCC's internal representation of instructions, and prepends each `call_insn` (function call) or `jump_insn` (jump) with an increment instruction on register `a9`. For example, the code of Listing 1 is transformed into that of Listing 2.

We ensure that the plugin is called after various optimisation passes, to avoid the extra instructions being optimised away. The reserved register is thread-local, i.e. context-switched like any register. The kernel only treats it as special during the synchronisation protocol, where it is monitored to determine when replicas have caught up. After syncing, it is reset to avoid overflow.

Synchronisation instructions also cause a problem: Armv7-A provides the load exclusive (`ldrex`) and store exclusive (`strex`) instructions, which are used in a retry-loop, for implementing atomic updates. As a result, the number of executions may differ between replicas, even when they do not diverge. We avoid this problem by requiring the use of a system call for atomic updates. This step can be automated in the future by using a binary rewriting tool, which scans the `ldrex-strex` pairs and converts them into system calls.

Armv7 does not have an equivalent of the x86 `resume` flag, which can temporarily disable a breakpoint without clearing it. We therefore need to handle two debug exceptions for every breakpoint: one for the target breakpoint and a second one for a

```
1000:    add r7, r7, #1
1004:    add r8, r8, #1
       ...
1108:    add r9, r9, #1
110c:    b 1000
```

Listing 3. Race condition on counter maintenance.

mismatch breakpoint, to single-step over the target breakpoint before re-enabling it again. This increases the overhead of synchronising the branch counter.

Another complication arises from the fact that our branch counter is not updated atomically with the branch, and execution may be preempted between the two instructions. Consider the code in Listing 3, and assume the primary has just executed the back branch, i.e. its instruction pointer is `0x1000`. If another replica with the same counter value is about to execute address `0x110c`, its branch counter already reflects the branch that has not yet been taken. Simply comparing instruction pointers would falsely indicate the primary as trailing. Therefore, we also need to check whether the last instruction executed by a replica is the counter incrementation and handle the case accordingly when voting the leader.

As CC-RCoE on Armv7-A depends on a compiler extension, all code (including libraries) must be recompiled. The critical systems we target tend have strict assurance requirements, which require source-code access. We also need to scan for necessary modifications to any assembly files or in-line assembly. It would be straightforward to build a tool that checks for such cases of assembly, to reduce the chance of overlooking some.

E. Device-Driver Support

The two RCoE models differ significantly in the support required for device drivers. Our implementation is based on the seL4 microkernel [11], which runs drivers in user mode. As such, they are almost normal processes, automatically replicated by RCoE. However, as mentioned in Section II-B, the actual device access is done by the primary replica only, so drivers are aware of the SoR boundary.

In LC-RCoE, device drivers are supported by augmented system calls, `ARM_Page_Map` and `IA32_Page_Map`, which create *cross-replica shared memory regions*; this allows the driver replicas to conduct input data replication in user mode.

In CC-RCoE, the replicas of a device driver must behave identically due to the requirement for precise preemption. This means that, unlike normal seL4 drivers, which directly access memory-mapped device registers, replicated drivers must delegate device-memory access and its input data replication to the kernel (where branches are not counted). We support this with two new system calls, `FT_Mem_Access` and `FT_Mem_Rep`; their signatures are shown in Listing 4. These calls are synchronisation points, so they only perform operations when all replicas are in sync.

`FT_Mem_Access` performs a read or write (as specified by `access_type`) of the device memory at address `va_mmio`, transferring the data to/from memory address `va_src_dest`.

```
int FT_Mem_Access(Word access_type, Word
    va_mmio, Word va_src_dest, Word size);
int FT_Mem_Rep(Word va, Word size);
```

Listing 4. Signatures of driver-support system calls.

When called by the primary for reading, this will copy from device memory to the kernel shared-memory region. Non-primaries block until the primary has performed the read, after that, each replica copies the shared value to `va_src_dest`. On writing, the primary replica writes the data to the device memory. The driver may optionally call `FT_Add_Trace` to force the output data into the signature. `FT_Mem_Rep` replicates a buffer used for direct memory access (DMA). Executed by the primary, it copies the specified buffer to the shared memory region, executed by another replica, it copies from the shared region to the caller's address space.

F. CC-RCoE vs LC-RCoE trade-offs

CC-RCoE requires more effort to maintain the logical clock than LC-RCoE, which means that we can expect a higher performance degradation. Hence, LC-RCoE is the preferred approach when its requirements are met.

For code that is known to contain data races, or that is too complex to assure free from races (including virtual machines), CC-RCoE is the only option. LC-RCoE is also more restricted in its ability to recover from errors, see Section IV-A.

For *hardware-assisted* CC-RCoE, as we use it on x86, overheads result from (i) reading performance counters, (ii) programming debug registers, and (iii) handling debug exceptions. These overheads can be significant if an instruction breakpoint is inside a tight loop. Furthermore, it is by no means sure that all future x86 processors (or even present, non-Intel ones) will provide the required hardware support.

The overhead of *compiler-assisted* CC-RCoE, as we use it on Arm, also has overheads beyond that of reserving a register. For each breakpoint we must program debug registers and handle the resulting debug exceptions. The catch-up overhead is again high if a breakpoint is inside a loop.

IV. Error Recovery

Once an error is detected, recovery is desirable. One approach would be to roll the system back to a checkpoint, or just restart the faulty replica and roll it forward from logged inputs. These options could be combined with RCoE, but checkpointing raises its own dependability issues (fault-tolerant storage of the checkpoints and logs), and would result in high storage overheads. Furthermore, restore and roll-forward would take considerable time, impacting availability.

A. Downgrading on Errors

A DMR configuration can only *detect* divergence, after which the only safe operation is to shut the system down (presumably after raising some alerts). In contrast, a TMR configuration can safely continue operation without service disruption by downgrading to DMR.

We support this [40], within limits. At present, we only recover from a failed vote on the state signature, not from a timeout while waiting for a straggler replica (although this limitation would not be hard to lift by shutting down the straggler's core). Also we require agreement on the identity of the diverging replica for downgrading.

All replicas *independently* vote on the signatures. If they agree that the signatures agree, execution continues normally. If all replicas agree that there is divergence, *and* they agree on the identity of the diverging replica, we downgrade: the faulty replica removes itself while the others wait for this to complete. If there is no consensus, we halt the system. Disagreement may result from faults in multiple replicas, corruption of the checksums, or a fault during voting.

When removing the primary, the other replicas need to elect a new primary (the remaining node with the smallest ID) and re-route interrupts to it. CC-RCoE handles I/O operations in the primary kernel, requiring reconfiguration of DMA buffers if the primary is removed. On x86 we support this by marking DMA buffers, using an unused bit in the page tables, and patching page-table entries when removing the primary. We do not have an unused bit in the page tables of Cortex-A9 processors, so we presently do not support error masking for CC-RCoE on Arm. The 64-bit Armv8 architecture as well as Armv7 processors with the large-physical-address extension do have such bits, so we will be able to support error masking for CC-RCoE on newer Arm processors.

