# Can We Put Concurrency Back Into Redundant Multithreading?

Björn Döbel, Hermann Härtig
TU Dresden, Germany
{doebel,haertig}@tudos.org

## Abstract

Software-implemented fault tolerance (SIFT) mechanisms allow to tolerate transient hardware faults in commercial off-the-shelf (COTS) systems without using specialized resilient hardware. Unfortunately, existing SIFT methods at both the compiler and the operating system levels are often restricted to single-threaded applications and hence do not apply to multithreaded software on modern multicore platforms.

We present *RomainMT*, an operating system service that provides replication for unmodified multithreaded applications. Replicating these programs is challenging, because scheduling-induced non-determinism may cause replicated threads to execute different valid code paths. This complicates the distinction between valid behavior and the effects of hardware errors.

*RomainMT* solves these problems by transparently making multithreaded execution deterministic. We present two alternative mechanisms that differ in the assumptions made about the respective applications and investigate their performance implications. Our evaluation using the SPLASH2 benchmark suite shows that the overhead for triple-modular redundancy (TMR) is 24% for applications with two application threads and 65% for four application threads.

## 1 Introduction

The hardware components that form modern multiprocessor systems can suffer from transient errors caused by cosmic radiation, hardware aging, and thermal effects [7, 38, 41]. With decreasing feature sizes and increasing hardware complexity these single-event upsets (SEUs) are no longer only a problem for avionics and space travel. Catastrophic SEUs were reported in data centers [19], scientific computing [17], as well as automotive systems [45]. Future technology scaling will amplify this problem: at a structure size of 8 nm only half of a CPU's circuits may be powered at the same time [15]. The resulting fluctuations in supply voltage will increase hardware fault rates. Transient errors are furthermore not limited to CPUs. Industry studies showed that hardware components, such as memory and disks, also exhibit transient failures before breaking completely [19, 34].

Researchers and vendors devised fault tolerance solutions ranging from specialized hardware [3] to compiler and runtime solutions for commercial-off-the-shelf (COTS) platforms. Such mechanisms often use one or more forms of redundancy either in time (repeated execution [36]), space (replication on different hardware units [40]), or data (fault-tolerant encoding [16]).

Our work focuses on fault tolerance for modern multicore computers, which are used in today's high-end embedded and mobile platforms. These devices share two main properties that limit the choice of fault tolerance method: first, they consist solely of COTS hardware components (and IP cores) and we cannot rely on fault tolerance through hardware extensions. Second, such systems mainly run binary software that is obtained through internet downloads and application stores. This property rules out compiler-level solutions, which usually require the program's source code to be available. We address these issues using software-implemented binary-level replication based on our previous *Romain* replication service [12].

*Romain* as well as many other SIFT mechanisms were evaluated using single-threaded workloads only [12, 16, 36, 40]. This contradicts the needs of modern software: even in the embedded domain we find multicore computers that can be efficiently used by multithreaded applications. These programs often exhibit non-determinism that makes replication challenging.

In this paper we present *RomainMT*, an extension to *Romain* that replicates multithreaded binary applications. *RomainMT* leverages deterministic multithreading techniques from Olszewski et al.'s Kendo [33] to transparently make multithreaded software deterministic.

This paper makes the following contributions:

1. We extend *Romain* to support multithreaded replication for `pthread` [26] applications. We intercept synchronization operations in replicas using debug traps and use the replication master to enforce determinism.
2. We show that this straightforward solution leads to high runtime overheads. We then present an alternative solution that leverages a replication-aware thread library to lower this runtime cost.
3. Evaluating our implementation with the SPLASH2 benchmark suite [44] we find the overhead for triple-modular redundant (TMR) replication is 24% when replicating two application threads and 97% when replicating four application threads. We find that lock density is a main contributor to overhead.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ESWEEK'14, October 12 - 17 2014, New Delhi, India Copyright is held by the owner/author(s). Publication rights licensed to ACM. ACM 978-1-4503-3052-7/14/10...$15.00. http://dx.doi.org/10.1145/2656045.2656050

In the remainder of this paper we first give an overview of *Romain*'s approach to OS-assisted replication and relate our work to other research in the context of fault tolerance against SEUs and deterministic multithreading in Section 2. We introduce *enforced* and *cooperative determinism* as two alternatives to enforcing deterministic replicated execution in Section 3 and evaluate our implementation using the SPLASH2 benchmarks in Section 4. In Section 5 we discuss limitations of our approach.

