Operating System Support for Redundant Multithreading

Björn Döbel
TU Dresden
Dresden, Germany
doebel@tudos.org

Hermann Härtig
TU Dresden
Dresden, Germany
haertig@tudos.org

Michael Engel
TU Dortmund
Dortmund, Germany
michael.engel@tu-dortmund.de

ABSTRACT

In modern commodity operating systems, core functionality is usually designed assuming that the underlying processor hardware always functions correctly. Shrinking hardware feature sizes break this assumption. Existing approaches to cope with these issues either use hardware functionality that is not available in commercial-off-the-shelf (COTS) systems or poses additional requirements on the software development side, making reuse of existing software hard, if not impossible.

In this paper we present Romain, a framework that provides transparent redundant multithreading[1] as an operating system service for hardware error detection and recovery. When applied to a standard benchmark suite, Romain requires a maximum runtime overhead of 30 % for triple-modular redundancy (while in many cases remaining below 5 %). Furthermore, our approach minimizes the complexity added to the operating system for the sake of replication.

Categories and Subject Descriptors

D.4.5 [**Reliability**]: Fault-tolerance

General Terms

Design,Reliability

Keywords

Microkernel, Redundant Multithreading

1. INTRODUCTION

Modern processor development exploits smaller hardware feature sizes to provide more functional units on the same die size in order to increase processors' performance and feature sets. This trend has a major drawback: chips become more error prone and constantly or temporarily exhibit incorrect behavior.

Sources for hardware-related errors are manyfold. The lithographic process used for implementing hardware features at sizes as small as 22 nm still uses wavelengths of 193 nm. This may result in processors that don't work right from the start [7] or expose a large variety in terms of gate switch delays [25]. Furthermore, functional units may suffer from heat-induced errors [37], aging effects, or errors caused by undervolting or frequency scaling [49]. Finally, radiation originating from space or the packaging of the chip may lead to *single-event upsets (SEU)* [24], transient errors that manifest as bit-flips at the software level.

Coping with these problems at the hardware level has the advantage that software does not need to be aware of them. However, such solutions often involve specialized hardware circuitry, making these solutions infeasible in commercial-off-the-shelf (COTS) systems. In contrast, software-implemented fault tolerance can easily be adapted by patching the involved applications and integrating with operating system (OS) resource scheduling. This flexibility comes at a price: Software developers need to be aware of potential hardware errors and use the right techniques to handle them. This requirement decreases developers' productivity. Therefore, we argue that it should be the operating system's responsibility to deal with hardware errors and provide resilience guarantees to user-level applications.

OS support for handling errors has been extensively researched with respect to software errors [15, 19, 29]. Hardware errors have been investigated with respect to I/O devices and device drivers [18, 30]. However, tolerance against CPU failures is often limited to systems that can make use of specialized hardware components or employ dedicated software techniques [5, 12].

Our goal is to have the operating system provide fault tolerance against transient hardware errors in the CPU while still running on cheap COTS hardware and without restraining the software developers to using dedicated compilers or programming techniques.

In this paper we make the following contributions:

- We present our implementation of Romain, an operating system service using *software-implemented transparent redundant multithreading* [33] (RMT) to detect hardware errors during the execution of user-level applications. Romain recovers from detected errors using n-way modular redundancy.

  To minimize runtime overhead, replicas are distributed across all available CPU cores. Application state is only compared before externalizing state, e.g. when

[1]Note that the term *redundant multithreading* refers to using multiple threads for replication, not necessarily to support multithreaded applications.

performing system calls. In the current implementation, Romain is restricted to replicate single-threaded applications. We discuss how to extend our approach to handle multithreaded programs.

- Romain is implemented within a state-of-the-art capability-based operating system and we show how capabilities can efficiently be managed for replicated applications.
- Shared memory needs special treatment with respect to replicated execution. We investigate trap & emulate techniques for shared memory accesses and present a solution that works without a complex instruction emulator.
- We discuss how Romain fits into ASTEROID, an operating system architecture designed with the possibility of hardware errors in mind. ASTEROID provides a critical core of software components, that must be trusted to work correctly and that need to be hardened using other techniques than the rest of the system. We refer to this critical core as the *reliable computing base* (RCB). Romain adds less than 3,000 lines of code to this RCB

After giving an overview of the broad spectrum of available research in our area in Section 2, we discuss our assumptions about the hardware fault model in Section 3. Section 4 introduces the ASTEROID OS architecture and gives a short introduction to the Fiasco.OC kernel, which provides the necessary operating system services in our RCB. In Section 5 we then discuss the implementation of Romain, before we evaluate Romain with experiments based on the MiBench benchmark suite [13] in Section 6.

2. RELATED WORK

Research into operating systems fault tolerance often focuses on preventing or recovering from software errors, because these occur at a much higher frequency than hardware errors in today's systems. The Minix3 operating system [15] provides fault-tolerant execution by monitoring state-less applications for crashes and restarting them if necessary. David and colleagues extended this concept with kernel-protected state storage that can survive restarts [10]. Vogt et al. added Minix3-like restartability to the L4/Fiasco microkernel, focusing on the reintegration of restarted services into a capability-based system [46]. The SeL4 microkernel aims at fault avoidance by formally verifying that certain software errors can never happen [19]. All these approaches focus on software errors, whereas our approach aims at detecting and recovering from errors in computational hardware.

The existence of faulty hardware has been researched by the OS community in the context of I/O devices. Hard disks and network communication were hardened using redundancy at the data and device level [30, 31] and these approaches were widely adopted in practical systems. Apart from that, most research on device-related issues focuses on software errors, especially in device drivers [29, 35]. In contrast to these works, this paper is mainly concerned with hardware errors occurring in the CPU instead of I/O devices. As a notable exception, Kadav and colleagues proposed a tool to validate device driver code to not trust data read from hardware without proper validation [18].

Operating systems for highly available mainframe computers are designed to cope with all potential malfunctions at the hardware and software level [5, 6, 17]. These systems rely on tight integration with specialized hardware [16, 32]. In our work, we provide tolerance against hardware faults using a commodity operating system running on COTS hardware.

Chip manufacturers produce hardware that masks errors, so they never become visible to software. A prominent example for such technologies is the application of error-correcting codes (ECC) [24] to protect memory and CPU registers. Researchers also proposed hardware extensions to handle errors in functional units: The DIVA architecture suggested to augment a non-reliable CPU core with a checker core built in a less complex and less error-prone way [4]. IBM's Power6 CPUs enhanced the processor pipeline with signature checks [32]. IBM's PowerPC 750GX allows executing software redundantly on lockstep processors [16].