Since I/O operations are not redundantly executed, we cannot downgrade if the primary is faulty and any replica is currently accessing I/O devices, as we cannot determine whether the faulty primary has initiated I/O operations that might corrupt the system.

B. Voting Algorithm

Listing 5 shows the voting algorithm, which is invoked if the checksums differ. It returns the ID of the diverging replica if there is consensus on the faulter, or flags an error otherwise. Note that the voting algorithm can be affected by transient faults as well, although the window is tiny. Thus, the algorithm is designed to be executed by all the replicas redundantly, with barriers to ensure fail-stop behaviour.

We first compare each replica's state signature with that of the other replicas, and increment a per-replica counter if the signatures match (lines 8–11). The barrier at line 12 ensures all replicas finish before proceeding to the next stage, or halts the system if the barrier times out. Lines 13–18 find the smallest value in the array `ft_votes`; the values in the array represent the number of checksums in the array `checksum` which agree with the one indexed by `my_rid`. Thus, the replica with the smallest value is the faulty one.

Lines 19–22 check for the following cases: (1) more than one replica is faulty, (2) all the checksums are the same. The votes received by each non-faulty replica should be the replica number (`N`) minus one if only one checksum is incorrect. Each replica stores the ID of the replica it has determined as faulty in the globally shared, per-replica variable

```
1  global_shared int ft_votes[N];
2  global_shared int ft_fault_replica[N];
3
4  int vote_fault_replica(void) {
5    int least_vote = N + 1;
6    int fault_replica = N + 1;
7    ft_votes[my_rid] = 0;
8    for (int i = 0; i < N; i++) {
9      if (checksum[i] == checksum[my_rid])
10       ft_votes[my_rid]++;
11   }
12   kbarrier(bar, N);
13   for (int i = 0; i < N; i++) {
14     if (ft_votes[i] < least_vote) {
15       least_vote = ft_votes[i];
16       fault_replica = i;
17     }
18   }
19   if (ft_votes[my_rid] != N - 1)
20     ft_fault_replica[my_rid] = my_rid;
21   else
22     ft_fault_replica[my_rid] =
           fault_replica;
23   kbarrier(bar, N);
24   for (int i = 0; i < N; i++) {
25     if (ft_fault_replica[i] !=
           ft_fault_replica[my_rid]) {
26       return ERROR_DIFF_FAULT_REPLICA;
27     }
28   }
29   kbarrier(bar, N);
30   return fault_replica;
31 }
```

Listing 5. The algorithm for voting a faulty replica.

`ft_fault_replica[my_rid]` if the check succeeds; otherwise, the replica stores its own ID. The barrier at line 23 ensures that all the replicas have finished the checking stage.

Finally, all the replicas check if the faulty replica voted by others is the same as the one chosen by itself. An error is returned if the faulty replica IDs are different, and the system halts (lines 24–28). If all replicas agree on the faulty replica, they pass the third barrier and the faulty replica ID is returned by the function (lines 29–30).

Table I shows two examples of voting. In the first, R_2 has an incorrect checksum, resulting in the lowest `ft_votes`, a consensus of R_2 being faulty. In the second example, all the checksums differ, resulting in all `ft_votes` being 1. In this

TABLE I
EXAMPLES OF VOTING.

	R_0	R_1	R_2
checksum	0xdeadbeef	0xdeadbeef	0xdeedbeef
ft_votes	2	2	1
ft_fault_replica	2	2	2
checksum	0xdeadbeaf	0xdeadbeef	0xdeedbeaf
ft_votes	1 (<2)	1 (<2)	1 (<2)
ft_fault_replica	0	1	2

case, each replica set its `ft_fault_replica[my_rid]` to its own ID; thus the check (lines 24–28) returns `ERROR_DIFF_FAULT_REPLICA` to indicate multiple faulty replicas. Note that this voting algorithm supports any number of replicas $N \geq 3$.

C. Re-integration

Re-integrating an off-lined replica is the same as upgrading from DMR to TMR operation. While not a critical feature, it is definitely desirable for systems unattended for an extended period. Upgrading is possible by copying all kernel and user state of the present non-primary replica to the new replica [40]. We have not implemented this yet, and for now require a full reboot to upgrade a DMR configuration to TMR.

V. Evaluation

We evaluate RCoE on both architectures. Our x86 processor is a Core i7 6700 quad-core running at 3.40 GHz, 2×32 KiB L1 and 256 KiB L2 cache per-core and a shared 8 MiB L3 cache, equipped with 8 GiB of DDR4-2133 memory, and an Intel I219-LM network card. The kernel and native apps run in 64-bit mode. For x86 VM benchmarks, the kernel and seL4's virtual-machine manager (VMM) run in 32-bit mode, and the VMM presently only supports 32-bit guests.

Our Arm platform is a SABRE Lite board based on an i.MX6 SoC [41], which features a quad-core Cortex A-9 processor (32-bit Armv7-A ISA), 2×32 KiB L1 caches, a shared 1 MiB L2 and 1 GiB of DDR3-1066 memory.

We use LC-D, LC-T, CC-D, and CC-T to represent **D**MR and **T**MR configurations using the **LC**-RCoE or **CC**-RCoE models. Note that the benchmarks for virtual machines are only conducted on the x86 machine, since running replicated virtual machines is not yet supported on the Arm board.

In tables, numbers in parentheses indicate standard deviations in units of the least significant digit.

A. Microbenchmarks

1) Tolerating Data Races: We use a simple program to demonstrate that CC-RCoE is able to tolerate multithreaded applications with data races. The benchmark starts 32 threads; each thread reads a shared counter to a local register, idles for a short interval, increases the local register, and writes the register back to the shared counter, in a loop. When all threads finish, we compare the shared counters of different replicas. The shared counter is not protected by a lock, so this setup contains data races.

For LC-RCoE, we observe that the counter values of the replicas diverge with high probability. With CC-RCoE, while the counter values tend to differ from the "correct" value (i.e. if locking were used), we never see a divergence between replicas in 1,000 runs on each architecture.

2) Dhrystone and Whetstone: To evaluate our framework's effect on CPU-bound applications, we port the Dhrystone [42] integer benchmark, as well as the Whetstone [43] floating-point suite, to run natively on seL4. Table II shows execution times (average of 10 runs) for the various configurations,

TABLE II
Dhrystone/Whetstone execution times in seconds.