# 2 Related Work

We now introduce OS-assisted replication as implemented by *Romain* and motivate the underlying SEU fault model. Thereafter we describe how multithreading affects replicated execution and how deterministic multithreading methods can alleviate these issues.

## 2.1 SEU Fault Tolerance

Single-event upsets are used to model *transient* errors that stem from cosmic radiation [41], hardware aging [38], and thermal effects. SEUs are distributed uniformly over time and chip area, but their probability varies with geographical location and altitude. In line with related work [36] we make the assumption that SEUs happen seldom so that only a single error is active at one point in time. In contrast to related work, we do *not* assume memory to be protected against SEUs using error-correcting codes (ECC) [29] as such hardware is often too expensive (in terms of energy consumption and chip area) for embedded or mobile devices.

A common approach to tolerating SEUs is to increase fault masking by applying redundancy. This can for instance be achieved by adding custom hardware units, such as DIVA's checker cores [3] or Razor's flip-flops for monitoring gate switch times [14]. High-availability systems, such as IBM's z-Series [20] and HP NonStop [6] additionally provide software-level solutions, which increase fault tolerance if developers use specific programming models. In contrast to these mechanisms, our goal is to provide fault tolerance for unmodified binary applications on top of COTS hardware.

At the software level, compiler developers have proposed techniques to generate fault-tolerant code. These approaches include adding data encoding [16] and generating additional code that checks computed data [36] and the validity of jump targets at runtime [32]. These compiler techniques require the target application's source code to be available. In contrast to compiler methods, we aim to provide fault tolerance for practical systems where software is often distributed in binary form and comes from a variety of vendors so that enforcing common compiler constraints is not a viable solution.

Studies [1,37,43] tried to understand the impact of SEUs on system behavior. One major result from these studies is that a significant amount of SEUs (usually around 20-30%) does not lead to misbehavior of the affected system. These benign faults are masked by hardware or software before producing wrong results. *Redundant multithreading (RMT)* [35] tries to leverage this fact: replicas execute in independent threads as long as they only compute internal results. The threads' states are only compared whenever they try to generate externally visible results, for instance by performing system calls.

RMT implementations exist at the hardware [35] and compiler [42, 46] levels and in the form of runtime extensions for Linux [40], L4 [12], and MPI [17]. The runtime-level approaches share our goal of providing replication transparently to arbitrary binary applications. Unfortunately, most software RMT implementations share one limitation: they lack support for replicating multithreaded applications. The purpose of this paper is to dispose of this limitation.

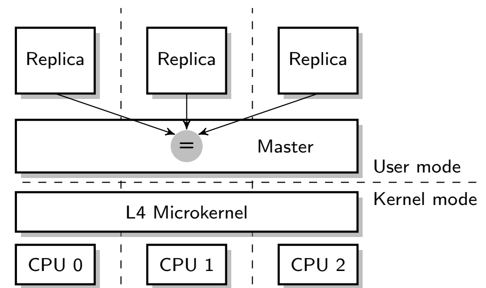


**Figure 1: Replicated Application**

## 2.2 OS-Assisted Replication

Due to space constraints we can only give a brief overview of how *Romain* achieves replication as an operating system service. For details please refer to our original paper [12].

*Romain* replicates unmodified binary applications on top of the L4 microkernel as shown in Figure 1. A user-level master process launches $N$ instances (*replicas*) of a binary application. Each replica executes in its own address space to achieve fault isolation. Replicas are scheduled on different physical CPUs to decrease runtime overhead through parallel execution.

