# Microkernel Mechanisms for Improving the Trustworthiness of Commodity Hardware

Yanyan Shen
NICTA and UNSW
yanyan.shen@nicta.com.au

Kevin Elphinstone
NICTA and UNSW
kevin.elphinstone@nicta.com.au

***Abstract*—Trustworthy isolation is required to consolidate safety and security critical software systems on a single hardware platform. Recent advances in formally verifying correctness and isolation properties of a microkernel should enable mutually distrusting software to co-exist on the same platform with a high level of assurance of correct operation. However, commodity hardware is susceptible to transient faults triggered by cosmic rays, and alpha particle strikes, and thus may invalidate the isolation guarantees, or trigger failure in isolated applications.**

**To increase trustworthiness of commodity hardware, we apply redundant execution techniques from the dependability community to a modern microkernel. We leverage the hardware redundancy provided by multicore processors to perform transient fault detection for applications and for the microkernel itself. This paper presents the mechanisms and framework for microkernel-based systems to implement redundant execution for improved trustworthiness. It evaluates the performance of the resulting system on x86-64 and ARM platforms.**

***Keywords—Microkernel, Security, Reliability, SEUs, Trustworthy Systems***

## I. Introduction

Security-critical and safety-critical systems require rigorous design and implementation approaches to provide a high degree of assurance that they meet their application requirements. Conventional wisdom is that even with rigorous development processes, software bugs are inevitable in large complex systems, which leads to solutions based on separate or redundant hardware for information isolation or fault isolation. An *air gap* is clearly a trustworthy isolation approach.

It has long been envisioned that trustworthy isolation implemented in software could be used for security or safety critical systems. The aim of *separation kernels* is to isolate applications by creating "an environment which is indistinguishable from that provided by a physically distributed system" [1], with applications communicating only through the channels explicitly provided by the separation kernel. The *multiple independent levels of security* (MILS) approach also uses a separation kernel to consolidate previously disparate systems onto a single hardware platform [2]. Virtualisation and virtual machine monitors (VMMs) can also be viewed as platforms for consolidating multiple applications (in this case operating systems) while retaining isolation for security or reliability concerns [3], [4], [5].

These approaches share the common aim of reducing exposure to operating system (OS) bugs by reducing the amount of privileged code responsible for the software platform's integrity and isolation. The approaches are at least philosophically aligned with microkernels in regard to espousing a trustworthy core extended by user-level (i.e. independent and isolated) services and applications [6]. However, these approaches also share a common vulnerability — the overall system security or safety is only as trustworthy as the underlying OS, be it a separation kernel or a VMM.

If truly small microkernels are applied as separation kernels or VMMs, then an opportunity exists to construct trustworthy systems using software isolation (a virtual air gap). Recent advances in system software verification and kernel design have resulted in a microkernel (specifically, the seL4 microkernel) being mathematically proven to be bug-free [7], and thus can be trusted to implement its specified behaviour. Furthermore, additional proofs provide guarantees of integrity and isolation of applications [8], [9], [10], with proof properties extending to the microkernel binary itself, thus removing the compiler as a source of defects [11].

We expect a general approach to constructing trustworthy software systems to evolve from the recent system software verification work. The vision is a software stack consisting of software components ranging from fully formally verified components (i.e. trustworthy) to untrusted or malicious components, separated by the strong isolation provided by the verified microkernel, thus forming a trustworthy system. We define trustworthy as having a high degree of confidence in correct operation of the trusted components of the system.

However, we believe there are still barriers to adoption of verified microkernels and components in safety and security critical situations, as the proofs make the assumption that the hardware itself behaves as specified in the proof. Any deviation in behaviour of the hardware invalidates any guarantees derived from formal verification. The focus of this paper is to obtain extra assurance of correct system behaviour in the presence of transient faults in commodity hardware.

Transient faults are not necessarily related to (in)correct hardware design or manufacture. Transient faults can be triggered by single event upsets (SEUs), that result from alpha particle strikes or cosmic rays [12], [13], [14]. Transient faults also occur as commodity hardware ages [15]. Transient faults can manifest themselves in almost any way as a result of fault propagation if they occur within the OS kernel itself [16]. Studies have shown that transient faults can introduce security vulnerabilities in the Linux kernel, network services, and in virtual machines. The vulnerabilities can continue to exist until the system is rebooted, leaving the potential for a SEUs to leave a system vulnerable for long periods of time [17], [18], [19]. A recent study [20] demonstrated that commodity DRAM chips are vulnerable to disturbance errors. Malicious programs may be able to corrupt data in a row by reading nearby rows repeatedly to create an exploit, as demonstrated by the *row hammer* attack [21].

Our work's goal is to address two issues: (1) the lack of assurance (or indication) of error free operation of commodity hardware, and (2) the threat of a transient fault creating a vulnerability that can be exploited to violate isolation or circumvent the security or safety policy of the system. Our motivating use-case is high-assurance embedded systems where power, weight and cost are issues and the risk profile does not warrant the expense or inflexibility of bespoke hardware solutions. Additionally, our initial target is application domains where reboot is a sufficient recovery mechanism (e.g. a secure firewall).

Existing redundant execution approaches that address system reliability (and thus our two goals) rely on either specialised hardware or make assumptions about the correct behaviour of the underlying OS, i.e. the lower-level privileged software layer is assumed unaffected by SEUs. We elaborate further in Section II.

We address these two issues by leveraging the redundancy available in modern multicore processors to create a framework for co-operative self-checking of replicated execution of a system *that includes the underlying operating system itself.*

### A. Overview

Our approach takes a single core system and replicates it to form replicated state machines [22]. Unlike pure replicated state machines, replicas have non-identical behaviour when managing non-redundant hardware. We introduce the concept of a compressed trace inside the kernel of each replica to capture a subset of the evolution of the state of each replica. We exclude (and aim to minimise) the non-identical behaviour of replicas to ensure the compressed trace deterministically evolves in each replica and thus if the traces are found inconsistent on comparison, an error has occurred in a replica and remedial action can be taken (fail-stop in our case).

Non-redundant hardware devices are handled by the introduction of a small number of microkernel API changes that enable a framework for redundant device drivers of non-redundant devices to co-ordinate access and hide device non-determinism from the rest of the system.

We have constructed a prototype system using the seL4 microkernel on both an ARM Cortex A9 and an Intel x86-64 platform. We evaluate the performance impact of the addition of trace management and redundant execution on several benchmarks. We also perform fault injection experiments to demonstrate that our system can detect transient faults in replicas.

Our contributions are as follows.

- A set of microkernel mechanisms together with a framework for co-operative self-checking using redundant execution of a uniprocessor system for increased assurance of correct operation. The framework allows system designers to trade latency of fault detection against the overhead due to frequency and coverage of self-checking, to match the system's risk profile.
- An approach to supporting redundant device drivers that manage non-redundant devices.
- A performance evaluation of the approach showing a modest performance penalty in return for increased assurance of correct operation.
- A fault injection evaluation demonstrating our framework's ability to detect errors resulting from transient faults, and effectively convert uncontrolled failures into graceful fail-stops.

## II. Related Work

### A. Hardware

Tolerating hardware faults by using isolated and redundant processors, buses, memory modules and I/O peripherals is a mature approach in safety critical computer systems [23], [24], [25]. Faults can be detected by dual modular redundancy, or masked by triple modular redundancy. Redundant results are *voted* on to form the final output. Hardware may be radiation-hardened or feature self-checking or self-correcting circuity to improve reliability.