Hardware error detection, masking and recovery is attractive, because it simplifies the lives of software developers. However, integrating these measures into a platform is costly and it takes a long time (if it happens at all) until these solutions reach the COTS market. We therefore develop an operating system architecture that solely relies on hardware features available in COTS hardware.

A range of software-only solutions dealing with hardware errors have been proposed. Oh et al. implemented checking of application control-flow using compiler-generated signatures [26] and duplicated instructions to detect computational errors [27]. Reis et al. integrated control-flow checking and error detection into one compiler, SWIFT [34]. SWIFT still had a couple of vulnerabilities that were addressed by Fetzer et al. [11] using arithmetic coding techniques.

Software-only solutions work without any support from the underlying hardware. However, they usually come as compiler extensions and require the whole software stack to be recompiled, which is impossible for proprietary third-party software. Our solution replicates execution at the binary code level and therefore works for all kinds of applications. Furthermore, it is transparent and does not require cooperation by the replicated application.

Hardware-level redundant multithreading executes the same code on multiple hardware threads and validates the order and content of memory accesses [33]. Wang et al. presented a compiler solution that achieves the same without the need for specialized hardware [47], but requires to recompile the whole software stack with their compiler. Shye and colleagues presented a runtime approach to RMT [38]. Their solution runs applications redundantly and uses binary recompilation to redirect system calls and accesses to shared memory to a master process that compares replica states. We chose the same approach with Romain and take Shye's approach to the operating system level, but provide three advantages: First, we also inspect OS-level events such as page faults, increasing the amount of monitored events for the sake of earlier error detection. Second, we do not rely on a binary recompiler to replicate applications, and thereby can come up with a solution less complex to comprehend. Third, by using a microkernel as the basis of our work, we enable to use replication to harden OS services, such as device drivers and protocol stacks.

3. FAULT MODEL

The design and experiments described in this paper assume a *single event upset* (SEU) fault model [50]. SEUs are *transient* errors caused by cosmic or environmental radiation, which can cause a transistor state change, resulting in a bit turning from 0 to 1 or vice versa.

Computer memory was the first device found to suffer from SEUs [50]. ECC techniques have been developed to detect and recover from memory SEUs and we observe these techniques to be widely available in today's COTS systems [24]. Hence, we focus our efforts on SEUs that affect the functional units of the CPU.

The probability of radiation effects depends on the geographical location (natural radiation levels vary) and the height above sea level [24,41]. Time does not influence SEU probabilities. Therefore, we assume SEUs to be distributed uniformly over time. In line with related research we assume that SEUs occur rarely enough so that only a single error in a single functional unit is active at a given point in time [34].

Several studies show that a significant amount of SEUs are masked either by the hardware or by application code before they manifest as observable deviation in program behavior. Saggese injected microarchitectural faults in a simulator and found hardware to mask between 0.5 and 30 percent of all SEUs before they propagate to the software level [36]. Arlat et al. injected faults into a microkernel operating system and found between 25 and 30 percent of all faults to be masked by the hardware or the OS [2]. Wang and colleagues identified programming constructs that make the application execute correctly even if SEUs cause the CPU to take a wrong branch. These outcome-tolerant branches were found to make up between 20 and 30 percent of all branches in the SPEC CPU 2000 benchmarks [48]. Romain benefits from these observations and delays state comparisons between replicas until the point where this state becomes visible to external applications.

Despite focusing on SEUs, we believe that the error detection mechanisms used in Romain will also detect other fault classes, such as permanent errors, as well as semi-permanent errors that occur for instance when heat crosses a certain threshold and vanish once the chip cools down. We think the difference between these error models is not in detection but in the way recovery works. While SEUs can be fixed by overwriting faulty state with correct state, non-transient errors need more complex recovery such as migrating the faulty job to another hardware node [8] or switching to an alternative implementation that does not make use of faulty hardware circuitry [23].

4. SYSTEM ARCHITECTURE

The ASTEROID system architecture is depicted in Figure 1. The hardware consists of COTS components, which are manufactured to come at a cheap price and for this reason do not incorporate specialized hardware fault tolerance mechanisms. As an exception, we assume memory hardware to be protected by ECC, which is a widespread feature of COTS memory devices these days.

The software stack is split into two layers. The critical core includes all services user-level applications need to rely on. These services include fundamental operating system services such as resource scheduling, multi-tasking and address space isolation. Additionally, the core also provides functionality that allows higher-level components to correctly execute in the presence of hardware faults. In our case, this includes the Romain framework.

The higher-level *application layer* comprises all applications running on top of the ASTEROID system and making use of its services. These applications are implemented by

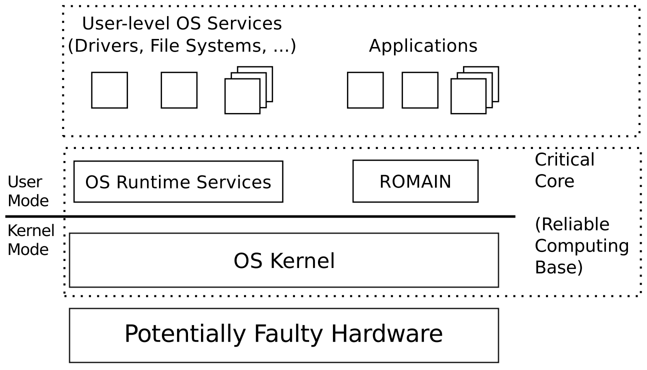


Figure 1: ASTEROID System Overview

developers without specific knowledge about system and hardware internals. Furthermore, we observe a growing body of software that is delivered only in binary form, ranging from mobile applications downloaded from AppStores to device drivers that are distributed without their source code. Therefore, our goal is to run unmodified applications and have the critical core provide the necessary mechanisms to transparently detect and recover from hardware errors.

While the critical core provides all means to safely execute applications in the application layer, the core's fault tolerance mechanisms cannot be used to protect core software itself. Therefore, other mechanisms need to be used at this level to ensure correct execution. In contrast to the application layer, we have full control over the software stack running in the critical core. This allows us to make use of potentially expensive techniques available at the software level such as applying basic block signatures [26], operand encoding [11], and compiler-inserted assertions [45].