The L4 microkernel provides a mechanism that allows the master process to intercept any externalization event, such as system calls, page faults, and other CPU traps. Apart from this mechanism, *Romain* does not require specialized features and runs on COTS x86-32 hardware.[1]

*Romain* replicates single-threaded applications. The master process ensures that replicas obtain identical inputs even from non-deterministic function calls, such as `gettimeofday()`. As a result, correctly functioning replicas execute the same code and hence generate the same externalization events. Whenever a replica raises an externalization event, the master blocks it until all replicas have reached their next externalization event. The master then compares the replicas' architectural states to detect and correct errors before handling these events (e.g., by resolving page faults or redirecting a system call to the L4 kernel).

Replicated execution on different CPUs allows the master process to detect and correct errors that arise in functional units of a CPU, such as bit flips in registers or errors during instruction decoding. To also protect replicas against SEUs in memory, *Romain* adapts replica memory management. Every replica works on a dedicated copy of its memory. Hence, a memory fault will only affect a single replica and the resulting misbehavior can still be detected and corrected by the master process.

The replication framework does not protect the *Romain* master process as well as the underlying microkernel against hardware errors. This part of the system, called the *Reliable Computing Base* (RCB), needs to be protected separately. Protection of the RCB [10] is orthogonal to the work presented in this paper.

[1] L4 itself also runs on x86-64 and ARM. We therefore believe that our solution is also applicable to these architectures with the respective porting effort.

```
int x = 1;
pthread_mutex_t m = PTHREAD_MUTEX_DEFAULT;

void *thread_A(void *data)
{
  pthread_mutex_lock(&m);
  x = x + 1;
  pthread_mutex_unlock(&m);
  return NULL;
}

void *thread_B(void *data)
{
  pthread_mutex_lock(&m);
  x = x * 2;
  pthread_mutex_unlock(&m);
  return NULL;
}
```

**Listing 1: Data-race free thread example**

## 2.3 The Multithreading Problem

Multithreaded programs consist of concurrently executing threads, which are scheduled non-deterministically by the underlying OS. Due to this fact, in every application run, threads may obtain inputs in different order and in turn generate different output depending on scheduling order.

Listing 1 shows an example to illustrate the problem. Suppose there are two threads executing the functions concurrently. Mutex `m` makes sure that accesses to the variable `x` are serialized. Depending on the exact scheduling, the threads may execute either in order A;B (yielding `x == 4`) or B;A (yielding `x == 3`).

Now assume we replicate this program and run two replicas. One replica may execute A;B, while the second replica may execute B;A. When the variable `x` is then used for an externalization event, the replication master will detect this divergence and flag a potential hardware fault. In the best case this will cause unnecessary runtime overhead for recovery. In the worst case, the replicas may have diverged in a way that does not allow recovery at all.

To solve this problem, we need to make sure that threads behave deterministically. Deterministic multithreading provides this property and has been extensively researched. We now review deterministic multithreading approaches to find out which best fits our goal of transparently replicating binary-only multithreaded applications.

## 2.4 Deterministic Multithreading

Eve [23] provides deterministic execution in a distributed environment and leverages request batching to optimize performance. Batching is unfortunately no option for replicating applications as an operating system service, because we cannot batch system calls without changing system semantics.

Tern [9] and Storyboard [22] exploit application knowledge to determine possible schedules and modify the locking library to enforce those schedules by assigning locks to threads in a pre-determined order. The required pre-analysis places a burden on the user and during analysis the application is not protected against hardware faults.

Determinator [4], dOS [5], and Conversion [28] provide operating system interfaces for applications to fork private copies of memory regions, modify these regions, and later merge them back into the original copies. Replication on these systems does not suffer from non-determinism. However, these solutions restrict developers to use a specific parallel programming model that leverages fork/join memory semantics.

DThreads [27] provides a replacement for the commonly used `pthread` thread library and ensures deterministic ordering of memory accesses by transparently providing such fork/join semantics. DThreads' threads work on private memory copies that are deterministically merged whenever a `pthread` synchronization operation is called. Unfortunately, maintaining per-thread memory copies increases applications' memory footprint: In the worst case, running $N$ threads in DThreads will require $N$ times the amount of memory as the original program. If we combine this approach with *Romain*'s replication and run $M$ replicas, we end up requiring $M \times N$ memory. This may seriously restrict the applicability of such a solution in embedded systems, where additional memory may be scarce.