	Dhrystone		Whetstone	
	Arm	x86	Arm	x86
Loops	200 M	1000 M	0.5 M	2 M
Base	146.098 (2)	108.1 (0)	108.9 (1)	120.3 (0)
LC-D	146.991 (0)	108.6 (1)	109.8 (5)	120.3 (1)
LC-T	146.992 (0)	108.6 (0)	109.8 (4)	120.4 (1)
CC-D	153.422 (0)	110.7 (1)	122.9 (66)	138.7 (40)
CC-T	153.427 (0)	111.9 (1)	133.5 (42)	143.0 (55)

as measured by the CPU cycle counters. We observe that for both benchmarks and on both architectures, LC-RCoE shows negligible overhead, in both the DMR (row LC-D) and TMR (LC-T) configurations. This is not surprising, as these benchmarks are CPU-bound, perform no system calls, and have small working sets that fit into the caches, avoiding contention on the memory bus. As such, they represent a best case for RCoE. The only overheads are from kernel entries resulting from preemption-timer ticks.

A striking feature of the CC-RCoE results is that the relative standard deviation of Whetstone runs is up to 5%. This is a consequence of the overhead being very sensitive to the location of the synchronisation point: if it is inside a loop, overhead will be high as explained in Section III-D, else it will be low. These simple benchmarks approach a worst-case scenario for maintaining our precise logical times, as they consist mostly of tight loops. The main difference is that Whetstone is structured as several tight loops, resulting in about 20% TMR overhead, while the main body of Dhrystone is one long loop, resulting in a TMR overhead of 4–5%.

3) Virtualised Dhrystone and Whetstone: To examine the cost of RCoE in virtualised environments, we run the benchmarks inside a Linux VM on top of our CC-RCoE seL4 kernel acting as the hypervisor (remember from Fig. III-A that LC-RCoE cannot support VMs). As the seL4 kernel version we use does not support hypervisor mode on Arm (it was added in a later version) we can run virtualised setups only on x86.

Table III shows the results. Note that the baseline numbers are not comparable to Table II, as the benchmarks are built quite differently: The native versions run in 64-bit mode, are compiled with optimisation disabled (per the comments in the source file of Dhrystone) and statically linked, while the virtualised programs run in 32-bit mode (seL4 does not

TABLE III
Virtualised microbenchmark execution times (s) and slowdowns on x86.

	Dhrystone		Whetstone	
Base	86 (0)		55 (0)	
CC-D	130 (11)	1.5×	159 (11)	2.9×

TABLE IV
VIRTUALISED SPLASH-2 EXECUTION TIME (S) ON X86 FOR N RUNS.

Name	N	Base	CC-D	Fact
BARNES	30	61 (0)	93 (19)	1.52
CHOLESKY	300	66 (0)	792 (150)	12.08
FFT	100	64 (0)	142 (13)	2.22
FFM	20	76 (0)	160 (37)	2.11
LU-C	30	64 (0)	437 (17)	6.83
LU-NC	20	62 (0)	381 (27)	6.12
OCEAN-C	1000	64 (0)	173 (1)	2.71
OCEAN-NC	1000	65 (1)	171 (1)	2.65
RADIOSITY	25	66 (0)	75 (0)	1.12
RADIX	20	66 (0)	89 (4)	1.34
RAYTRACE	1000	60 (0)	65 (1)	1.09
VOLREND	100	86 (0)	133 (1)	1.54
WATER-NS	600	66 (1)	92 (2)	1.41
WATER-S	600	67 (0)	84 (0)	1.25
Geometric mean				2.30

presently support 64-bit VMs) are built with optimisation level `-Os` and dynamically linked (the Buildroot [44] settings to create Linux user-mode applications optimised for size); they also use different libraries.

The results show a high performance impact of CC-RCoE in a virtualised environment, with execution time increasing 55% for Dhrystone and almost tripling on Whetstone.

The overheads are dominated by the cost of VM exits and entries. Intel virtualisation support reduces the impact of this cost by minimising the need for VM exits through system-call redirection, extended page tables and other optimisations; so normal execution has few VM exits. In contrast, our breakpoints force VM exits. The take-away is that CC-RCoE's support for virtualisation comes at significant cost.

4) SPLASH-2: SPLASH-2 [45] is a suite of parallel scientific computing kernels, which we again run in a Linux VM on x86, with `NPROC=2` (i.e. two threads); results are summarised in Table IV. Overheads range from 10% to a factor 12, with mean execution time around 2.3 times the baseline, which is comparable to the virtualised Whetstone overhead and thus in the expected range. The different sensitivities of the benchmarks is a reflection of the time spent in tight loops. The results confirm that the code taking a signficant share of overall execution time should be ported to native execution, rather than virtualised. With `NPROC=1` (single-threaded applications) the mean overhead drops slightly to 2.02.

5) Memory Bandwidth: To quantify the effect of redundant co-execution on the memory bandwidth available to applications, we stress the memory system with a simple copy benchmark. It uses `memcpy()` between two page-aligned memory buffers, each of which is four times the size of the last-level cache. We pre-map the buffers to avoid page faults, and each run repeats the copy 100 times. We report the average of 100 runs (i.e. 10,000 `memcpy()` invocations) in Table V.

TABLE V
MEMORY COPY BANDWIDTH (BW, GIB/S), AND REMAINING FRACTION (%) UNDER RCOE.

	Base	LC-D		CC-D		LC-T		CC-T	
	BW	BW	%	BW	%	BW	%	BW	%
x86	25.4	12.5	49	12.4	49	7.9	31	7.8	31
Arm	1.7	1.1	64	1.1	64	0.7	40	0.6	32

We synchronise the replicas by executing a barrier at the start and end of each run. Relative standard deviations are $\leq 1\%$.

As expected, the replicas competing for memory bandwidth reduces observable throughput to roughly 50% for DMR and 33% for TMR on x86. On Arm, a single core cannot saturate the memory system, and this bandwidth reserve lessens the impact on throughput. Predictably, there is little difference between the LC and CC approaches.

B. System Benchmarks

We run Redis [46], a key-value store, as a system benchmark. Redis has a number of desirable features, such as exercising CPU, memory, and network. It is implemented in ANSI C without external dependencies, and adopts a single-threaded, event-driven design and thus saves us from analysing source code for data races. To avoid hiding overheads behind I/O latencies, we run Redis as a volatile store without persistence.

We run Redis as a native seL4 process, with a second process running the lwIP network stack [47] and an Ethernet driver. Due to the different handling of I/O interfaces, the drivers for the two configurations are slightly different. We investigate three configuration options for each setup, which differ in the effort put into detecting divergence:

No arguments (N): minimal effort, synchronise on I/O only (device register or DMA buffer access and interrupt handling);

Arguments (A): in addition, add all arguments to the signature on each syscall. This is the default version, as described in Section III-C;

Synchronise (S): as above, but also vote on each syscall.

Obviously, from N to S cost will increase but detection latency will decrease, so this represents a performance-safety trade-off.