4.1 Reliable Computing Base

In the context of secure systems, the term *trusted computing base* (TCB) refers to the set of hardware and software components an application needs to trust in order to maintain certain security goals [14]. To reduce testing and validation efforts, it has been proposed to minimize the TCB as much as possible [9,39]. Similar to the TCB, we think of the critical core as the software components that are necessary to provide transparent replication to applications and that itself need to be hardened using different means. Hence, we coined the term *reliable computing base* (RCB) for the critical core.

The mechanisms used to harden the RCB will be potentially expensive in terms of implementation effort or runtime overhead [11]. Therefore, we strive to minimize the amount of time spent executing in the RCB as well as the amount of code requiring special treatment. To this end, we make three design decisions: First, we minimize interaction between a replicated application and the RCB during normal runtime by only inspecting replica states at points where this state is externalized. Second, we minimize the amount of code that is added to the system by Romain. This rules out using large code bases, such as a binary recompilation tool used in Shye's work [38]. The runtime recompiler alone would add about 100,000 lines of code to the RCB. Instead, we use facilities that already exist in the operating system and augment them for our purposes where needed.

Third, the operating system kernel is part of the RCB. However, not all OS functionality lies on the critical path for providing fault tolerant execution. By using a componentized microkernel, we minimize the RCB and additionally enable Romain to be applied to system-level components such as device drivers and protocol stacks, so that these components are executed outside the RCB.

4.2 Fiasco.OC: Minimizing the RCB

The ASTEROID operating system is based on the Fiasco.OC microkernel [43]. Software running on top of Fiasco.OC consists of objects implemented in dedicated processes.

Similar to other third-generation microkernels such as Nova [40] and SeL4 [19], Fiasco.OC enables fine-grain access control over kernel objects by implementing object references as kernel-protected capabilities. A thread of execution can only use functionality of an object if it possesses a capability for this object. Capabilities are managed by the kernel on a per-process basis and are stored in a capability table. Similar to file descriptors in POSIX operating systems, a process only references its capabilities using indices into this capability table, but it never knows where the object behind this capability is implemented and cannot forge capabilities it did not explicitly obtain.

If owning an object capability, an application can use its functionality through a system call, which translates a function call into a message sent through the kernel's interprocess communication (IPC) mechanism. The parameters for a system call are passed through architectural registers and through the user-level thread control block (UTCB), a thread-specific memory region shared between the kernel and the user process. Fiasco.OC's IPC mechanism does not only support sending messages, but also attaching capabilities to a message. This is called a mapping and can be used to transfer access rights to memory pages or object capabilities between processes.

Memory management for Fiasco.OC applications is performed at the user-level using an implementation of hierarchical paging which has been derived from SawMill [3]. Every process runs a dedicated memory manager that is responsible for managing this process's address space. The memory manager knows about all sources of memory mappings (dataspaces in SawMill's terms). Whenever another thread of this process raises a page fault, the kernel reflects this fault to the memory manager, which in turn resolves the page fault by obtaining a memory mapping from the proper memory source.

Fiasco.OC has another feature, which aids our goal of transparently replicating binary applications: A thread can execute user-level code within an address space. If at any point in time, this thread causes a CPU exception, for instance by issuing a system call or raising a page fault, it is migrated to another process, where this exception can be handled before resuming execution. This whole mechanism [20] is fully transparent to the thread in question.

5. TRANSPARENT REDUNDANT MULTITHREADING

Romain provides tolerance against hardware SEUs by using software-implemented redundant multithreading to replicate program execution. The replication approach is transparent to user-level applications and allows execution of any program written to run on Fiasco.OC without relying on source code availability or cooperation. Our prototype implementation was done on x86/32, but there are only few platform-specifics in Romain and we are convinced that it will in the end work on all platforms that Fiasco.OC is running on, which includes "real" embedded platforms, such as ARM.

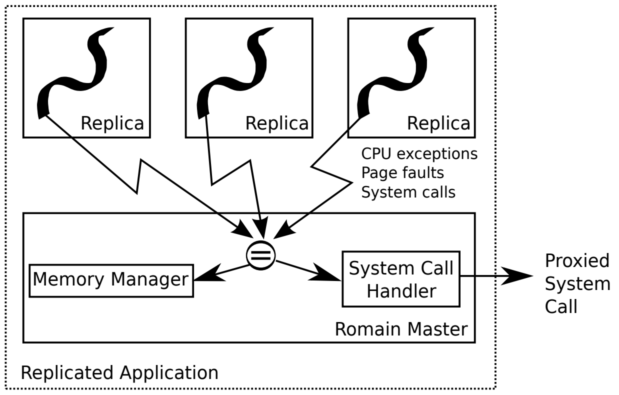


Figure 2: Romain architecture

Figure 2 depicts the internal layout of an application replicated using Romain. Every *replica* is run in a separate address space, leveraging hardware memory isolation to avoid propagation of an error from one replica to others. The amount of replicas is configurable and allows for n-way modular redundancy. A *master* process is responsible for managing the replicas of a single application, comparing their states, and performing recovery if necessary. It is furthermore the only part of the replicated application that performs any communication with other applications. This master is part of the RCB as defined in Section 4.1.

At application startup, the master serves as the program loader for the replicated application. It creates the initial replica address spaces and sets up an environment consisting of an initial stack, environment variables, and an initial set of capabilities. These actions are identical to those a program loader performs in the Fiasco.OC environment.

After the initial setup phase, the master creates one initial thread for each replica and configures it to start running within the respective replica address space. Whenever the replica thread raises a CPU exception, such as a page fault or a system call, the Fiasco.OC kernel migrates this thread to the master address space, where an exception handler is executed.

This handler function is the entry point to the second role performed by the master: runtime state comparison. A faulting replica thread is blocked until all other replica threads raise their next exception. The master then compares their states to make sure that all replicas raised the same exception and their architectural states match. The state comparison includes architectural registers, kernel-level exception state, and UTCB contents.