Kendo [33] provides a weaker form of determinism. A multithreaded application that is free of data races under a given locking scheme will execute deterministically if we make sure that locks are acquired in the same order upon every run. Kendo threads run in the same address space and share global resources. Kendo furthermore provides lower runtime overheads for deterministic execution. We therefore decided to accept the restriction to race-free programs and base our replicated multithreading solution on Kendo.

While deterministic multithreading research is often motivated using replication as an example, only few solutions actually implement this use case. One of these few is Mushtaq's extension to Linux, which provides a leader-follower-based multithreaded replication scheme similar to *RomainMT* [30]. Their mechanism differs from *RomainMT* in that they use checkpoint/rollback for recovery. Predictions about the future of checkpoint/rollback show that this overhead may become dominating in future highly parallel systems [13]. In contrast, *RomainMT* provides the user with the option to choose between double-redundancy (costing fewer resources, but requiring an additional checkpoint mechanism) and triple-modular redundancy (allowing for forward recovery by majority voting).

Mushtaq's work has another drawback: they use `fork()`-based checkpoints in Linux. The checkpointed address space is only created as a copy-on-write copy of the caller. Hence, all memory that is never modified does not need to be copied, which reduces runtime overhead. While this approach is fast, it makes the solution vulnerable to memory SEUs. Whenever an untouched page gets corrupted and leads to a program failure, their solution rolls back to the last checkpoint and this checkpoint will suffer from corruption as well. Unfortunately, the authors discuss neither this issue nor its implications (reliance on ECC-protected memory) in their paper.

# 3 Multithreaded Replication

*RomainMT* shares Kendo's assumption that applications are race-free under a given locking scheme. This means that whenever two threads access the same location in shared memory, they use a pair of lock/unlock operations to protect this access. We base our work on the commonly used `pthread` library and its mutex operations. Thereby, our findings also apply to higher-level thread packages, such as Intel's thread building blocks [21], which internally use `pthread`.

*RomainMT* makes use of the fact that on modern operating systems vendors distribute dynamically linked binary applications. These programs assume that the platform vendor provides a concrete instance of the most common libraries, such as the `pthread` library. By providing a deterministic `pthread` library on our target system we can make

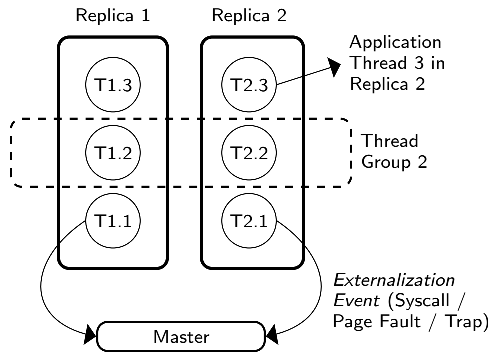


**Figure 2: Terminology overview**

all dynamically linked programs deterministic and still do not require modifications to the actual applications.

A closer look at four different Linux installations with several thousands of applications shows that this approach covers all but four applications and therefore suffices for our needs. We also implemented a modification to *RomainMT* that uses binary-patching to redirect statically linked `pthread` calls to our deterministic thread library, but we do not discuss this solution any further for space reasons.

We now describe how *RomainMT* extends *Romain* to achieve deterministic multithreaded replication. We first present the main terminology in Section 3.1. In Section 3.2 we describe how *enforced determinism* transforms lock operations into externalization events that the replication master uses to make execution deterministic. Section 3.3 then presents *cooperative determinism*, which avoids this expensive transformation.

## 3.1 Terminology

In the context of this paper a "thread" has different roles: first, a concurrent application executes multiple threads inside its address space. Second, the replication service creates multiple replicas of the application and a copy of each application thread exists inside each replica.

Figure 2 gives an overview of the terminology we use in the rest of this paper. A single application instance (its threads and the respective address space) is called a *replica*. Each replica executes multiple *application threads*. The set of threads that conceptually execute the same code within different replicas are called a *thread group*.