Commercial high-availability servers also use DMR (dual-modular redundancy) or TMR (triple-modular redundancy) in locked-step or loosely-coupled configurations for improved availability. The IBM G5 processor has replicated pipelines. Each instruction is executed independently and the results are compared before being committed [26], [27]. NEC combines a special chipset and redundant Intel Xeon processors to allow the processors to run in locked-step mode, enabling the detection of errors and initiation of recovery [28]. The designers of the NonStop Advanced Architecture [29] (NSAA) recognise that running commodity processors in locked-step is more challenging because of dynamic core frequency scaling, the increasing CPU frequency, and the fact that multicore processors may not expose individual cores through sockets. The NSAA loosely lock-steps processors to allow redundant instruction streams execute at different rates. The processors are connected to self-checking logical synchronisation units where the processor I/O outputs are compared for fault detection and masking.

Our goal differs from these hardware-based approaches in that we target commodity multicore processor platforms which feature very few hardware-based fault tolerance approaches (e.g. no ECC memory and non-redundant devices). Our approach relies only on the availability of multiple cores, but can theoretically take advantage of hardware-based fault detection or correction if available.

### B. Software

Prior to discussing software approaches to improving reliability in the presence of transient faults, we first re-introduce the concept of the *sphere of replication* (SoR) [30]. Computation within the SoR forms a finite state machine with transient fault detection due to redundant execution. Computation outside the SoR does not possess fault detection and must either rely on fault propagation to the SoR, or be covered by other techniques, or become an area of vulnerability to transient faults.

Software-based approaches to redundant execution can be broadly classed into compiler-based, OS or runtime-based, and hypervisor-based approaches. We examine exemplars of each class in terms of their SoR and relate them to our goal of including the OS-itself within the SoR.

The SWIFT project modifies the compiler to insert redundant computation together with result and control flow checking [31]. SWIFT avoids replicating memory by assuming fault-free (e.g. ECC) memory and caches. Framing SWIFT in terms of a SoR, general computation is within the SoR, the checking code and memory are not. Thus SWIFT's surface of vulnerability includes writes to memory in addition to the checking code itself. No mention of OS-level SoR issues are discussed.

SWIFT was evaluated on an Itanium 2 which has 128 registers for compiler use. The performance penalty will be higher on more register-constrained architectures [32]. Our goal is to support commodity hardware which does not necessarily feature ECC memory, and thus we cannot simply assume memory can be omitted from the SoR.

The ubiquity of multicore processors provides an opportunity to utilise redundant computation across multiple cores for improved reliability. A compiler and runtime approach is to modify the compiler to create redundant threads that replicate computation within the SoR, one leading thread and one trailing [32]. The SoR excludes shared memory, system calls, and the OS itself.

Process-level redundancy [33] also exploits multicore processors by instantiating multiple instances of an application on different cores, i.e., the SoR is the process itself. Input replication and output comparisons are all conducted at the system call level. The approach does not support non-deterministic events and assumes correct OS operation.

Romain is an operating system service on a microkernel that also replicates program execution [34]. A master process initialises environments and creates process replicas. The master also handles CPU exceptions triggered by replicas, and the exception handling code compares the states of the replicas. Romain is broadly similar to our approach except that the microkernel itself, the user-level device drivers, and the Romain service lie outside of the SoR and are assumed to be protected by other measures, such as a hardened processor in a heterogeneous multicore system. We make no such assumption and include our kernel in the SoR.

Reliable virtual machine hosting has obvious parallels with reliable application hosting with the hypervisor creating redundant guest OSs to improve reliability. One approach is to record non-deterministic events injected into a primary guest and replay the same events to a backup guest on the same host or a backup host [35], [36]. These record-replay approaches focus on availability (i.e. fail-over), not error detection, and thus in the presence of transient faults, the primary can propagate erroneous results, or the primary and backup may diverge undetected. The hypervisors themselves are also outside the SoR.

Small hypervisors, such as SecVisor, can guarantee the integrity of the guest operating system [37]. SecVisor's goal is to defend against guest OS exploitation via OS modification, not detect hardware transient faults. It does not feature a SoR for the guest OS or itself. TinyChecker aims to improve reliability in the face of software faults via failure detection using a small hypervisor [38] as the foundation of nested virtualisation. The authors do not consider hardware faults, nor does the SoR include the TinyChecker itself.

To summarise, existing approaches either require specialised hardware or exclude the lowest-level OS from the SoR. Our aim is to include the lowest-level OS and device drivers within the SoR on commodity hardware in order to improve the trustworthiness of the underlying microkernel.

## III. seL4 Background

We now outline seL4's most relevant characteristics. seL4 has been described in detail previously [39], with detailed documentation also available [40].

The seL4 microkernel is a uniprocessor event-based kernel, i.e., the microkernel is not preemptable except for a few strategically placed preemption points where pending interrupts are polled. The lack of concurrency support stems from the challenges in verification of concurrent software. The gain from this restriction is that seL4 has been formally verified to be correct, i.e. bug free [41]. Interrupt latency of such a design can be kept within tight bounds with careful design [42]. Low-level interrupt handling is either triggered by an exception while at user-level, or by an observation of a pending interrupt at fixed points within the microkernel.

User-level applications are supported via microkernel primitives to create address spaces, direct the mapping of physical frames, create threads and assign them to address spaces, thus forming a process-like abstraction under the direction of a user-level OS personality. Processes form system services and their client applications, which interact via interprocess communication (IPC).

Authority is conveyed to applications via in-kernel (i.e segregated) capabilities. A seL4-based system starts with a single *root task* which possesses all the authority in the system. The root task initialises the remainder of the system according to the application domain, and may become the OS personality for the system itself.

The seL4 microkernel (and L4-kernels in general) feature user-level device drivers. Drivers are just applications providing a device service via IPC to other higher-level services (e.g. a TCP/IP stack), or drivers maybe directly incorporated into a user-level system service. Drivers access hardware devices either via memory-mapped I/O or via access to legacy port-based I/O (via system calls in seL4's case). Drivers receive interrupts via an IPC-based notification mechanism similar to a binary semaphore. The precise details are not important for our discussion except the observation that device driver interrupt handlers only receive notification of a pending interrupt when an interrupt handler waits for a notification, i.e. at a fixed point in their execution.

The microkernel does not have an API to access time. Systems can use timer device drivers as part of their OS personalities to provide an API to access time related services.

## IV. Approach

In this section we discuss our approach in detail. We divide the discussion into system replication, the compressed trace, kernel state consistency, and device drivers. The microkernel changes required to support our approach are highlighted in each section. We also highlight important properties of the microkernel relied on for our approach.

### A. *Replication*

The initialisation of L4-based systems (seL4 included) consists of loading a boot image containing the microkernel and an initial user-level *root task* that is responsible for starting the remainder of the system from executables and data also contained within the image. The in-kernel initialisation only consists of initialising the microkernel itself, creating the root task, and giving the root task access to the in-memory boot image. The majority of the system is started by the root task. A system can vary from a complete set of applications and user-level drivers loaded from the image, or it may be a minimal set of drivers and applications that effectively chain-loads a more complex system from a disk or a network.

To initialise a replicated system as illustrated in Figure 1, the seL4 in-kernel initialisation executes the following steps (our discussion assumes DMR for clarity).

- Physical memory is partitioned according to the number of replicas desired, but is limited by the number of available cores.
- The kernel code and boot image are copied into each partition.
- The remaining cores are brought online.
- All cores synchronise on a barrier[1] prior to the first kernel exit.