We evaluate performance of the Redis server using the *Yahoo! cloud serving benchmarks* (YCSB) [48], running on dedicated load generator machines, connected to the evaluation platforms by dedicated Gigabit Ethernet links. We ensure that throughput is not limited by the load generators. For all runs we set `recordcount` to 70,000; we set `operationcount` to 10×`recordcount`, except for YCSB-E, where it is 1×`recordcount`. These settings result in a database size of around 160 MiB on Arm and 190 MiB on x86 (as reported by the `info memory` Redis client command), significantly larger than the last-level cache sizes. For each platform, we run the YCSB benchmark set 10 times. Error bars show standard deviations.

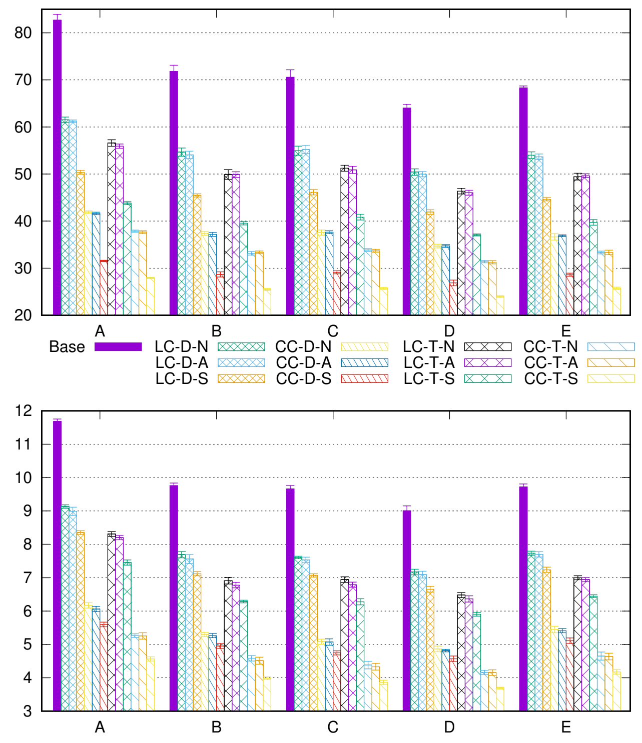

Fig. 3. Average Redis throughput (A–D: 1000 transactions per second, E: 20 transactions per second) on x86 (top) and Arm.

TABLE VI
REDIS THROUGHPUTS NORMALISED TO BASELINE.

	LC-RCoE		CC-RCoE	
Mode	Arm	x86	Arm	x86
D-N	78–80%	74–79%	53–56%	51–54%
D-A	77–79%	74–79%	52–56%	50–54%
D-S	72–75%	61–66%	48–53%	38–42%
T-N	71–72%	68–73%	45–48%	46–49%
T-A	69–71%	68–73%	45–48%	46–49%
T-S	63–66%	53–58%	39–43%	34–38%

Results are shown in Fig. 3, we summarise the performance degradation in Table VI. For readability we omit results for YCSB-F, which is very similar to A and always shows virtually indistinguishable results. We also observe that the results are remarkably similar for the two platforms, so we can examine them together without making reference to the processor.

Loosely-coupled DMR loses 20–38% throughput, the additional degradation of TMR is smaller, about an extra 15% over DMR. The additional cost of including syscall arguments in the signature ("A" vs "N") is negligible, which justifies using "A" as the default configuration. Voting on each system call ("S") has a higher cost impact, but that impact is less than the baseline cost of replication.

The overhead of the CC approach is significantly higher than LC. The cost is dominated by the need to move all device accesses into the kernel, significantly increasing the number of system calls. The extra cost of voting on system calls is comparatively small, indicated by the fairly moderate cost of reducing the error-detection latency.

C. Error Detection

We use software fault injection to test the ability of RCoE to detect errors, and also experiment with over-clocking.

1) Random memory faults: We run the Redis server from Section V-B and use a spare CPU core to flip random bits in memory. We modify the client to embed CRC32 checksums into the values sent to the store on the server. When the client reads back the values, it can detect data corruptions by comparing the embeded checksums and the recalcuated checksums of the return vaules. We run until the server fails, the client detects a checksum mismatch, or the error-detection mechanisms report an error, then restart the system and repeat.

The left-hand columns of Table VII show results on x86, using the default LC-A configuration of Section V-B. We target the memory of all kernel replicas, including the kernel shared region, as well as the *user memory of the primary replica*. When injecting faults we ensure the same sequence of pseudo-random numbers for all configurations by seeding the generator with the same number.

For the base case, we find that about 4% of injected faults lead to observed errors, the majority of which lead to corruption detected by the client or even the client faulting ("YCSB errors"), the remainder leading to faults in the server.

DMR and TMR are about equally effective in detecting errors, with both LC configurations failing in about 1% and CC in about 1.4% of cases to detect the error before leading to corrupt output or YCSB errors. The slightly higher failure rate of CC results from the slightly reduced SoR: The primary copies input data from its private DMA buffer to a shared one, from which the non-primaries replicate the data.

Uncontrolled errors are due to the following reasons: (1) The DMA buffers are outside the SoR. Data corruptions in input buffers cannot be detected if they do not lead to divergence, and data in output buffers can be corrupted after the replicas have voted (with `FT_Add_Trace`) and released the data into the output buffers. (2) If the primary becomes unresponsive, it cannot trigger synchronisations for interrupts, leading to a hang. (3) Errors in the shared kernel-memory regions can affect multiple replicas.

The *kernel exceptions* merit further scrutiny. An analysis of the logs of the LC-T configuration reveals that two of them were caused by corrupted kernel instructions, the third was potentially caused by a change of kernel-object type, resulting de-referencing an invalid pointer. The seL4 verification proves that there are no kernel exceptions (assuming correctly-functioning hardware) and the kernel halts if an exception is raised, so these are controlled errors.

In the right-hand columns of Table VII we show results obtained on Arm. Here we *inject faults into all replicas'*

TABLE VII
ERRORS RESULTING FROM MEMORY FAULT INJECTIONS ON X86 AND ARM.

	x86 – no exception-handler barriers					Arm – with exception-handler barriers						
	Base	LC-D	LC-T	CC-D	CC-T	Base	LC-D	LC-T	LC-D-N	LC-T-N	CC-D	CC-T
Injected faults	60k	91k	92k	86k	89k	243k	202k	184k	224k	214k	205k	185k
Observed errors	2297	2340	2340	2500	2500	1000	1000	1000	1000	1000	1000	1000
YCSB corrup	1001	11	16	22	13	647	3	1	381	299	3	0
YCSB errors	137	6	4	10	24	57	1	0	13	10	3	6
User mem faults	820	2	2	0	0	291	0	0	0	0	0	0
Other user faults	339	0	0	0	0	5	0	0	0	0	0	0
Kernel exceptions	0	0	3	3	2	0	0	0	0	0	0	0
Barrier timeouts	N/A	1238	1184	1632	1675	N/A	304	304	602	678	536	516
Signature mism	N/A	1083	1131	833	786	N/A	692	695	4	13	458	478
Uncontrolled	2297	19	22	32	37	1000	4	1	394	309	6	6
Controlled	N/A	2321	2318	2468	2463	N/A	996	999	606	691	994	994

memory, not just the primary. Furthermore, we avoid the kernel exceptions observed on x86 by *adding barriers to the kernel's exception handler*.