If the master finds the replica states to be identical, it is responsible for handling the respective exception. In Romain this handling is implemented by *observer modules*, which get notified whenever an exceptional situation is encountered. An observer handles the exception by inspecting the state of one replica and acting upon it. Then, the result of this operation (e.g., system call return values) are injected into the other waiting replicas by overwriting their thread states. This allows the master to remain in full control of all interactions between the replicated application and the rest of the system. After successful exception handling, every re-

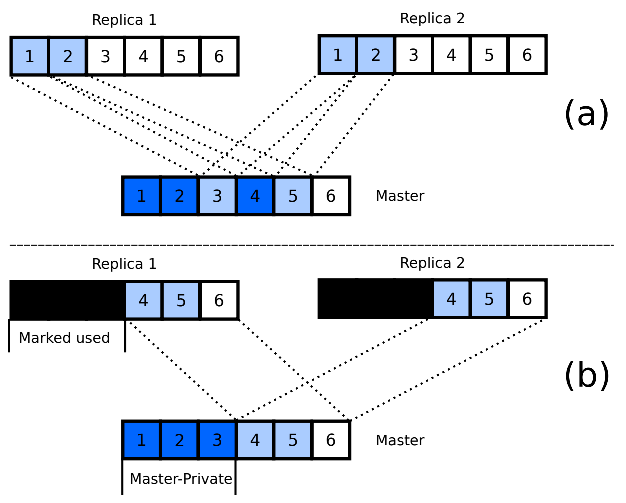


Figure 3: Matching capability tables between master and replicas

plica thread is migrated back to its respective address space and continues execution.

While the master is able to determine that the replicas' outputs to the external world match, it still needs to maintain control over inputs reaching the replicas. In Fiasco.OC this exclusively happens through synchronous system calls, which are covered by the exception handling mechanism. This is slightly different for shared memory, as we will discuss in Section 5.2.

5.1 Replica Resource Management

The exceptions caused by replicas and handled by the master fall into one of two categories: operations on kernel objects and operations on memory. Kernel objects are represented by capabilities, which Fiasco.OC maintains in a per-process capability table. To remain in control of all actions performed by a replica, the master needs to possess a capability to every object the replicas access. In addition, the master also requires capabilities to objects that are only used by itself. This means that over time the master's capability table will diverge from the replicas' ones.

To handle a replica's system call, the master needs to identify the kernel object the replica is referring to. This requires a data structure virtualizing each replica's capability table and mapping its entries to those owned by the master as shown in Figure 3 a). This approach complicates system call handling: for every system call, the master needs to identify all capabilities within the replica's architectural state and its UTCB. Then it must translate them to master capabilities and execute the system call. As the system call reply may also contain capabilities, these need to get translated back to replica capabilities before resuming execution.

To avoid the complexity of translating capabilities, Romain partitions the master's capability table as shown in Figure 3 b). One part is used by the master to obtain private capability mappings. This part is marked as reserved when setting up the initial environment for each replica, thereby making sure that replicas will only obtain mappings in the remainder of their capability tables. Due to this partitioning scheme, the master can carry out system calls on behalf of a replica and receive capability mappings into the replica-specific partition of its capability table. Capabilities are mapped to the same index in the master and the replicas. The master does not have to perform any translation upon a system call.

While this approach allows redirecting all system calls without modifying their parameters or emulating their behavior, there remains a subset of system calls that need to be emulated by the master. First, some objects exist locally in the replicated task and are therefore existent in every replica. Modifications to these objects need to be applied in all replicas. Second, all system calls relating to the layout of the replicas' address spaces and requests for memory mappings are executed by the master as well.

To manage memory at the user level, Fiasco.OC assigns every thread a memory manager capability, which is provided during application startup. This capability is used a) by a thread to attach a memory object to its address space in order to use it, and b) by the Fiasco.OC kernel to reflect page faults occurring at runtime. The Romain master virtualizes this capability for the replicas in order to remain in control of how replica address spaces are laid out.

The master maintains a representation of every replica's address space and distinguishes between read-only and writable regions. As discussed in Section 4 we rely on memory to be protected by ECC techniques. This allows the master to use a single copy of every read-only memory region and share it among all replicas. In contrast, the master allocates a dedicated copy of writable memory regions for each replica. This increases the memory overhead imposed by Romain. However, the alternative would be to maintain a single copy of the region, map it to replicas read-only and emulate the write operation whenever a write page fault occurred in this region. This *trap & emulate* approach would drastically increase the runtime overhead for replication.

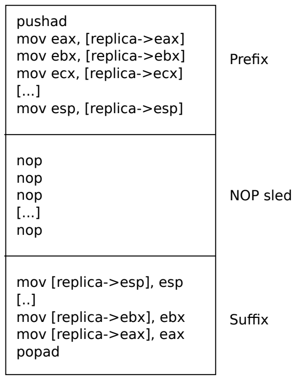
5.2 Shared Memory

The mechanisms discussed so far provide replicated execution under the assumption that all interactions between the replicated application and the outside world can be trapped and handled by the master. This assumption does not hold in the case of shared memory, as its content may be modified by a thread outside of the master's control at any time. The master still needs to make sure that whenever replicas access shared memory, they read the same data regardless of the timing or order of access. Therefore, in contrast to anonymous memory that is only used internally by the replicated application, a trap & emulate approach is required for handling shared memory regions.

Trap & emulate

Fiasco.OC applications obtain anonymous memory by invoking a memory allocator capability. The returned anonymous memory region is thereafter attached to the local address space by invoking the previously discussed memory manager capability. Romain virtualizes the memory allocator capability in order to keep track of all memory regions that have been allocated through it. Thereby the master is able to distinguish between anonymous memory that can be directly mapped to the replicas' address spaces and shared memory objects that require special handling.

When a replica attaches a shared memory region to its address space, the master does not add a memory mapping, but lets the replica continue execution as if a mapping had been established. Thereby every future memory access to



```
pushad
mov eax, [replica->eax]
mov ebx, [replica->ebx]
mov ecx, [replica->ecx]
[...]
mov esp, [replica->esp]

nop
nop
nop
[...]
nop

mov [replica->esp], esp
[..]
mov [replica->ebx], ebx
mov [replica->eax], eax
popad
```

Figure 4: Executing a replica's instruction in the master

the respective region will lead to a page fault that is then handled by the master in a trap & emulate observer module.