Once we achieve deterministic multithreading, the threads of each thread group will obtain identical inputs, generate identical output and can therefore be compared for error detection. Each thread group is handled independently by the *master* process. While one thread group's externalization event is handled, the other thread groups continue to execute or raise and handle different events.

## 3.2 Enforced determinism

Our first approach to enable multithreaded replication is to make lock acquisition and release an externalization event (similar to a system call) that is handled by the *RomainMT* master. For this purpose we modified our L4 system's thread library so that all relevant lock functions issue a debug breakpoint instruction (`INT3` on x86) upon function entry. Thereby, any call to one of these functions will raise a debug exception that is reflected to the master.

The debug exception blocks the faulting thread and the master's exception handler waits until all threads of this

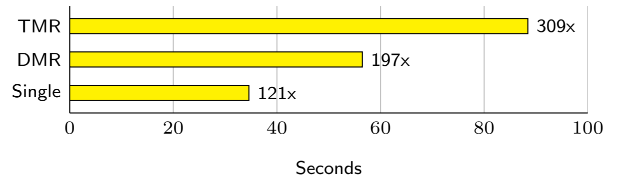


**Figure 3: Worst-Case Microbenchmark: Enforced determinism overhead for single-replica, double-modular (DMR) and triple-modular (TMR) execution. Native execution time: 0.286 seconds.**

group have reached their next event. If the threads executed deterministically and did not encounter a hardware fault, their states will match at this point. The master validates this. Upon success we know that all threads of this group are ready to acquire (or release) the same lock.

The master then determines the lock that is to be accessed by inspecting the parameters on the replica's stack. The lock ID is mapped to a master-internal lock and the master acquires this lock before resuming replica execution. The last step ensures that concurrent accesses to the same lock by different thread groups are correctly serialized inside the master. As this approach enforces determinism through an exception handler in the *RomainMT* master, we call it *enforced determinism*.

Our L4 system uses $\mu$Clibc[2] as its standard C library. We analyzed its `pthread` library and found that we have to instrument four functions to achieve determinism at the level of `pthread` mutexes: `__pthread_lock`, `__pthread_unlock`, `mutex_lock`, and `mutex_unlock`. These four functions suffice to run the test cases and benchmarks used in this paper. This includes support for `pthread` condition variables as they are implemented using lower-level mutex operations. We do *not* yet support timed versions of synchronization operations (e.g., `mutex_timedlock`).

We performed an initial microbenchmark to evaluate runtime overhead. Two threads increment a global counter variable 5 million times each. For each increment, a lock is acquired and released. The critical section is very short and the microbenchmark therefore allows us to estimate the worst-case overhead for instrumenting lock operations.

We executed the benchmark on a system equipped with 12 physical Intel Xeon X5650 CPU cores (on two sockets) running at 2.67 GHz. We pinned each thread to a dedicated physical CPU. No further applications were running on the system during our benchmarks. When executing the benchmark natively (without *RomainMT*), it completed in 0.286 seconds. We then ran the benchmark with a single replica in *RomainMT*. While this does not add fault tolerance, it allows us to determine the baseline overhead that is introduced by intercepting lock accesses. Thereafter, we measured the runtime overhead for double-modular (DMR) and triple-modular (TMR) execution.

Figure 3 shows the runtimes for this microbenchmark and compares them to native execution. We see that intercepting the 20,000,000 lock and unlock operations in our benchmark has a drastic impact on replication overhead. We investigated the origins of this overhead. First of all, our mechanism transforms 20 million lock function calls into CPU traps. The cost of delivering such an event to the master and returning to the replica amounts to about 2,200 CPU cycles per event. Throughout a benchmark run this cost sums up to about

[2] `http://uclibc.org`

8 seconds and remains constant regardless of the number of replicas we are running.

The second source of overhead we are seeing is the handling of lock events in the master process. Event handling consists of three phases: first, we wait for all replicas to reach their externalization event. Second, we validate replica states and handle the event. Third, the replicas are woken up again. We measured the time spent in these three phases for every benchmark run.