The replicated root task then creates identical replicated systems (replicated state machines) on each core. The root task starts in the same initial state on each core, and the root task's initialisation steps are deterministic.

**Microkernel Change:** Microkernel, root task, and boot image replicated on each core partition.

The root task determinism follows from a property of the seL4 API. The API does not directly expose physical addresses to applications. While the microkernel itself sees the different physical addresses of each partition, its behaviour is not a function of the specific addresses themselves. Applications manipulate the system through capabilities to kernel objects, which provide a level of indirection between identifiers in the API and actual physical addresses. Where addresses are required in the API, they are specified as offsets within a capability-specified memory region.

The lone exception to this restriction is that all root tasks receive a map of the physical memory available in the first partition as part of their initial state. See Section IV-D4 for

[1]We use *barrier* to denote a synchronisation primitive where all cores involved wait until they all reach the same execution point prior to progressing.

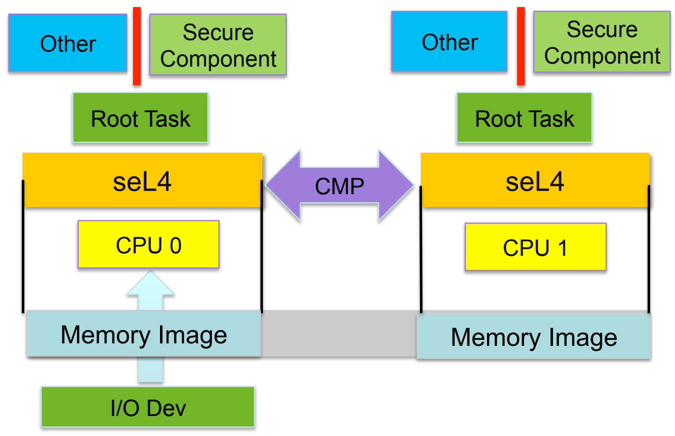

Fig. 1. System Replication for Redundant Execution

a discuss of physical addresses and DMA-based user-level drivers.

**Microkernel Property:** Microkernel API does not expose physical addresses to applications.

### B. *Compressed Trace*

The most significant microkernel change is the addition of a per-replica (per-core) compressed trace that captures events on each core. We define an *event* very generally as a value at an instance in the trace. By instrumenting the microkernel, the sequence of events captures a subset of the evolution of the microkernel's state.

Each replica's trace consists of a *count* and a *value*. The count tracks the number of events observed by the replica, and the value is a running checksum of the values associated with each event. We use Fletcher's checksum for the running checksum as it is dependent on both the values and the order they are observed [43].

The compressed trace is managed via the following microkernel-internal API.

- `add_event(value)` adds an event to the local per-core trace. It is an entirely local to the core operation requiring no cross-core synchronisation.
- `add_compare_event(value)` adds an event to the local trace and waits until all cores reach the same point in the trace and then compares the current state. A mismatch implies the replicas are inconsistent due to a fault, and our prototype triggers a graceful fail-stop. Our implementation requires two barriers to coordinate the comparison.
  `add_event(value)` and `add_compare_event(value)` are interchangeable with the former being a fast local operation and the latter forcing an immediate trace comparison. This creates a trade-off between performance and frequency of trace comparison for system designers to resolve as required. See Section IV-D for our prototype's primary use case in the context of device drivers.
- `trigger_action(action)` flags a pending action for all replicas to handle. *Actions* are asynchronous activities that can not be delivered at precisely the same instant across

all replicas. For instance, device interrupts are directed to core 0, and are propagated via inter-processor interrupts (IPIs). By flagging an action as pending for each replica, all replicas can perform the action at a consistent point in the trace. This primitive does not directly contribute to the trace, only indirectly via `handle_action()`.

- `handle_action()` performs the pending action when the following pre-conditions hold: (1) all replicas are within `handle_action()`; (2) all replicas are at the same point in the trace; and (3) all replicas have observed the action is pending. If the pre-conditions do not hold, the leading replica waits until the trailing replicas reach the same point in the trace, upon which the pre-conditions can be satisfied and the action performed.
  The implementation is more complex than alluded to, as a naive implementation can deadlock due to the timing of visibility of flags to replicas, resulting in some replicas waiting on a barrier in `add_compare_event()`, and others waiting in `handle_action()` . We avoid the issue with the aid of a conditional barrier in `handle_event()` that can be aborted by the barriers in the other primitives to allow replicas to synchronise traces.

The software barrier implementation features a timeout. If replicas diverge in their control flow because of a transient fault, replicas may wait forever in a barrier. The timeout value needs to exceed the maximum execution time a replica may validly trail the leading replica to avoid triggering false positives. We choose a conservative timeout of 2 seconds, which upon expiration, the system is assumed to have become inconsistent and a graceful fail-stop is triggered.

**Microkernel Change:** A compressed trace of events is added to the microkernel.

## C. In-kernel Trace Contributions

In this section we describe the relationship between the microkernel and the compressed trace.

*1) Non-determinism:* The primitives that manage the trace assume they are invoked deterministically with the same result across all replicas, with the exception being `trigger_action()`. Thus care must be taken to prevent non-determinism that is visible in a small fraction of the microkernel from contributing to the trace and causing it to diverge.

Non-determinism is visible directly and indirectly as a result of hardware triggered interrupts. Noting that interrupts are disabled within the microkernel itself, the non-determinism is visible directly due to hardware interrupts being delivered differently to individual cores (core 0 receives the interrupt and propagates it to the others via the `trigger_action()` primitive and IPIs), and indirectly as variation of the contents of the saved user-level register sets, which are dependent on the timing of interrupt delivery.

Interrupts to each replica are observed consistently in the trace analogous to consistent input event ordering in state machine replication. The lowest-level in-kernel interrupt handler simply records that an interrupt occurred in shared state, and IPI's the other replicas. We guarantee `handle_action()` (which does the majority of the in-kernel interrupt handling) is invoked between the same surrounding trace events across all replicas, but the exact point between the two events is imprecise. This implies that saved user-level registers cannot contribute to the trace as they are not consistent on kernel entry. If user-level state suffers an SEU on saving or restoring, we rely on fault propagation to observe the failure, not direct comparison of the state.

**Microkernel Change:** Lowest-level interrupt handling and interrupt-triggered user-level state observation do not contribute to the trace.

*2) Coverage:* Replicas in our approach have fault detection coverage without invasive instrumentation, as if all of the outputs of the replicated systems to devices are consistent, then a transient fault is benign. However, one or more replicas may still be vulnerable to compromise without visibly diverging. Once a single replica microkernel is compromised, all replicas can be subverted as isolation between replicas is software controlled.

In seeking extra assurance of isolation, we chose the following kernel state updates to incorporate into the trace in addition to context switches and device outputs, to balance the trade-off between the overhead of instrumentation of all kernel state changes, and the high-latency of detection of no microkernel-derived contributions to the trace.

**Pagetable content** affects the isolation between replicas, between user-level and the kernel, and between trusted and untrusted application within a replica.

**Capability node content** affects the distribution of authority and thus the isolation boundaries enforced by the microkernel.

Our implementation uses `add_event()` to include these updates in the trace. Recall `add_event()` is a local operation that does not compare the traces. Instead, we rely on `handle_action()`, which is called prior to every kernel exit, to compare the traces for mismatch when the call results in an action being taken. Effectively, we compare traces at least on every context switch and device interrupt.