Emulating read and write operations on shared memory regions is expensive in terms of overhead and implementation complexity. It requires an instruction emulator which in our case needs to cope with the full semantics of x86/32 memory operations and addressing modes. As a matter of fact, we started implementing such an emulator using the `udis86` disassembler library [42], which consists of about 6,000 lines of code. We added an instruction emulator of about 500 lines of code, restricting it to handle the instructions `mov`, `push/pop`, `stosd`, `movsd`, and `call`. This emulator was by far not complete, but allowed to replicate a tiny Fiasco.OC application using trap & emulate for accessing memory. Still, a full emulator would add thousands of lines of code to the RCB. This does not fit well with our goal of minimizing the RCB.

Copy & Execute

Looking closer at shared memory accesses, an instruction emulator needs to perform two tasks: First, it needs to translate virtual addresses in the replica's address space to virtual addresses in the master's address space so that emulation operates on the proper region. Second, the emulator needs to perform the memory operation.

As the master is in full control of the replica's address space layout, it can make sure that shared memory regions are mapped to the same virtual address in the master and replica address spaces. Thereby, if a replica raises a page fault in shared memory, the master can directly deduce where to perform this operation in its own address space. To additionally avoid the complexity and overhead of emulating the current instruction, we decided to make the master execute this instruction on the fastest and most complete instruction emulator available to us: the physical CPU.

To execute the replica's instruction locally in the master, the master maintains a buffer as shown in Figure 4. It contains prefix code, a NOP sled, and suffix code. The prefix code is responsible for storing the master's architectural registers on the stack (using `pushad`) and copying the trapping thread's register state to the architectural registers. The suffix code is responsible for reverting these actions. The NOP sled is patched with the instruction from the remote replica. To do so, we determine the instruction length using the `MLDE32` instruction length decoder [44] and then copy the instruction into the NOP sled, before executing the instruction sequence in the buffer. Implementing this approach requires less than 300 lines of code and therefore minimizes the RCB complexity for emulating remote memory accesses.

Our solution has two limitations: if the remote instruction executed within the master performed a memory-indirect branch (such as '`jmp [eax]`'), execution would divert and never reach the suffix code. This can be mitigated by using CPU support for single-stepping to ensure that only one instruction is executed before control is handed back to the master's code. However, this would increase runtime overhead because of the need for handling the additional single-step interrupt. So far we did not find any use case, where an application makes use of indirect branch targets that are stored in shared memory and we assume that this would already be an exceptional case for non-replicated execution. Therefore, we chose to not address this limitation in our implementation of Romain yet.

Furthermore, executing the shared memory access directly in the master does not replicate this single instruction. Therefore, execution is vulnerable against hardware errors for this single instruction.

Leveraging application knowledge

As we will see in Section 6.2, executing the replica's shared memory instruction in the master yields a better performance than emulating the instruction using a custom-built emulator. Nevertheless, it is still much slower than letting the replica execute shared-memory access directly without any interaction with the master.

In some scenarios, it is possible to leverage application knowledge to decrease the replication overhead in the presence of shared memory usage. One such scenario is the use of shared memory regions that are used mostly for static, read-only data.

The Fiasco.OC kernel provides applications with a kernel info page (KIP), which contains mostly static information, such as the kernel version number and configured features. Its most prominent content, is the code for entering the kernel, which is provided by Fiasco.OC similar to Linux' vsyscall page. The KIP also contains dynamic data: whenever the kernel decides to schedule a thread, it updates a time field within the KIP, which the thread can then use to obtain the current wall clock time through the C library's `gettimeofday` function. In a replicated application, the master needs to consider the dynamic time an external input and only if we can make sure that all replicas see the same value upon an access, we can make sure that their behavior will be consistent after this event.

The static nature of its content and the requirement to access the KIP for every system call prohibit emulating accesses to it. Instead, we currently statically analyse the replicated binary for memory accesses to the well-known clock address within the KIP. We then place a software breakpoint on these instructions and provide an observer module that emulates accesses to the clock field. While patching with interrupt instructions is a generic solution, which can be applied to an arbitrary amount of mostly-static shared memory regions, it still requires the binary to be analysed beforehand. An alternative would be to use hardware breakpoints provided by most of today's CPUs. Then, one could set a data breakpoint on the addresses of interest and provide handler code to emulate the corresponding accesses. However, we did not do so yet, because hardware breakpoints are a limited resource.

5.3 Recovery

Romain uses n-way modular redundancy and can therefore perform majority voting if it detects replica states to mismatch. Faulty replica state can then be overwritten with the state of a correct replica. This approach has the advantage of nearly instant recovery and does not require a combination with sophisticated checkpoint and rollback techniques, which could impose additional overhead.

As we will see in Section 6, replicated execution leads to runtime overhead. Therefore, it depends on the workload and usage scenario, whether a user is willing to run the minimum amount of three replicas necessary for using majority voting. One might instead opt to use double-modular redundancy and only detect errors. In this case, we believe Romain can be paired with restart strategies such as the one used by Minix [15], or application-level checkpointing [1].

6. EVALUATION

One fundamental assumption we based our work on was that a substantial amount of SEUs does not lead to erroneous behavior and it therefore makes sense to delay state comparison between replicas in order to decrease our solution's runtime overhead. We validate this assumption using fault injection experiments in Section 6.1. Thereafter, we evaluate Romain's overhead for double- and triple-modular redundancy in Section 6.2 and give an overview of our solution's code complexity in Section 6.3.

6.1 Fault Injection Experiments

We implemented a fault injection suite as an observer module within Romain. Using this suite, we simulated SEU injections of four types of errors in functional units. The error types were selected to mimic the hard error types injected by Li [21]:

- *Register flip:* We set a breakpoint on a random instruction from the application's code section and in the breakpoint handler trigger a flip of a random bit in a random general-purpose register to simulate SEUs in the register file.
- *Decode:* We simulate an SEU during instruction decoding by sampling a random instruction, flipping a bit in this instruction, single-stepping over the newly generated instruction and reverting the bit flip afterwards.
- *ALU:* We select a random arithmetic instruction and upon encountering it, randomly do one of three things: (1) modify the instruction to perform another arithmetic operation, (2) randomly flip a bit in one of the input operands, or (3) modify the output target. This simulates SEUs occurring in the arithmetic-logic unit.
- *RAT:* The register-allocation table (RAT) maintains a mapping between the general purpose registers (GPRs) exposed to an application and the physical register file provided by the underlying CPU. We simulate an SEU in the RAT by randomly intercepting an instruction that uses register operands. We then modify this instruction to use a random index within the RAT – in 10 % of the cases another register from the available GPRs is selected, in the other 90 % the register access is skipped, simulating writing to an unused register file index. This is a simplification: the register file entry might be used by another hardware thread and thereby the write would influence the computations of this thread.