We observe that the failure rates are reduced in all four configurations, although the change is not statistically significant. Importantly, there are no kernel exceptions. According to the log, there were several kernel-data aborts, which were caught by barrier timeouts, indicating that this mechanism is effective.

We show a further configuration, labelled LC-D-N/LC-T-N, where we do not include checksums of output network packages in the state signature. It shows a dramatically increased failure rate of 39% for DMR and 31% for TMR, meaning the benefit of replication is dramatically reduced. This clearly demonstrates the importance of allowing drivers to contribute data into the state signature to trigger a vote.

We finally observe that the CC variants have about double the kernel-barrier timeout rates of the LC variants. CC-RCoE is sensitive to execution divergence, which is results in inconsistent numbers of branches, which can result in the replicas voting an incorrect leader.

2) CPU register faults: Standard fault-injection experiments target memory. However, cosmic rays are not particularly discriminatory and can also cause faults in registers, which are much harder to defend against.

To inject faults into user registers we utilise the fact that on an interrupt, the kernel preempts the running thread and saves its context. We pick a random bit in the saved user register state and flip it, then restore context and continue the user program. We only inject into the primary replica.

As workload we run the CPU-bound `md5sum` from BusyBox [49], which implements the MD5 [50] algorithm and produces a 128-bit hash value of a file. Secure hashes like MD5 are designed to be sensitive to random bit flips. We run MD5 [50] in a Linux VM on the base system, as well as a CC-RCoE DMR VM configuration.

For each run we generate a file filled with 128 MiB random data, and its fault-free digest. We then run `md5sum` in a loop, each iteration computing a digest and comparing it with the correct value. If the digests differ we count this as a data corruption, while we count abnormal termination as a crash.

At each run we inject faults until the digests differ, the application crashes, or CC-RCoE detects a divergence, up to a total of 2500 errors. Table VIII shows the results. We find that the CC-RCoE DMR VM setup is 100% effective in detecting divergence without ever producing corrupted outputs, with most corruption caught by voting on signatures, and 4% by timeouts resulting from control-flow corruption.

3) Physically-induced CPU faults: The standard practice of software fault injection has a degree of artificiality, as it can only reveal faults in explicitly targeted components, and may not be representative of SEUs produced by ionising radiation. Ideally this would be complemented by radiation experiments, but we were unable to access a suitable radiation source.

Instead we resort to overclocking the processor, which is known to degrade reliability [51], and is not limited to software-accessible parts of the system. We do not claim that faults resulting from overclocking are representative of SEUs,

TABLE VIII
REGISTER FAULT INJECTIONS ON `MD5SUM` ON X86.

	Base	CC-D
Total Injected	2872	2812
Crashes	887	0
Corruptions	1613	0
Timeouts	N/A	99
Mismatches	N/A	2401
Uncontrolled Errors	2500	0
Controlled Errors	0	2500

TABLE IX
ERRORS RESULTING FROM OVERCLOCKING THE ARM BOARD.

	Base	LC-D	LC-T
Observed errors	1000	1000	1000
User memory fault	632	0	0
User other exception	345	0	0
Kernel exceptions	2	0	0
YCSB corruptions	1	0	0
YCSB errors	20	25	24
Timeouts	N/A	724	853
Signature mismatches	N/A	251	123
Uncontrolled	1000	25	24
Controlled	N/A	975	976

only that they test a different scenario. In fact, overclocking is likely to first affect parts of the circuitry that are, as a result of random fluctuations in the manufacturing process, closest to the borders of design-tolerance windows. As such, overclocking is more likely to cause multiple faults in the same circuitry within a short period, which is a much more pessimistic scenario than the true randomness of SEUs.

We overclock our Arm system at 1.092 GHz, 9% above the manufacturer-specified maximum rate. Again we run the Redis benchmark of Section V-B, Table IX shows LC results.

We find quite a different error pattern compared to the Arm results of Table VII, with user-mode errors dominating. The system crashes very quickly, usually before entering the actual benchmarking stage. In the RCoE configurations, 2.5% of errors are not detected but lead to externally observable failures. Inspection of the logs reveal that in 6 of these 49 cases overclocking caused system reboots, the remaining 43 were network exceptions, indicating an unresponsive system.

The overclocking experiments reveal an important, although expected, limitation of our RCoE schemes: when multiple components experience errors within a short time window, the system may enter the state of complete failure that is beyond the capability of our software-implemented mechanisms, which are based on the assumption that the hardware functions mostly correctly. This kind of failure is easy to handle with a watchdog. Also, this scenario is unlikely to occur with radiation, unless it is very intense.

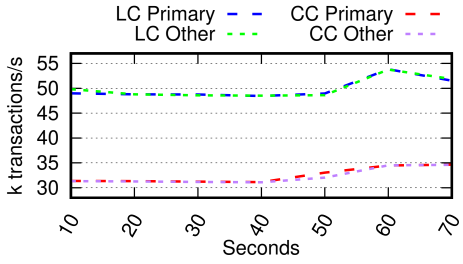

Fig. 4. Redis throughputs with error masking on x86.

TABLE X
TIME (MICROSECONDS) TAKEN FOR ERROR RECOVERY.

	LC primary	LC other	CC primary	CC other
x86	532	8	2,869	3
Arm	2,621	21	N/A	N/A

D. Error Recovery

We now examine a TMR system, running the Redis benchmark, recovering from a signature mismatch by downgrading to DMR. We distinguish between triggering an error in the primary, where the system needs to select a new primary, and another replica, where this step is not needed.

Fig. 4 shows reported YCSB-B throughput on x86 every 10 seconds. The graph shows a throughput *increase* after the error is triggered about 50 seconds after YCSB started. This is the result of the downgrade to the lower-overhead DMR. The actual recovery operation is not visible in the throughput graphs, nor do we see a performance difference between removing the primary or another replica.

Table X shows the measured cost for recovery on x86 as well as Arm (remember from Section IV-A that we cannot implement downgrading for CC on our 32-bit Arm platform). Removing the primary is two orders of magnitude more expensive than removing another replica, but the overall cost is still only about 2 ms, small enough to be virtually invisible from the outside.

VI. DISCUSSION

To understand how RCoE-provided fault tolerance compares to the use of radiation-hardened hardware, we compare our Cortex-A9 processor with the widely used RAD750 processor [52]. Running at 133 MHz, the RAD750 achieves around 240 Dhrystone 2.1 MIPS (DMIPS) and draws less than 6 W of power [53], or 40 DMIPS/W.