In addition, we wanted to explore the performance penalty of a higher frequency-of-comparison variant of our system. We created variant which instruments every system call to contribute a summary to the trace with `add_compare_event()` instead of `add_event()`, thus incurring the expense of increased synchronisation overhead on every system call.

In general, one can reduce the latency of trace divergence detection with judicious use of `add_compare_event()` instead of `add_event()` at the expense of increased synchronisation overhead.

**Microkernel Change:** Microkernel instrumented to include safety or security critical kernel state changes in the trace.

**Microkernel Change:** Microkernel augmented with `handle_action()` on kernel exit to consistently invoke asynchronous actions.

### *D. Device Drivers*

As described earlier, device drivers on seL4 run at user-level. Hardware devices expose replica drivers to inconsistent input as either access is exclusive to a single replica (as we do not assume replicated hardware), or they can produce inconsistent behaviour for each replica (e.g., access to timer counters). The following sections describe how we deal with each aspect of device drivers so they provide consistent input to all replicas.

*1) Port-based I/O:* Port-based I/O on x86-64 processors involves the use of port-specific instructions (e.g. `inb` and `outb`) that can be invoked within kernel mode, or at user-level with appropriate hardware permissions enabled. The seL4 microkernel restricts access to port instructions to kernel mode and provides an API for drivers to invoke to access the hardware.

To ensure port-based I/O is be consistent across replicas, we make available the core's number (ID) in-kernel within a replica and use the following design pattern to access hardware registers, where `PRIMARY` is indicative of the core with exclusive access.

```
if (core == PRIMARY) {
    shared_variable = hardware_register;
}
barrier();
result = shared_variable;
barrier();
```

**Microkernel Change:** Core ID available within microkernel.

**Microkernel Change:** Hardware access coordinated for consistent results.

*2) Memory-mapped I/O:* Memory-mapped I/O differs from port-based I/O in that normal read and write instructions are used at user-level to access the hardware directly from within an application. We apply the same design pattern as applied for port-based access, except we do so from user-level. This implies that drivers must be modified to apply the pattern appropriately to observe consistent I/O (or time stamp counter access) across replicas.

To apply the pattern at user-level, a driver needs knowledge of the core ID. As part of the boot process, we make the core ID available to the root task in its initial state. The root task can then selectively propagate the ID to drivers in order to distinguish between replicas and coordinate hardware access. The root task and drivers are trusted to not to cause divergence based on this differing initial state in each replica.

Shared memory is also required to implement the barrier and distribute the I/O read results between replicas, and as such, we also augment the virtual memory mapping system call to dictate if a mapping should be private to the replica, i.e. within the replica's partition (the normal case), or shared between replicas at the corresponding partition offset within the primary partition, independent of the replica performing the mapping operation. Note that shared memory regions are between the same application (i.e. security domain) running on each replica, and that context switches are coordinated across replicas ensuring that when shared memory is accessible, all replicas are running within the same security domain.

**Microkernel Change:** Core ID made available in the initial state of a replica.

**Microkernel Change:** Virtual memory mapping primitives augmented to support limited memory shared across replicas.

*3) Interrupt Driven I/O:* Eventual interrupt delivery to drivers on seL4 is via IPC and thus interrupts are only observed by drivers when invoking an IPC receive primitive. The previously introduced `handle_action()` primitive within the microkernel ensures that interrupt visibility is consistently observed across all replicas. Thus interrupts are visible consistently across all drivers at the same point in execution without further kernel changes.

**Microkernel Property:** Interrupts are delivered consistently via IPC.

*4) DMA:* DMA-based devices create two issues. Firstly, in-memory descriptors used to communicate between the driver and the hardware device need to be observed consistently across all replicas and generally only written once. Secondly, buffers used for bulk transfer (e.g. Ethernet packets) should also be consistent, but ideally the amount of extra copying should be minimised.

We deal with descriptors in a similar manner to memory-mapped device registers using knowledge of the core ID and user-level barriers (implemented in the shared memory) to coordinate access and observations.

Input buffers (i.e. data from devices) are located in memory shared between replicas, thus avoiding copying to private buffers in each replica. Only existing copying is required (e.g. to a socket buffer within the IP stack). The physical addresses associated with the buffers are derived from the root task's initial state, and thus are consistent across all replicas and refer to physical addresses in the primary partition.

If a replica's observations of input buffers diverges due to transient faults, it will eventually result in trace divergence, or is benign. However, output buffers have no indirect checking. We improve the assurance of correctness of output buffers by using private buffers per replica which can be check-summed at user-level and compared for consistency via a system call that invokes `add_compare_event()` with the value of the sum. Thus output is generated per replica, compared, but then DMA-ed only from the `PRIMARY` replica.

Input buffers could theoretically also be checksummed and compared for extra assurance at the expense of some performance compared our prototype's reliance on eventual fault propagation. The advantage of providing a system call for contributing user-level state to the trace is that this trade-off between assurance and performance can be left to the specific system designer.

**Microkernel Change:** A system call added to contribute user-level state to the trace.

Note that software with access to this call could artificially trigger a fault if able to observe non-determinism (e.g., via preemption), and thus this call should be restricted to only drivers, but is not in our current prototype.

*5) User-level:* In order to ensure the determinism required to execute replicas consistently across multiple cores, we require user-level to be either race-free or to hide any non-determinism (e.g., using techniques we described previously for device drivers). The requirement enables us to support imprecise preemption of user-level without triggering divergence of execution.

The race-freedom requirement divides the system into two classes of application: applications that are trusted to avoid causing divergence, and thus have access to the full microkernel API; and untrusted (potentially malicious) applications that must only be given access to a restricted environment not exposed to non-determinism. The seL4 capability model is sufficient to enforce such a restricted environment which consists of no shared memory, only a single thread of execution, together with no rights to create shared memory (inter- or intra-partition) or more threads.

In theory, we could remove this restriction from user-level by consistently context switching at the same point in the user-level instruction stream across all replicas. This might be achieved by using performance counter triggered exceptions together with hardware-triggered breakpoints [44]. However, it is not straight forward due to the differing execution of the primary device managing replica and other replicas. We hope to explore the feasibility of consistent context switching in future work.

## V. Evaluation

We evaluate our framework from several perspectives. Firstly, we quantify the reduction in performance due to replicated computation for the two extremes of a CPU-bound benchmark and a memory-bound benchmark. These two microbenchmarks also serve to highlight the difference between the ARM and x86-64 platforms under test. Secondly, we measure the added interrupt latency of our approach. Thirdly, we measure the performance of Redis (a key-value store) using the Yahoo Cloud Serving Benchmark (YCSB), a real-world, I/O intensive application and benchmark. Lastly, we perform fault injection experiments to confirm our framework's ability to detect transient faults.

The configurations used in the evaluation are:

**Base** An unmodified (unprotected) system.

**DMR** A dual modular redundant variant of the system that does not include system call arguments in the trace, and compares traces on every device interrupt.

**DMRS** A dual modular redundant variant of the system as above with the addition of system call arguments in the trace, and trace comparison on every system call.

**TMR** A triple modular redundant variant of the system that does not include system call arguments in the trace, and compares traces on every device interrupt. Note: our TMR variant only supports error detection, not fault masking. We are exploring fault masking as future work.

**TMRS** A triple modular redundant variant of the system as above with the addition of system call arguments in the trace, and trace comparison on every system call.

We include the system call checking variant as an example of the trade-off of increasing the direct coverage and frequency of trace comparison at the expense of performance.