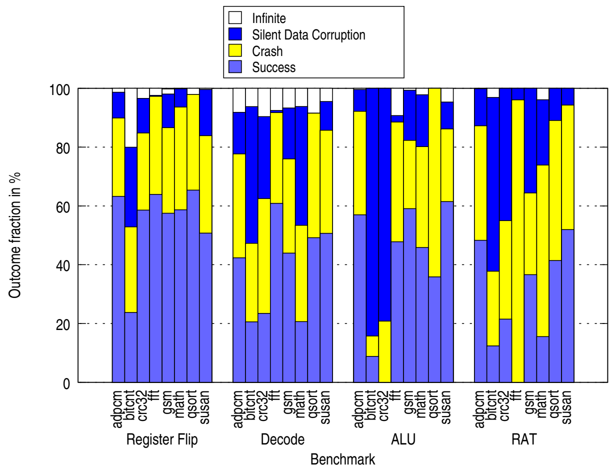


Figure 5: Distribution of benchmark outcomes for fault injection experiments

As Romain's purpose is to handle hardware errors occurring during the execution of user-level applications, we performed fault injection experiments targetting such applications. We ran fault injection campaigns on applications from the MiBench [13] embedded benchmark suite, observed the outcome of the experiments and classified them into four categories:

- *Success:* Execution continued and produced a correct result. This indicates that the SEU did not influence the outcome of execution at all.
- *Crash:* Execution terminated prematurely with an abnormal result. Such SEUs represent fail-crash errors which Romain immediately detects.
- *Infinite:* Execution did not terminate within a specified amount of time. This amount was selected to be double the amount of an error-free benchmark run. Such errors most likely represent the program being stuck in an infinite loop. Romain can detect them using a watchdog mechanism.
- *Silent data corruption (SDC):* Execution continued and produced an incorrect result. Romain detects such errors at the point the results are externalized, e.g., by making a system call.

We selected 8 benchmarks from the MiBench suite for fault injection experiments and performed injection runs for each of the error types and applications. On average we ran 7,000 GPR injections, 3,000 instruction flips, 2,000 ALU flips, and 750 RAT flips for each benchmark. The results in Figure 5 show that depending on the benchmark up to 60 % of the fault injections resulted in no observable misbehavior by the application.

It should be noted, that our fault injection campaigns targetted applications of little complexity (leading to few potential candidates for fault injection) and were biased in the way we selected the fault injection point, because we always injected a fault on the first hit of an instruction. We believe that these properties make our results appear overly optimistic in comparison to the studies we cited in Section 3. Nevertheless, the combination of our experiments and those

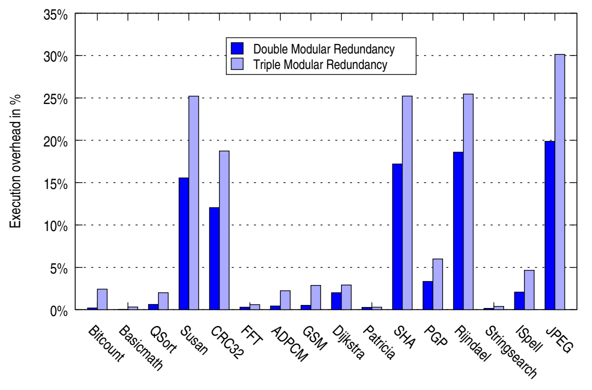


Figure 6: MiBench execution overheads for double- and triple-modular redundancy using Romain

Access type	Execution time
Direct mapping	0.16 s
Trap & emulate	400.39 s
Copy & execute	219.20 s

Table 1: Execution times for writing a 256 MB memory chunk word-wise using direct mapping and the two evaluated shared memory access mechanisms

studies confirms the assumption that a significant amount of transient faults causes no visible application misbehavior.

Instruction-level state comparison, such as lock-stepping or compiler-generated signatures, would detect mismatches between replicas here and trigger recovery (implying additional runtime overhead), whereas Romain continues execution unnotified. Crashes and SDC, which Romain also detects at the system call level, make up most of the remainder of the failures.

6.2 Execution overhead

To evaluate Romain's overhead, we ran the benchmarks from the MiBench suite as a single instance, as well as in double- (DMR) and triple-modular redundancy (TMR) mode and compared their runtimes. We executed this test on a computer with 12 physical Intel Core2 CPUs running at 2.6 GHz. Hyperthreading was turned off and every replica as well as the master were pinned to a dedicated CPU. Figure 6 shows the normalized runtime overheads, which vary between 0.5 % and 30 %, and in many cases are below 5 %. These results are comparable with Shye's PLR work [38].

We further investigated what behavioral difference causes a benchmark to exhibit 30 % TMR overhead in comparison to having less than 5 % overhead. We counted the amount of memory management requests the master handles during a benchmark run and normalized the results with respect to benchmark execution times. Benchmarks allocating lots of dynamic memory cause higher rates of memory-related exceptions (page faults, mapping requests) that need to be handled. As can be seen in Figure 7, these benchmarks yield higher runtime overheads when running replicated.

Also in line with other replication approaches, CPU utilization and memory overhead are multiplied when using replication. We try to limit memory overhead by sharing read-only regions among replicas.

We additionally performed a microbenchmark to estimate the overhead of trap & emulate on shared memory accesses.

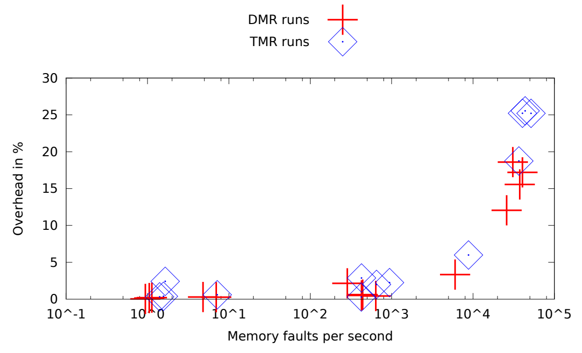


Figure 7: Normalized overhead is related to the amount of memory management requests (logarithmic x scale)

Base code (main, logging, locking)	325
Application loading	375
Replica management	628
Redundancy	153
Memory management	445
System call handling	311
Shared memory handling	281
Total Romain	**2,518**
Fault injection module	668
GDB server stub	1,304

Table 2: Romain complexity in lines of code

We measured the time for `memset`ting 256 MB of memory with a) the memory directly mapped to the replica, b) the memory accesses emulated with our trap & emulate approach, and c) the memory accesses handled using the copy & execute mechanism discussed in Section 5.2. We see that handling each memory access as an exception in a different process is much more expensive than direct accesses. However, we also see that our copy & execute approach is not only less complex, but also faster than using an instruction emulator.