The Cortex-A9 achieves 2.50 DMIPS per MHz per core [54], or 2,000 DMIPS per core at 800 MHz. We get 2,000 DMIPS overall if we use three cores for TMR and leave the remaining core idle. Even if we pessimistically assume a factor-two performance overhead for TMR, the system can achieve 1000 DMIPS with a total power draw of 5 W, or 200 DMIPS/W. (Power is measured for the whole SABRE Lite board [55], which over-estimates the power draw of the processor.)

This is over five times the energy efficiency of the RAD750. Each RAD750 processor costs around $ 200,000 [56], while the price tag for a SABRE Lite board is around $ 200 [57]. The RAD750 processors can tolerate a more stringent operating environment in terms of temperature ranges and radiation intensities, but within the SABRE's operating range, RCoE clearly has strong SWaP and cost advantages.

In the current form, the highly-variable overhead suggest that CC-RCoE will add significant pessimism in the worst-case execution-time (WCET) analysis for hard real-time systems.

We observe that reducing the number of breakpoint exceptions when synchronising the replicas is vital to improve the performance CC-RCoE. We plan to explore using performance counter interrupts instead of breakpoints when a catching-up replica needs to cover a large number of branches, and switching to breakpoints when the remaining branch number is smaller than a predefined value [38]. Furthermore, for a particular system, we can reduce overheads by profiling offline to identify preemption points that are not in tight loops.

Although the physical memory is split and assigned to the replicas equally, several small memory regions are shared among the replicas to implement the RCoE framework (including barriers and checksums) harnessing the replicas and buffers for replicating input data (Section III-E). Errors in the framework region mostly result in barrier timeouts or checksum mismatches, but we have not performed a complete analysis of all the possible scenarios. Errors in the input buffers can cause silent data corruptions, manifested as data corruption errors in the Redis fault injection experiments.

VII. Related Work

There are a number of approaches to software-implemented fault tolerance in the literature, providing redundancy at the instructions, processes, or virtual-machine level.

SWIFT [58] is a compiler-based solution for Itanium 2 processors. It detects transient hardware faults using a modified compiler, which duplicates instructions in order to recompute results with different registers. SWIFT assumes memory and caches are protected by ECC so that store instructions are not replicated. HAFT [59] takes a hybrid approach, combining compiler-based instruction-level replication for error detection with Intel's TSX transactional memory support for error recovery. Like SWIFT, HAFT does not duplicate memory load/store instructions, assumes ECC memory, and also assumes that the Linux kernel operates correctly, a courageous assumption even in the absence of hardware faults [60]. Also, SEC-DED ECC memory is insufficient to protect modern memory systems [22].

Wang et al. [61] exploit multiple cores, with a research version of Intel's ICC 9.0 compiler automatically generating a pair of threads for each thread in source code; system calls are only executed by one thread. PLR [62] targets unmodified single-threaded binary applications by creating replicas at the process level with Pin [63], transparently forking replicas. Both assume a reliable OS.

Romain [64] is an OS service based on the Fiasco.OC microkernel. It replicates user-level processes assuming the kernel, device drivers and the replication framework to operate correctly. Ulbrich et al. [65] hardens crtical user-mode code with CoRed (combined redundancy), which combines TMR, data encoding, and control-flow encoding, to eliminate the single point of failure in software-based redundancy; these techniques for protecting user code complement our aims. Rex [66] proposes an execute-agree-follow model to efficiently replicate multithreaded applications on multicore servers. The model allows a primary replica to handle requests concurrently; non-deterministic decisions are recorded in traces. After all replicas agree on the traces by executing a consensus protocol, secondary replicas replay the traces concurrently to reach the same state as the primary.

Bressoud and Schneider [67] design protocols for coordinating non-deterministic event delivery for a hypervisor running on HP's PA-RISC architecture, enabling the hypervisor to manage a primary-backup virtual machine pair for fault tolerance. The approach relies on the PA-RISC processor's ability to deterministically deliver interrupts and on correct operation of the hypervisor.

Remus [68] aims for high availability by replicating the *protected* and *backup* virtual machines on a pair of physical hosts using the live-migration capability of the Xen virtual machine monitor to support fine-grained checkpoints, and relying on correct operation of Xen. The fault-tolerant feature of VMware vSphere 4.0 [69] runs primary and backup virtual machines in *virtual lockstep* on different physical machines, where the hypervisor, assumed to be reliable, manages the virtual CPU of the backup VM to execute the same instructions committed by the primary VM. A logging channel is used to transmit input data and nondeterministic events captured by the primary VM to the backup VM, which applies the data and replays the events deterministically. These fault-tolerant systems based on virtual machines assume that the kernel or hypervisor is not affected by hardware faults.

FT-Linux [70] is the only other system in the literature (although predated by LC-RCoE [23]) that replicates virtually the complete software stack without hardware support. It implements a full-stack, primary-backup, fault-tolerant Linux system on a single machine by partitioning hardware resources, instantiating two Linux kernels, and replicating OS services as well as selected applications. Non-deterministic events are logged on the primary and replayed on the secondary. Failure detection is achieved by interchanging heartbeat messages between the Linux kernels and also relying on hardware error-detection features. The replicas managed by RCoE sychronise before they observe non-deterministic events, removing the latency of recording and replaying non-deterministic events and thus extending the SoR.

VIII. Conclusions and Future Work

Our results show that it is feasible to provide protection against random hardware faults, by redundantly executing a complete software stack on commodity multicore processors. Without non-standard hardware support we can replicate everything except low-level device access. Specifically, replicating applications is transparent: we do not have to modify user-mode code other than drivers and for porting to seL4. This paper introduces CC-RCoE to overcome some limitations of LC-RCoE, significantly extending the range of supported applications.

Our evaluation shows that while performance cost are noticeable, we can trade them against the latency of error detection, by choosing voting frequency, and deciding on how much state to accumulate into the state signatures which the kernel replicas use for voting.

Compared to other software approaches, which only protect selected applications and rely on the kernel not being affected by faults, we dramatically extend the sphere of replication to include practically the complete system.

Current RCoE can only replicate a logical single-core system. With increasing core counts in commodity processors, it is now feasible (and desirable) to replicate multicore systems on a single processor. Furthermore we found that a significant portion of errors is detected by barrier timeouts (Section V-C), recovering from those would be beneficial. Finally we would like to investigate how we can provide real-time guarantees with RCoE.

Acknowledgement

The authors would like to thank the shepherd, Sameh Elnikety, and anonymous reviewers for their feedback and suggestions.

Availability

Source code for our RCoE implementations, as well as evaluation rigs and complete raw data sets are available for download from https://trustworthy.systems/projects/TS/cots.pml.

References

[1] D. Bernick, B. Bruckert, P. Del Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen, "NonStop advanced architecture," in *Proceedings of the 35th International Conference on Dependable Systems and Networks (DSN)*, Washington, DC, US, 2005, pp. 12–21.