### A. Platforms

Our evaluation is on two different platforms, an Intel x86-64 platform (a Dell Optiplex 990) and an ARM Cortex A9 platform (a Freescale SABRE Lite featuring an i.MX6 system on a chip). Further details are shown in Table I. For the x86-64 we disable Hyperthreading and Turboboost to ensure consistent performance for each benchmark run.

TABLE I. Hardware Platforms

| | x86-64 | ARM |
|---|---|---|
| CPU | Core i7 2600 | Cortex A9 iMX6 |
| | Quad-core@3.40 GHz | Quad-core@1 GHz |
| Cache | L1 32 KiB D/I per core | L1 32 KiB D/I per core |
| | L2 256 KiB / core | L2 1 MiB shared |
| | L3 8 MiB shared | none |
| Mem | 16 GiB DDR3-1333 | 1 GiB DDR3 |
| | Dual InterLeave | @532 MHz |
| Net | Intel 82579LM | 1000 Mbit/s ENET |

### B. Micro-benchmarks

Our microbenchmarks compare an unmodified *Base*(and thus unprotected) seL4 with DMR and TMR variants on both the ARM and x86-64 platforms.

*1) CPU-bound Micro-benchmark:* To evaluate our framework's effect on CPU-bound applications, we chose a floating point computation from the LMbench benchmark suite [45]. Specifically, we chose `bogomflops`, an array-based floating point benchmark. The size of the floating-point array is 62.5 KiB. The size is 25% of the per-core private L2 cache size on the x86-64 processor and 6% of the shared L2 cache on the ARM processor. Within `bogomflops`, the operations on the data array are repeated 100 times. We read the cycle counter before and after the `bogomflops` call. We repeat this benchmark 100 times in a loop and report the average number of cycles per benchmark run.

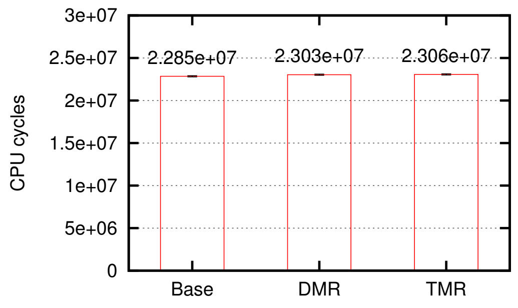

Fig. 2. A comparison of CPU-bound (floating-point) computation on x86-64.

Figures 2 and 3 show the results of the benchmark for x86-64 and ARM respectively. Standard deviations are shown as error bars, and are less than 0.62%. As expected, our

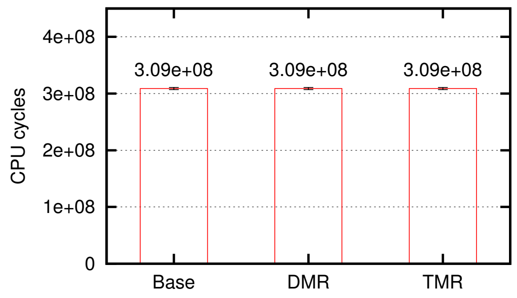

Fig. 3. A comparison of CPU-bound (floating-point) computation on ARM.

framework has nearly no effect on CPU-bound applications as they mostly execute independently within each core's cache.

*2) Memory-bound Micro-benchmark:* To quantify the effect of the redundant execution on a memory-intensive application, we used the following simple copy-based benchmark. The application uses two memory regions (a source and a destination) that are four times the size of the last-level cache on the platform under test (the regions are 32 MiB on x86-64 and 4 MiB on ARM). The regions are pre-mapped to avoid page faults. The benchmark uses `memcpy()` (which is based on `movsq` instruction on x86-64 and `ldm/stm` instructions with preloading on ARM) to copy the source buffer to the destination buffer 100 times. We use a barrier to coordinate the start and finish of the replicas, together with the platform's time-stamp counter to record start and finish times for each run. We run the benchmark 10 times in a loop, and report average copy bandwidth achieved in megabits/s together with standard deviations in Figures 4 and 5.

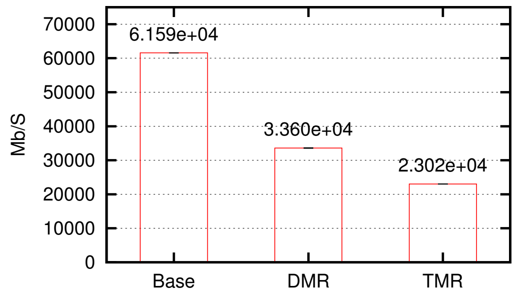

Fig. 4. A comparison of a memory-bound (`memcopy`) application on x86-64.

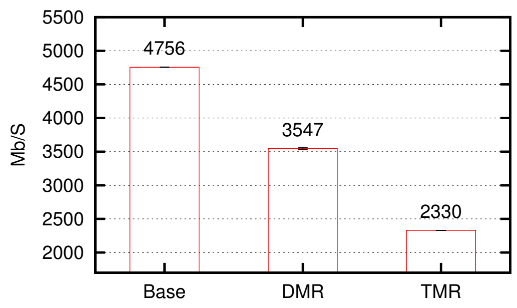

Fig. 5. A comparison of a memory-bound (`memcopy`) application on ARM

On both platforms, we observe that the replicated configurations split the total available memory copy bandwidth between the replicas. Compared to x86-64 the ARM platform has a lower performance penalty when moving from a non-replicated to a replicated scenario. This is due to a single core on ARM being insufficient to saturate the available memory copy bandwidth of approximately 7 Gb/s. For x86-64 a single core is much closer to saturating the available memory copy bandwidth of approximately 68 Gb/s.

*3) Interrupt Latency:* Interrupt delivery in our approach now requires propagation to all replicas and agreement on when the interrupt becomes visible to the replicas in order to preserve consistency. Interrupt latency is highly dependent of system activity at the time (e.g. interrupt disabling), so our benchmark involves an idle system consisting of only the in-kernel idle thread, and a user-level timer driver which is effectively an interrupt handling thread.

The timer driver programs the platform-dependent timer to trigger an interrupt and then blocks waiting for its arrival via IPC. To measure the effect on latency, we instrument the in-kernel interrupt handler on the interrupt-handling core to take a timestamp early in the interrupt handling code. We also take a timestamp after the user-level driver receives the interrupt notification, i.e. after the interrupt is propagated across all cores and the notification is synchronised. The difference between the two timestamps is our metric for interrupt latency. The latency is measured 100 times in a loop for a baseline unprotected seL4, and DMR and TMR variants. The average of the 100 runs together with standard deviations are shown for each variant for each platform in Figures 6 and 7.

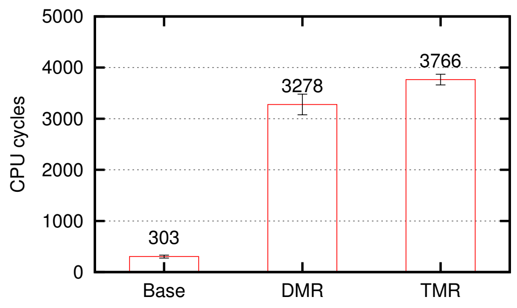

Fig. 6. Interrupt Latencies x86-64

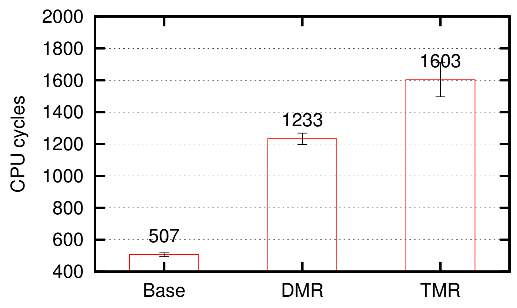

Fig. 7. Interrupt Latencies ARM

As we can observe from the results, the interrupt latency increases significantly for DMR and TMR modes. The increase in latency is due to the latency of sending and receiving an IPI, and the three barriers used to coordinate consistent interrupt observation across replicas.