6.3 Complexity

The main reason to implement Romain as an OS service was to keep the solution's complexity, and thus the RCB, minimal. Table 2 gives an overview about our tool's complexity in terms of lines of code. The features described in this paper amount to 2,518 lines of code. To put these numbers into a context, we also show the lines of code needed for the fault injection module used in Section 6.1 and for a GDB stub we implemented to aid debugging on top of Romain.

Especially for the system call handling module, we added code only as far as it was required by our test applications. We expect the lines of code to grow over time as we explore further use cases. Additionally, adding multithreading support to Romain will realistically also add several hundreds of lines of code.

7. LIMITATIONS AND FUTURE WORK

In its current implementation, Romain has three major limitations, which we intend to address in future work. First, so far we enabled replicated execution only for the single-threaded applications from the MiBench benchmark suite.

While this is in line with other related work [11, 33, 34, 38, 47] and while these benchmarks represent widespread workloads, it is insufficient with respect to our goal of addressing operating system services.

To successfully replicate OS services, Romain needs to support multithreaded execution, which is inherently nondeterministic, complicating state comparison between replicated threads. Additionally, multithreading also includes the problem of handling memory shared between threads of a replica. Both issues can be addressed by making execution of replicated threads deterministic, because then the events generated by replicated threads will match their replicated counterparts and racy memory accesses will produce the same outcome in all replicas. (Note, that we don't want to fix data races, we only want to verify that racy accesses produce the same outcome in all replicas.)

These problems are similar to the issues solved by deterministic multithreading [22, 28]. They rely on instrumenting synchronization events to make sure that all executions chose the same path. In terms of Fiasco.OC, these synchronization points are exposed through kernel objects, which Romain is able to instrument at the system call boundary.

Enhancing Romain to support device driver software, requires taking care of I/O memory and registers. These are represented by kernel objects and memory pages, both of which are in control of the master process. However, performing replicated I/O accesses implies all kinds of troubles because I/O devices' behavior may differ from traditional memory. We think that the copy&execute method presented in Section 5.2 may be used to handle I/O operations.

We will furthermore look into decreasing memory-related overhead. To decrease space overhead, we will use copy-on-write mappings of writable memory regions instead of dedicated copies as they are used right now. To reduce the runtime overhead related to handling memory exceptions, we will look into memory management strategies that allow handling page-faults at a coarser granularity. For instance mapping super-pages instead of single 4 kB pages might decrease the amount of exceptions that need to be handled and therefore decrease the related runtime overhead.

Close integration with Fiasco.OC enables us to minimize Romain with respect to the RCB, but leaves open the question whether our approach can be transferred to other systems. We will investigate whether similar mechanisms to ours can be implemented based on Linux' `ptrace` or virtualization features.

8. CONCLUSION

In this paper we presented Romain, a framework that provides software-implemented redundant multithreading to unmodified binary-only applications on top of the Fiasco.OC operating system. The framework allows to detect and recover from hardware single-event upsets. The induced runtime overhead is less than 30 %, and in most cases even less than 5 % for the MiBench benchmark suite. The required additions to reliable computing base could be minimized to less then 3,000 lines of code. Romain therefore provides the same features as previous works at lower complexity.

Romain is part of the L4 Runtime Environment (L4Re) running on Fiasco.OC. Download and build instructions can be obtained from `http://www.tudos.org/l4re`.

Acknowledgment

This work was partially supported by the German Research Foundation (DFG) as part of the priority program "Dependable Embedded Systems" (`spp1500.itec.kit.edu`). We furthermore thank our colleagues Carsten Weinhold, Adam Lackorzynski, Michael Roitzsch and Thomas Knauth for their feedback on drafts of this paper.

9. REFERENCES

[1] Ansel, J., Arya, K., and Cooperman, G. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *23rd IEEE International Parallel and Distributed Processing Symposium* (Rome, Italy, May 2009).

[2] Arlat, J., Fabre, J.-C., Society, I. C., Rodriguez, M., and Salles, F. Dependability of COTS microkernel-based systems. *IEEE Transactions on Computers 51* (2002), 138–163.

[3] Aron, M., Deller, L., Elphinstone, K., Jaeger, T., Liedtke, J., and Park, Y. The SawMill framework for virtual memory diversity. In *Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference* (Bond University, Gold Coast, QLD, Australia, Jan. 29–Feb. 2 2001).

[4] Austin, T. DIVA: a reliable substrate for deep submicron microarchitecture design. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on* (1999), pp. 196–207.

[5] Bartlett, J. F. A nonstop kernel. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1981), SOSP '81, ACM, pp. 22–29.

[6] Bernick, D., Bruckert, B., Vigna, P., Garcia, D., Jardine, R., Klecka, J., and Smullen, J. Nonstop: Advanced architecture. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on* (june-1 july 2005), pp. 12–21.

[7] Borkar, S. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro 25*, 6 (Nov.-Dec. 2005), 10–16.

[8] Bressoud, T. C., and Schneider, F. B. Hypervisor-based fault tolerance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 1–11.

[9] Brown, J., and Knight, T. F. A minimal trusted computing base for dynamically ensuring secure information flow. Tech. rep., 2001.

[10] David, F. M., Chan, E. M., Carlyle, J. C., and Campbell, R. H. CuriOS: Improving Reliability through Operating System Structure. In *USENIX Symposium on Operating Systems Design and Implementation* (San Diego, CA, December 2008), pp. 59–72.

[11] Fetzer, C., Schiffel, U., and Süsskraut, M. AN-encoding compiler: Building safety-critical systems with commodity hardware. In *Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security* (Berlin, Heidelberg, 2009), SAFECOMP '09, Springer-Verlag, pp. 283–296.

[12] Gray, J. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems* (1986), pp. 3–12.

[13] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop* (Washington, DC, USA, 2001), IEEE Computer Society, pp. 3–14.