[2] J. Bartlett, W. Bartlett, R. Carr, D. Garcia, J. Gray, R. Horst, R. Jardine, D. Lenoski, and D. McGuire. (1990) Fault tolerance in Tandem computer systems. [Online]. Available: http://www.hpl.hp.com/techreports/tandem/TR-90.5.pdf

[3] L. Spainhower and T. A. Gregg, "IBM S/390 parallel enterprise server G5 fault tolerance: A historical perspective," *IBM Journal of Research and Development*, vol. 43, no. 5, pp. 863–873, Sep. 1999.

[4] W. Bartlett and L. Spainhower, "Commercial fault tolerance: A tale of two systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 87–96, Jan. 2004.

[5] M. N. Sweeting, "Modern small satellites—changing the economics of space," *Proceedings of the IEEE*, vol. 106, pp. 343–361, Mar. 2018.

[6] J. F. Ziegler and W. A. Lanford, "Effect of cosmic rays on computer memories," *Science*, vol. 206, no. 4420, pp. 776–788, 1979. [Online]. Available: http://science.sciencemag.org/content/206/4420/776

[7] R. Baumann, "Technology scaling trends and accelerated testing for soft errors in commercial silicon devices," in *9th IEEE On-Line Testing Symposium*, Jul. 2003, pp. 4–.

[8] ——, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Devices and Materials Reliability*, vol. 5, no. 3, pp. 305–316, Sep. 2005.

[9] N. Seifert, P. Slankard, M. Kirsch, B. Narasimham, V. Zia, C. Brookreson, A. Vo, S. Mitra, B. Gill, and J. Maiz, "Radiation-induced soft error rates of advanced CMOS bulk devices," in *Proceedings of the 44th IEEE International Reliability Physics Symposium*, San Jose, CA, US, Mar. 2006, pp. 217–225.

[10] V. Ferlet-Cavrois, L. W. Massengill, and P. Gouker, "Single event transients in digital CMOS – a review," *IEEE Transactions on Nuclear Science*, vol. 60, no. 3, pp. 1767–1790, Jun. 2013.

[11] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an OS microkernel," *ACM Transactions on Computer Systems*, vol. 32, no. 1, pp. 2:1–2:70, Feb. 2014.

[12] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich, "Using Crash Hoare logic for certifying the FSCQ file system," in *ACM Symposium on Operating Systems Principles*, Monterey, CA, Oct. 2015, pp. 18–37.

[13] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill, "Ironclad apps: End-to-end security via automated full-system verification," in *USENIX Symposium on Operating Systems Design and Implementation*, Broomfield, CO, US, Oct. 2014, pp. 165–181. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/hawblitzel

[14] S. Chen, J. Xu, Z. Kalbarczyk, R. K. Iyer, and K. Whisnant, "Modeling and evaluating the security threats of transient errors in firewall software," *Performance Evaluation*, vol. 56, no. 1–4, pp. 53–72, Mar. 2004.

[15] J. Xu, S. Chen, Z. Kalbarczyk, and R. K. Iyer, "An experimental study of security vulnerabilities caused by errors," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2001, pp. 421–430.

[16] S. Govindavajhala and A. W. Appel, "Using memory errors to attack a virtual machine," in *IEEE Symposium on Security and Privacy*, 2003, pp. 154–165.

[17] X. Li, K. Shen, M. C. Huang, and L. Chu, "A memory soft error measurement on production systems," in *Proceedings of the 2007 USENIX Annual Technical Conference*, Santa Clara, CA, US, 2007.

[18] S. Z. Shazli, M. Abdul-Aziz, M. B. Tahoori, and D. R. Kaeli, "A field analysis of system-level effects of soft errors occurring in microprocessors used in information systems," in *2008 IEEE International Test Conference*, Oct. 2008, pp. 1–10.

[19] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: A large-scale field study," in *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems*, Seattle, WA, US, 2009, pp. 193–204.

[20] 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 6th EuroSys Conference*, Salzburg, AT, Apr. 2011.

[21] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi, "Feng shui of supercomputer memory: Positional effects in DRAM and SRAM faults," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Denver, CO, US, 2013.

[22] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory errors in modern systems: The good, the bad, and the ugly," in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, Istanbul, TR, Mar. 2015, pp. 297–310.

[23] Y. Shen and K. Elphinstone, "Microkernel mechanisms for improving the trustworthiness of commodity hardware," in *European Dependable Computing Conference*, Paris, France, Sep. 2015, p. 12.

[24] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *Proceedings of the 27th International Symposium on Computer Architecture*, Jun. 2000, pp. 25–36.

[25] T. C. May and M. H. Woods, "Alpha-particle-induced soft errors in dynamic memories," *IEEE Transactions on Electron Devices*, vol. 26, no. 1, pp. 2–9, 1979.

[26] K. P. Rodbell, D. F. Heidel, H. H. K. Tang, M. S. Gordon, P. Oldiges, and C. E. Murray, "Low-energy proton-induced single-event-upsets in 65 nm node, silicon-on-insulator, latches and memory cells," *IEEE Transactions on Nuclear Science*, vol. 54, no. 6, pp. 2474–2479, Dec. 2007.

[27] B. D. Sierawski, M. H. Mendenhall, R. A. Reed, M. A. Clemens, R. A. Weller, R. D. Schrimpf, E. W. Blackmore, M. Trinczek, B. Hitti, J. A. Pellish, R. C. Baumann, S.-J. Wen, R. Wong, and N. Tam, "Muon-induced single event upsets in deep-submicron technology," *IEEE Transactions on Nuclear Science*, vol. 57, no. 6, pp. 3273–3278, Dec. 2010.

[28] M. P. King, R. A. Reed, R. A. Weller, M. H. Mendenhall, R. D. Schrimpf, B. D. Sierawski, A. L. Sternberg, B. Narasimham, J. K. Wang, E. Pitta, B. Bartz, D. Reed, C. Monzel, R. C. Baumann, X. Deng, J. A. Pellish, M. D. Berg, C. M. Seidleck, E. C. Auden, S. L. Weeden-Wright, N. J. Gaspard, C. X. Zhang, and D. M. Fleetwood, "Electron-induced single-event upsets in static random access memory," *IEEE Transactions on Nuclear Science*, vol. 60, no. 6, pp. 4122–4129, Dec 2013.

[29] NEC. (2011, Mar.) Fault tolerant server white paper. [Online]. Available: http://www.nec.com/en/global/prod/express/collateral/whitepaper/ft_WhitePaper_E.pdf

[30] C. M. Fuchs, T. P. Stefanov, N. M. Murillo, and A. Plaat, "Bringing fault-tolerant Gigahertz-computing to space: A multi-stage software-side fault-tolerance approach for miniaturized spacecraft," in *2017 IEEE 26th Asian Test Symposium (ATS)*, Nov. 2017.