## C. I/O-intensive Benchmark

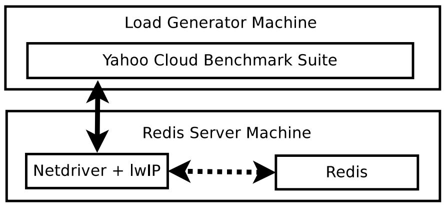

Fig. 8. Redis-based benchmark architecture.

To test our system under a more realistic load we chose the Redis key-value store set up as shown in Figure 8. The target system runs seL4 with one process dedicated to lwIP combined with the Ethernet device driver, including handling I/O interrupts. Another process runs an instance of Redis. Note that we run Redis as volatile store (we disabled file system access), as our prototype lacks a port of a file system. The device driver was modified as described in the description of our framework so as to support replication in DMR, DMRS, TMR, and TMRS configurations.

We evaluate performance using Yahoo! Cloud Serving Benchmarks (YCSB) [46], running on a dedicated load generator machine, with a dedicated Gigabit Ethernet link between the load generator and the machine under test. During the benchmarks we monitor the CPU-load and network bandwidth to ensure the benchmark performance is not limited by the load generator.

YSCB consists of several workloads. We use the same A–E benchmarks as presented by the benchmark developers [46], which are as follows.

**A:** update-heavy workload (50/50 read and writes) using zipfian distribution for record selection in the store.
**B:** read-mostly workload (95/5) with zipfian distribution.
**C:** read-only workload with zipfian distribution
**D:** new records inserted, then most recently inserted record are read.
**E:** short ranges of records are queried, where record selection is zipfian, but the number of records in the range is uniformly distributed.

We set `recordcount` to 70000 on ARM and x86-64 for all workloads. `operationcount` was set to 10 and 20 times `recordcount` respectively for ARM and x86-64 except for 'E' which was 1. The goal of tuning the parameters was to give a run time of 30–90 seconds, except for 'E' on ARM which was 370-500 seconds, and a database size (around 160 MiB on ARM and 190 MiB on x86-64 as reported by the 'info memory' Redis client command) significantly larger than the last-level cache size.

For each platform, we ran the YCSB benchmark set 3 times for an unprotected uniprocessor baseline, DMR, DMRS, TMR, and TMRS variants of the system. The averaged throughput results are reported in Figures 9 and 10, with standard deviations being less than 2.9% (x86-64) and 1.5% (ARM). Note: we multiplied the 'E' throughput by 50 to make it comparable on the scale.

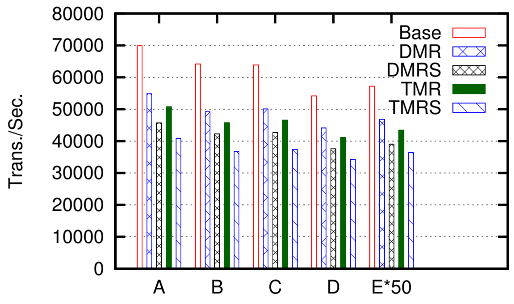

Fig. 9. Average Redis transactions per second on x86-64 for each configuration and workload. 'E' multiplied by 50.

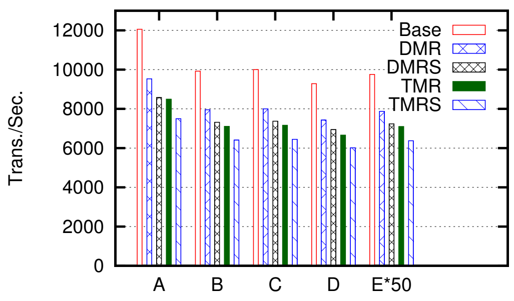

Fig. 10. Average Redis transactions per second on ARM for each configuration and workload. 'E' multiplied by 50.

We see that for the x86-64 platform the performance of the DMR is between 77–82% of the baseline, and TMR is 71–76% of the baseline performance across all workloads. The system call checking variants, DMRS and TMRS are 65–69% and 57–64% of baseline respectively. For the ARM platform, DMR achieves 79–81% of the baseline, and TMR achieves 70–73% of the baseline performance. DMRS and TMRS variants achieve 71–75% and 62–65% of the baseline. Replicated execution does impose a modest performance penalty, with system call checking further increasing the penalty in return for increased direct coverage and more frequent trace comparison.

## D. Fault Injection Experiments

Our fault injection experiment aims to empirically validate our approach's ability to detect faults, and convert those faults into graceful failures. In this experiment, we use a spare CPU core to perform fault injection into memory. While our goal is to handle relatively rare transient faults or SEUs, our fault model in this experiment injects multiple SEUs to compress the time required to run the campaign. The campaign repeatedly injects until the system under test fails, restarts the system, and then continues fault injection. The physical address and bit to flip are chosen randomly. The time between fault injection is also chosen randomly between 0x10000000 to 0xffff0000 CPU cycles, as reported by the `rdtsc` and `c15` cycle counter registers on x86-64 and ARM respectively. Given the high degree on non-determinism of the system under test when combined with the benchmark harness, we don't attempt to seed the random number generator consistently across the two configurations. We rely on having enough samples to observe overall trends in the data.

The system under test is the Redis and YCSB benchmarking software used previously. The YCSB benchmarking client is modified to embed CRC32 checksums of the key-value pairs into the values written to the Redis server. The YCSB client can then validate the correctness of data returned by Redis by comparing the embedded checksum with a recalculated checksum.

We test an unprotected and a DMR version of the system. A script monitors the outputs from the machines and YCSB client. Once the script detects the Redis server failure or errors reported by YCSB client, it logs the reason for failure, and restarts the machines and YCSB client.

The following totals are accumulated over all the repeated runs.

**Total:** Total faults injected.
**Observed Failures:** The number of system failures (graceful or otherwise) reported by either Redis or the YCSB client.

In our DMR configuration, we count the following graceful fail-stops that occur when our framework observes inconsistency between the replicas. A graceful failure is a failure resulting in invocation of our fault handling routine.

**Kernel Barrier Timeout:** A graceful fail-stop because of a kernel-mode barrier timeout in DMR mode. A consequence of one of the cores becoming non-responsive.
**Kernel Trace Checksum:** A graceful fail-stop because of a trace comparison failure in DMR mode.

Other failures are uncontrolled, and we divide the failures into the following types.

**User VMF:** Virtual memory faults triggered from user level.
**User Other:** Other exceptions triggered from user mode.
**Kernel GP:** A general protection fault in kernel mode.
**YCSB Corrupt Result:** The number of results returned by Redis that are not correct.
**YCSB Errors:** Run-time exceptions reported by YCSB.

Tables II and III show the total number of fault injections performed over all runs for an unprotected Redis configuration and a DMR configuration, and the breakdown of the observed failures as absolute numbers and as a percentage of the number of failures. The difference in number of faults injected is not indicative of anything other than length of time each experiment has executed. However, we do observe that in the DMR configuration, a higher proportion of failures are observed. We believe that is due to the increased scrutiny our DMR configuration imposes on the system.