The single-replica case does not require synchronization between the replicas and spends 90% of its master execution time within the lock implementation, acquiring or waiting for a lock. In contrast, DMR execution spends 74% of its time in the replica synchronization phases. For TMR synchronization contributes to 83% of the total overhead.

*Optimizing Replica Synchronization* We investigated the high cost of replica synchronization and were able to attribute it to two factors: CPU placement and use of message-based synchronization primitives.

**CPU Placement** Our test machine's twelve CPU cores are organized in two sockets with six cores each. Replica synchronization requires messages to be sent between replicas. These inter-processor interrupts (IPIs) are an order of magnitude faster if sent within a single socket.

As explained before, *RomainMT* distributes replica threads across physical CPU cores. In the benchmark we sequentially assigned these threads to cores starting at CPU 0. Running two application threads in triple-modular redundancy therefore means we are using 6 CPUs for those threads, which should fit into one of the sockets. However, $\mu$Clibc's `pthread` library runs an additional thread per application, which is used for management purposes.

Therefore, we are in fact running three replicas of three threads and use 9 cores. Due to our CPU assignment, the replicas of the manager and first worker thread run on the six cores on socket 0, whereas the three replicas of the second worker thread run on socket 1. The manager thread performs no real work, but spends nearly 100% of its time waiting for messages from the worker threads. We manually *optimized CPU placement* to put the manager thread group on core 0 and distribute the replicas for the two worker threads across cores 0 to 5 respectively. Thereby all replicas run on a single socket in TMR mode.

**Blocking synchronization** *RomainMT* uses L4's messaging primitives to block replicas as long as the master is processing their events and to wake them up once this is done. These messages require additional IPIs to be sent between replicas. We modified synchronization so that the replicas send a trap event to the master and then poll a globally shared variable to wait until the master finished event processing and updated the replicas' states. Using this *Fast Synchronization* approach we avoid sending additional IPIs and messages.

However, the replicas now busily wait for the master instead of sleeping. This will increase their energy consumption and may render this optimization useless for battery-driven mobile devices. Hardware can aid with this problem by providing a possibility to block a CPU waiting for an event, such as Intel's `monitor/mwait` [8] and ARM's `wfe/sev` [2] instructions do.

**Performance impact** We evaluated optimized CPU placement and fast synchronization by repeating the previous benchmark and show the results in Figure 4. Single-replica execution is unaffected by our optimizations. DMR execution

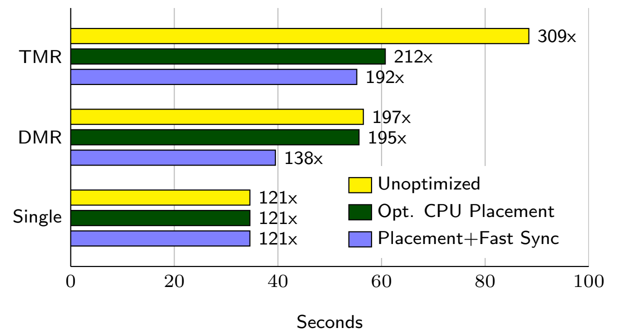


**Figure 4: Effect of optimizations on the worst-case microbenchmark using single-replica, double-modular (DMR) and triple-modular (TMR) execution. Native execution time: 0.286 seconds.**

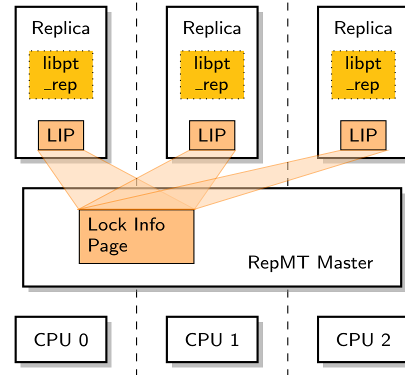


**Figure 5: Making applications replication-aware: a lock info page is shared among all replicas**

is not affected by optimizing CPU placement, because all replicas already run on a single socket. Fast synchronization decreases DMR overhead by 30%. Optimizing CPU placement reduces TMR overhead by 38%.