[31] BAE. (2018) Radiation-hardened processors products. [Online]. Available: https://www.baesystems.com/en/our-company/our-businesses/electronic-systems/product-sites/space-products-and-processing/processors

[32] D. D. Corporation. (2018) SCS750 single board computer for space. [Online]. Available: http://www.ddc-web.com/Products/Microelectronics/images/documents/SCS750_rev8_r6.pdf

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

[34] J. M. Mellor-Crummey and T. J. LeBlanc, "A software instruction counter," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 1989, pp. 78–86.

[35] J. G. Fletcher, "An arithmetic checksum for serial transmissions," *IEEE Transactions on Communications*, vol. 30, no. 1, pp. 247–252, Jan. 1982.

[36] V. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," in *IEEE International Symposium on Performance Analysis of Systems and Software*, Apr. 2013.

[37] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide*, Intel Corp., 2016, https://software.intel.com/en-us/articles/intel-sdm.

[38] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Revirt: Enabling intrusion analysis through virtual-machine logging and replay," in *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, US, 2002.

[39] J. H. Slye and E. N. Elnozahy, "Supporting nondeterministic execution in fault-tolerant systems," in *Proceedings of the 26th International Symposium on Fault-Tolerant Computing*, Jun. 1996, pp. 250–259.

[40] Y. Shen, "Microkernel mechanisms for improving the trustworthiness of commodity hardware," Ph.D. dissertation, UNSW, Mar. 2019.

[41] *i.MX 6Dual/6Quad Applications Processor Reference Manual*, NXP, 2015, http://www.nxp.com/assets/documents/data/en/reference-manuals/IMX6DQRM.pdf.

[42] R. P. Weicker, "Dhrystone benchmark: Rationale for version 2 and measurement rules," *SIGPLAN Notices*, vol. 23, no. 8, pp. 49–62, Aug. 1988.

[43] R. Painter, "C converted Whetstone double precision benchmark," http://www.netlib.org/benchmark/whetstone.c, 1998.

[44] Buildroot, "Buildroot," 2018. [Online]. Available: https://buildroot.org

[45] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proceedings of the 22nd International Symposium on Computer Architecture*. S. Margherita Ligure, IT: ACM, 1995, pp. 24–36.

[46] RedisLabs. (2009) Redis. [Online]. Available: https://redis.io

[47] A. Dunkels, "Minimal TCP/IP implementation with proxy support," SICS, Tech. Rep. T2001-20, Feb. 2001, http://www.sics.se/~adam/thesis.pdf.

[48] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *ACM Symposium on Cloud Computing*. Indianapolis, IN, US: ACM, Jun. 2010, pp. 143–154.

[49] BusyBox, "BusyBox: The Swiss army knife of embedded Linux," https://busybox.net/, 2017.

[50] R. Rivest, "The MD5 message-digest algorithm," Internet Requests for Comments, Internet Engineering Task Force, RFC 1654, Apr. 1992. [Online]. Available: https://tools.ietf.org/html/rfc1321

[51] G. Memik, M. H. Chowdhury, A. Mallik, and Y. I. Ismail, "Engineering over-clocking: reliability-performance trade-offs for high-performance register files," in *Proceedings of the 35th International Conference on Dependable Systems and Networks (DSN)*, Jun. 2005, pp. 770–779.

[52] BAE Systems. (2016) Rad750® radiation-hardened PowerPC microprocessor. [Online]. Available: https://www.baesystems.com/en-us/download-en-us/20161103152954/1434555668211.pdf

[53] R. W. Berger, D. Bayles, R. Brown, S. Doyle, A. Kazemzadeh, K. Knowles, D. Moser, J. Rodgers, B. Saari, D. Stanley, and B. Grant, "The RAD750™ - a radiation hardened PowerPC™ processor for high performance spaceborne applications," in *2001 IEEE Aerospace Conference Proceedings (Cat. No.01TH8542)*, vol. 5, Mar. 2001, pp. 2263–2272.

[54] ARM. (2009) The ARM Cortex-A9 processors. [Online]. Available: https://web.archive.org/web/20120522214159/http://www.arm.com:80/files/pdf/ARMCortexA-9Processors.pdf

[55] (2011) SABRE Lite hardware user manual. [Online]. Available: https://boundarydevices.com/SABRE_Lite_Hardware_Manual_rev11.pdf

[56] R. Ginosar, "Survey of processors for space," in *Proceedings of DASIA 2012, data systems in aerospace*, May 2012.

[57] BD-SL-i.MX6 development board. [Online]. Available: https://boundarydevices.com/product/sabre-lite-imx6-sbc/

[58] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *Proceedings of the 3rd IEEE Symposium on Code Generation and Optimization*, 2005, pp. 243–254.

[59] D. Kuvaiskii, R. Faqeh, P. Bhatotia, P. Felber, and C. Fetzer, "HAFT: Hardware-assisted fault tolerance," in *Proceedings of the 11th EuroSys Conference*, London, UK, Apr. 2016.

[60] S. Biggs, D. Lee, and G. Heiser, "The jury is in: Monolithic OS design is flawed," in *Asia-Pacific Workshop on Systems (APSys)*. Korea: ACM SIGOPS, Aug. 2018, Conference Paper - Refereed.

[61] C. Wang, H. S. Kim, Y. Wu, and V. Ying, "Compiler-managed software-based redundant multi-threading for transient fault detection," in *Proceedings of the 5th International Symposium on Code Generation and Optimization*, 2007, pp. 244–258.

[62] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors, "PLR: A software approach to transient fault tolerance for multicore architectures," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 2, pp. 135–148, Apr. 2009.

[63] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2005, pp. 190–200.

[64] B. Döbel, H. Härtig, and M. Engel, "Operating system support for redundant multithreading," in *Proceedings of the 12th International Conference on Embedded Software*, Tampere, SF, Oct. 2012, pp. 83–92.

[65] P. Ulbrich, M. Hoffmann, R. Kapitza, D. Lohmann, W. Schröder-Preikschat, and R. Schmid, "Eliminating single points of failure in software-based redundancy," in *Ninth European Dependable Computing Conference*, May 2012, pp. 49–60.

[66] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang, "Rex: Replication at the speed of multi-core," in *Proceedings of the 9th EuroSys Conference*, Amsterdam, NL, Jan. 2014.

[67] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," *ACM Transactions on Computer Systems*, vol. 14, pp. 80–107, 1996.

[68] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, US, 2008.

[69] D. J. Scales, M. Nelson, and G. Venkitachalam, "The design of a practical system for fault-tolerant virtual machines," *ACM Operating Systems Review*, vol. 44, no. 4, pp. 30–39, Dec. 2010.

[70] G. Losa, A. Barbalace, Y. Wen, M. Sadini, H.-R. Chuang, and B. Ravindran, "Transparent fault-tolerance using intra-machine full-software-stack replication on commodity multicore hardware," in *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems*, Jun. 2017, pp. 1521–1531.