In the unprotected case, we see the majority of the failures are application crashes due to user-level triggered VM faults (32% and 44%) or other user-level triggered exceptions (16%). We also see a significant fraction of failures as faults propagated as incorrect results returned to YCSB (46% and 51%) or results that cause YCSB to throw a java run-time exception (6% and 4%). There is also a small number of in-kernel exceptions observed (0.7% and 0.9%) The distribution of uncontrolled failures reflects the high proportion of memory used by the Redis server.

In the DMR configuration we observed no uncontrolled failures in our results. All observed failures are explicit fail-stops triggered by our framework except for erroneous results: 7% of failures for x86-64 and 1% for ARM. The higher proportion of observed failures for x86-64 is consistent with our x86-64 configuration featuring DMA buffers 12 times the size of the ARM configuration. Our system is not completely immune from uncontrolled failures as there are small parts of the system outside of the SoR (e.g. the DMA I/O buffers and device registers associated with non-redundant devices are fundamentally a single point of failure). However, in our experiments, any faults outside the SoR propagated to the SoR, resulting in graceful failure. We claim that our framework significantly improves the assurance of correct operation as it has captured all but the observed erroneous results in our experiments. The erroneous results observed could be detected within YSCB via integrity checksums computed within the SoR, and then re-requested, as the results are computed correctly, but are corrupted during I/O outside the SoR. The majority of graceful fail-stops observed were kernel barrier timeouts. Barrier timeouts occurs when one replica takes an exception (or performs a system call) and the other replica does not. The 60% and 57% observed are close to the total proportions of uncontrolled exceptions, kernel GP, and YCSB errors in the unprotected case which were 54% and 49%. The remainder of the fail-stops were due to I/O output comparisons failing because of inconsistent output observed from the replicas. Again the total proportions of output inconsistencies and YCSB corrupted results (40% and 43%) are close to the erroneous YCSB results in the unprotected case (46% and 51%).

TABLE II. NUMBER OF SYSTEM FAILURE TYPE OCCURRENCES AND PERCENTAGE OF TOTAL FAILURES (X64).

| | Unprot. | | DMR | |
|---|---|---|---|---|
| Total Injected | 34412 | | 27236 | |
| Failures | 812 | | 687 | |
| User VMF | 256 | 31.53% | 0 | 0% |
| User Other | 126 | 15.52% | 0 | 0% |
| Kernel GP | 6 | 0.74% | 0 | 0% |
| Kernel Barrier Timeout | - | - | 413 | 60.12 % |
| Kernel Trace Checksum | - | - | 224 | 32.60% |
| YCSB Corrupted Result | 375 | 46.18% | 50 | 7.28% |
| YCSB Errors | 49 | 6.03% | 0 | 0% |

TABLE III. NUMBER OF SYSTEM FAILURE TYPE OCCURRENCES AND PERCENTAGE OF TOTAL FAILURES (ARM).

| | Unprot. | | DMR | |
|---|---|---|---|---|
| Total Injected | 195662 | | 211759 | |
| Failures | 654 | | 921 | |
| User VMF | 286 | 43.73% | 0 | 0% |
| User Other | 0 | 0% | 0 | 0% |
| Kernel GP | 6 | 0.92% | 0 | 0% |
| Kernel Barrier Timeout | - | - | 529 | 57.44% |
| Kernel Trace Checksum | - | - | 383 | 41.58% |
| YCSB Corrupted Result | 335 | 51.22% | 9 | 0.98% |
| YCSB Errors | 27 | 4.13% | 0 | 0 |

## E. Size of Changes

To give an approximation of the effort required and invasiveness of our approach, Table IV shows the number of lines of code modified or added to the microkernel for the

TABLE IV. LINES OF MODIFIED CODE IN THE MICROKERNEL

| Microkernel code | LoC |
|---|---|
| machine-independent code | 576 |
| x86-64 specific code | 522 |
| ARM specific code | 440 |

machine independent and architecture-dependent parts of the microkernel. We see that on x86-64 1098 lines of code were modified or added to implement our microkernel mechanisms. For ARM, it required 1016 lines of code. From a software engineering perspective we see this as a modest cost.

In addition to microkernel changes, our approach relies on modifying device drivers to hide non-determinism, checksum output, and coordinate non-redundant device access. These changes come at a software engineering cost, but not necessarily a verification cost as the assurance needed for specific drivers is determined by the system architect.

TABLE V. LINES OF MODIFIED CODE IN THE ETHERNET DRIVERS

| Ethernet code changes | LoC |
|---|---|
| Common | 30 |
| x86-64 Ethernet code | 72 |
| ARM Ethernet code | 125 |

Table V shows the number of lines of code changed in the two Ethernet drivers used in our experiments. The number of changes are small and mostly related to managing the descriptor rings and accessing device registers.

## F. *Re-establishment of Proof Guarantees*

The motivation for our work is to provide further assurance of correct operation of hardware, given the provably correct operation of software. While our prototype does provide improved assurance of correct hardware operation, it also breaks the proofs of software correctness, and thus a discussion of their re-establishment is warranted.

- seL4 is a uniprocessor event-based microkernel to enable sequential reasoning about code correctness. Our prototype introduces concurrency only with respect to the shared *traces* of the replicas that otherwise execute independently and sequentially. We expect re-establishing the proof of correctness to be of similar complexity to converting seL4 to be multiprocessor kernel with a single coarse-grained lock, which is the subject of ongoing work.
- We believe re-establishment of the correctness proofs would enable re-establishment of the integrity guarantees and authority confinement guarantees proven previously without significant challenges [9].
- The non-interference proof that provides strong assurance of confidentially guarantees is expected to be the biggest challenge. Intuitively, a replica's trace contains a subset of the history of that replica across the isolation domains within a replica. If replicas diverge, and the system gracefully stops, then at least one-bit of information derived from a complex view of history has leaked to all replicas, thus violating non-interference.
  The formulation of non-interference that this behaviour can satisfy, while still providing confidentiality guarantees is not obvious and the subject of future research. In the worst case, we expect our approach to retain the integrity guarantees for safety and security critical systems where strong confidentiality is not a priority.

## VI. CONCLUSIONS AND FUTURE WORK

In order to improve the trustworthiness of commodity hardware, we have presented an approach that uses redundant and deterministic execution to cooperatively self-check replicated microkernel-based systems. The approach consists of a small number of changes to the microkernel, and a simple design pattern to hide non-determinism and enable consistent redundant execution.

We evaluated the performance of our approach in DMR and TMR configurations for microbenchmarks and a realistic Redis-based key/value store. The microbenchmarks showed little change for CPU-bound systems, but for memory bound applications, the limited total memory bandwidth available to the replicas does reduce performance. The latency of interrupt notification to user-level device drivers is also increased.

For a more realistic I/O intensive benchmark featuring a mix of computation, memory access, and interrupt delivery, the performance of a DMR configuration was approximately 77–82% of an unprotected system, and for TMR, 71–76% on an x86-64 platform (for ARM, the performance was approximately 79–81% and 70–73% respectively). This is a small performance reduction in return for the extra assurance provided by redundant execution that includes the lowest-level software in the system. Increasing the frequency of replica checking to every system call further reduced performance.