[14] Hendricks, J., and van Doorn, L. Secure bootstrap is not enough: shoring up the trusted computing base. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop* (New York, NY, USA, 2004), EW 11, ACM.

[15] Herder, J. N. *Building a dependable operating system: Fault Tolerance in MINIX3*. Dissertation, Vrije Universiteit Amsterdam, 2010.

[16] IBM. PowerPC 750GX Lockstep facility. IBM Application Note, 2008.

[17] IBM. z/OS – a smarter operating system for smarter computing. `http://www-03.ibm.com/systems/z/os/zos/`, 2011.

[18] Kadav, A., Renzelmann, M. J., and Swift, M. M. Tolerating hardware device failures in software. *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (2009), 59.

[19] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: Formal verification of an OS kernel. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)* (Big Sky, MT, USA, Oct. 2009), ACM, pp. 207–220.

[20] Lackorzynski, A., Warg, A., and Peter, M. Generic Virtualization with Virtual Processors. In *Proceedings of Twelfth Real-Time Linux Workshop* (Nairobi, Kenya, October 2010).

[21] Li, M.-L., Ramachandran, P., Sahoo, S. K., Adve, S. V., Adve, V. S., and Zhou, Y. Understanding the propagation of hard errors to software and implications for resilient system design. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2008), ASPLOS XIII, ACM, pp. 265–276.

[22] Liu, T., Curtsinger, C., and Berger, E. D. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 327–336.

[23] Meixner, A., and Sorin, D. J. Detouring: Translating software to circumvent hard faults in simple cores. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)* (2008), pp. 80–89.

[24] Mukherjee, S. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[25] Nassif, S. R. The light at the end of the CMOS tunnel. In *Int. Conf. on Application-specific Systems Architectures and Processors* (july 2010), pp. 4–9.

[26] Oh, N., Shirvani, P., and McCluskey, E. Control-flow checking by software signatures. *IEEE Transactions on Reliability 51*, 1 (mar 2002), 111–122.

[27] Oh, N., Shirvani, P. P., and McCluskey, E. J. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability 51* (Mar 2002), 63–75.

[28] Olszewski, M., Ansel, J., and Amarasinghe, S. Kendo: efficient deterministic multithreading in software. *SIGPLAN Not. 44* (Mar. 2009), 97–108.

[29] Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J., and Muller, G. Faults in Linux: Ten years later. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2011), ASPLOS '11, ACM, pp. 305–318.

[30] Patterson, D. A., Gibson, G., and Katz, R. H. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1988), SIGMOD '88, ACM, pp. 109–116.

[31] Postel, J. Transmission Control Protocol. RFC 793 (Standard), Sept. 1981. Updated by RFCs 1122, 3168, 6093.

[32] Reick, K., Sanda, P., Swaney, S., Kellington, J., Mack, M., Floyd, M., and Henderson, D. Fault-tolerant design of the IBM Power6 Microprocessor. *IEEE Micro 28*, 2 (march-april 2008), 30–38.

[33] Reinhardt, S. K., and Mukherjee, S. S. Transient fault detection via simultaneous multithreading. *SIGARCH Comput. Archit. News 28* (May 2000), 25–36.

[34] Reis, G. A., Chang, J., Vachharajani, N., Rangan, R., and August, D. I. SWIFT: Software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization* (2005), IEEE Computer Society, pp. 243–254.

[35] Ryzhyk, L., Chubb, P., Kuz, I., Le Sueur, E., and Heiser, G. Automatic device driver synthesis with Termite. *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles SOSP '09* (2009), 73.

[36] Saggese, G. P., Wang, N. J., Kalbarczyk, Z. T., Patel, S. J., and Iyer, R. K. An experimental study of soft errors in microprocessors. *IEEE Micro 25* (November 2005), 30–39.

[37] Schroder, D. K. Negative bias temperature instability: What do we understand? *Microelectronics Reliability 47*, 6 (2007), 841–852.

[38] Shye, A., Moseley, T., Reddi, V. J., Blomstedt, J., and Connors, D. A. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (Washington, DC, USA, 2007), DSN '07, IEEE Computer Society, pp. 297–306.

[39] Singaravelu, L., Pu, C., Härtig, H., and Helmuth, C. Reducing TCB complexity for security-sensitive applications: three case studies. *SIGOPS Oper. Syst. Rev. 40* (April 2006), 161–174.

[40] Steinberg, U., and Kauer, B. NOVA: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems* (New York, NY, USA, 2010), EuroSys '10, ACM, pp. 209–222.

[41] Taber, A., and Normand, E. Single event upset in avionics. *IEEE Transactions on Nuclear Science 40*, 2 (apr 1993), 120–126.

[42] Thampi, V. udis86 - disassembler library for x86 and x86-64. `http://udis86.sourceforge.net/`, 2009.

[43] TU Dresden OS Group. L4/Fiasco.OC microkernel. `http://www.tudos.org/fiasco`, 2012.

[44] uNdErX. Micro length-disassembler engine 32. `http://vx.netlux.org/vx.php?id=em24`, 2004.

[45] Venkatasubramanian, R., Hayes, J., and Murray, B. Low-cost on-line fault detection using control flow assertions. In *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE* (july 2003), pp. 137–143.

[46] Vogt, D., Döbel, B., and Lackorzynski, A. Stay strong, stay safe: Enhancing reliability of a secure operating system. In *Proceedings of the Workshop on Isolation and Integration for Dependable Systems (IIDS 2010), Paris, France, April 2010* (New York, NY, USA, 2010), ACM.

[47] Wang, C., Kim, H.-s., Wu, Y., and Ying, V. Compiler-managed software-based redundant multi-threading for transient fault detection. In *Proceedings of the International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2007), CGO '07, IEEE Computer Society, pp. 244–258.

[48] Wang, N., Fertig, M., and Patel, S. Y-branches: when you come to a fork in the road, take it. In *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on* (sept.-1 oct. 2003), pp. 56–66.

[49] Zhu, D., Melhem, R., and Mosse, D. The effects of energy management on reliability in real-time embedded systems. In *IEEE/ACM International Conference on Computer-Aided design* (Washington, DC, USA, 2004), ICCAD '04, IEEE Computer Society, pp. 35–40.

[50] Ziegler, J. F., and Lanford, W. A. Effect of cosmic rays on computer memories. *Science 206*, 4420 (1979), 776–788.