Our fault injection experiments showed that our approach converted an unprotected Redis-based system that featured uncontrolled failures and produce incorrect results into a system that detected transient faults and gracefully performed fail-stops. The system did propagate a small percentage of errors to clients in our experiments, which could be detected via integrity checksums computed within the SoR.

The changes required to implement our approach are modest in size and are a one-off cost. However, re-verification of the microkernel remains an open issue to be explored.

In the future, we plan to explore using the TMR configuration for fault masking. Fault masking requires fault isolation between replicas, which on commodity hardware is based on software-configurable virtual memory hardware. If the risk of fault propagation between partitions is acceptable, our TMR approach could also increase the reliability of microkernel-based systems.

When combined with a verified microkernel such as seL4, our mechanisms and framework together provide a very high degree of assurance of correct, i.e. trustworthy, operation.

## ACKNOWLEDGEMENTS

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

## References

[1] J. M. Rushby, "Design and verification of secure systems," in *8th SOSP*, Pacific Grove, CA, US, Dec 1981, pp. 12–21.

[2] J. Alves-Foss, P. W. Oman, C. Taylor, and S. Harrison, "The MILS architecture for high-assurance embedded systems," *Int. J. Emb. Syst.*, vol. 2, pp. 239–247, 2006.

[3] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *CACM*, vol. 17, no. 7, pp. 413–421, 1974.

[4] R. Meushaw and D. Simard, "NetTop commercial technology in high assurance applications," 2000. [Online]. Available: http://cps-vo.org/node/6511

[5] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *19th SOSP*, Bolton Landing, NY, US, Oct 2003, pp. 193–206.

[6] P. Brinch Hansen, "The nucleus of a multiprogramming operating system," *CACM*, vol. 13, pp. 238–250, 1970.

[7] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *SOSP*, Big Sky, MT, USA, Oct 2009, pp. 207–220.

[8] D. Elkaduwe, P. Derrin, and K. Elphinstone, "Kernel data – first class citizens of the system," in *2nd Int. WS Obj. Syst. & Softw. Arch.*, Victor Harbor, South Australia, Australia, Jan 2006, pp. 39–43.

[9] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein, "seL4 enforces integrity," in *Interactive Theorem Proving (ITP)*, Nijmegen, The Netherlands, Aug 2011, pp. 325–340.

[10] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "seL4: from general purpose to a proof of information flow enforcement," in *S&P*, San Francisco, CA, May 2013, pp. 415–429.

[11] T. Sewell, M. Myreen, and G. Klein, "Translation validation for a verified OS kernel," in *PLDI*, Seattle, Washington, USA, Jun 2013, pp. 471–481.

[12] R. Baumann, "Soft errors in advanced computer systems," *Design & Test Computers*, vol. 22, no. 3, pp. 258–266, May 2005.

[13] S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender, "Predicting the number of fatal soft errors in Los Alamos national laboratory's ASC Q supercomputer," *Trans. Dev. & Mater. Reliability*, vol. 5, no. 3, pp. 329–335, Sep 2005.

[14] G. P. Saggese, N. J. Wang, Z. T. Kalbarczyk, S. J. Patel, and R. K. Iyer, "An experimental study of soft errors in microprocessors," *IEEE Micro*, vol. 25, no. 6, pp. 30–39, Nov 2005.

[15] 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 *6th EuroSys*, Salzburg, AT, Apr 2011.

[16] J. Arlat, J.-C. Fabre, M. Rodríguez, and F. Salles, "Dependability of COTS microkernel-based systems," *Trans. Computers*, vol. 51, no. 2, pp. 138–163, Feb 2002.

[17] J. Xu, S. Chen, Z. Kalbarczyk, and R. K. Iyer, "An experimental study of security vulnerabilities caused by errors," in *DSN*, 2001, pp. 421–430.

[18] 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.

[19] S. Govindavajhala and A. W. Appel, "Using memory errors to attack a virtual machine," in *S&P*, 2003, pp. 154–165.

[20] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *41st ISCA*, Jun 2014, pp. 361–372.

[21] "Row hammer," http://en.wikipedia.org/wiki/Row_hammer.

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

[23] A. L. Hopkins Jr., T. B. Smith III, and J. H. Lala, "FTMP—a highly reliable fault-tolerant multiprocess for aircraft," *Proc. IEEE*, vol. 66, no. 10, pp. 1221–1239, 1978.

[24] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock, "SIFT: Design and analysis of a fault-tolerant computer for aircraft control," *Proc. IEEE*, vol. 66, no. 10, pp. 1240–1255, 1978.

[25] D. Briere and P. Traverse, "AIRBUS A320/A330/A340 electrical flight controls - A family of fault-tolerant systems," in *23rd Int. Symp. Fault-Tolerant Comput.*, Jun 1993, pp. 616–623.

[26] T. Slegel, I. Averill, R.M., M. Check, B. Giamei, B. Krumm, C. Krygowski, W. Li, J. Liptay, J. MacDougall, T. McPherson, J. Navarro, E. Schwarz, K. Shum, and C. Webb, "IBM's S/390 G5 microprocessor design," *IEEE Micro*, vol. 19, no. 2, pp. 12–23, Mar 1999.

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

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

[29] D. Bernick, B. Bruckert, P. Del Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen, "NonStop advanced architecture," in *35th DSN*, Washington, DC, US, 2005, pp. 12–21.

[30] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *27th ISCA*, Jun 2000, pp. 25–36.

[31] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *3rd Symp. Code Generation & Optimization*, 2005, pp. 243–254.

[32] C. Wang, H. S. Kim, Y. Wu, and V. Ying, "Compiler-managed software-based redundant multi-threading for transient fault detection," in *5th Int. Symp. Code Generation & Optimization*, 2007, pp. 244–258.

[33] A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, and D. A. Connors, "Using process-level redundancy to exploit multiple cores for transient fault tolerance," in *37th DSN*, Jun 2007, pp. 297–306.

[34] B. Döbel, H. Härtig, and M. Engel, "Operating system support for redundant multithreading," in *12th EMSOFT*, Tampere, SF, Oct 2012, pp. 83–92.

[35] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," *Trans. Comp. Syst.*, vol. 14, pp. 80–107, 1996.

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

[37] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *16th SOSP*, Stevenson, WA, US, Oct 2007, pp. 335–350.

[38] C. Tan, Y. Xia, H. Chen, and B. Zang, "TinyChecker: Transparent protection of VMs against hypervisor failures with nested virtualization," in *42nd Int. Conf. Dependable Syst. & Netw. WS (DSN-W)*, Jun 2012, pp. 1–6.

[39] K. Elphinstone and G. Heiser, "From L3 to seL4 – what have we learnt in 20 years of L4 microkernels?" in *SOSP*, Farmington, PA, USA, Nov 2013, pp. 133–150.

[40] "sel4," http://sel4.systems/.

[41] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an OS microkernel," *Trans. Comp. Syst.*, vol. 32, no. 1, pp. 2:1–2:70, Feb 2014.

[42] B. Blackham, Y. Shi, and G. Heiser, "Improving interrupt response time in a verifiable protected microkernel," in *EuroSys*, Bern, Switzerland, Apr 2012, pp. 323–336.

[43] J. G. Fletcher, "An arithmetic checksum for serial transmissions," *Trans. Comm.*, vol. 30, no. 1, pp. 247–252, Jan 1982.

[44] 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 *5th OSDI*, Boston, MA, US, 2002.

[45] "Lmbench - tools for performance analysis," http://lmbench.sourceforge.net/.

[46] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," Indianapolis, IN, US, Jun 2010.