Despite these optimizations, our results indicate that every lock operation with enforced determinism will cost about 200 times the original overhead in TMR mode. This makes enforced determinism prohibitively expensive especially for applications that have high lock frequencies and short critical sections, such as our microbenchmark.

## 3.3 Cooperative determinism

The remaining runtime overhead for enforced determinism is mainly due to handling lock operations within the *RomainMT* master. We implemented an alternative that avoids the debug trap and expensive lock handling overhead by making applications replication-aware as depicted in Figure 5. We extended our customized `pthread` library to integrate more tightly with *RomainMT*. During application startup, the master process maps a shared *lock info page (LIP)* into all replica address spaces. Instead of turning lock operations into traps, the replication-aware library uses this LIP to ensure that the replicas agree on the ordering of lock acquisitions and releases at runtime. As determinism is achieved by cooperation among the replicas, we call this approach *cooperative determinism.*

*Implementing a Replication-Aware Thread Library* Our replication-aware thread library, called `pthread_rep`, replaces $\mu$Clibc's `mutex_lock()` and `mutex_unlock()` functions. The `pthread_rep` library makes two assumptions: first, all threads

```c
struct LIP {
  unsigned int num_replicas;
  struct {
    unsigned int spinlock;
    Address      owner;
    unsigned int acq_count;
    unsigned int epoch;
  } locks[MAX_LOCKS];
};
```

**Listing 2: Lock Info Page**

```c
struct LIP *lip; // global LIP address

int
pthread_mutex_lock(pthread_mutex_t *mtx)
{
  self()->epoch += 1;

  while (1) {
    spin_lock(lip, mtx);

    if (lip[mtx].owner == lock_free) {
      /* Lock is free */
      lip[mtx].owner     = self();
      lip[mtx].epoch     = self()->epoch;
      lip[mtx].acq_count = lip->num_replicas;
      spin_unlock(lip, mtx);
      break;
    }
    else if (lip[mtx].owner == self()) {
      /* Lock owned by my thread group? */
      if (lip[mtx].epoch == self()->epoch) {
        spin_unlock(lip, mtx);
        break;
      } else {
        spin_unlock(lip, mtx);
        cpu_yield();
        continue;
      }
    }
    else {
      /* owned by someone else */
      spin_unlock(lip, mtx);
      cpu_yield();
      continue;
    }
  }
  return 0;
}
```

**Listing 3: Replication-aware lock acquisition**

of a thread group have the same `pthread` ID, and second, identical locks in different replicas have identical lock IDs. $\mu$Clibc's `pthread` implementation uses the memory addresses of thread and lock objects as their IDs. As the *RomainMT* master ensures an identical memory layout for all replicas, these two assumptions hold.

The LIP contains information about the number of replicas and a shared data structure for managing locks. We show this data structure in Listing 2. For each lock, we store a spinlock that protects access to the respective lock information across replicas. The `owner` field stores the thread that currently possesses the lock, `acq_count` counts the number of replicas that are in possession of the lock. The `epoch` counter serves as a logical progress indicator for threads and is incremented whenever a thread calls a lock/unlock function. Our current implementation limits the LIP's size to support 2,048 locks. This suffices for the benchmarks used in this paper and can be increased or decreased as needed.

Listings 3 and 4 show the code for acquiring and releasing a lock. We use ConcurrencyKit's spinlock[3] to synchronize access to the LIP between replica threads. If the mutex'

[3] `http://concurrencykit.org/`

```c
int
pthread_mutex_unlock(pthread_mutex_t *mtx)
{
  spin_lock(lip, mtx);

  lip[mtx]->acq_count -= 1;
  /* Last one turns off the lights. */
  if (lip[mtx]->acq_count == 0) {
    lip[mtx].owner = lock_free;
  }

  spin_unlock(lip, mtx);
}
```

**Listing 4: Replication-aware lock release**

entry in the LIP indicates that it is unowned, we store the current thread's `pthread` ID in the owner field along with the thread's epoch and the global number of replicas. If the `owner` field indicates the lock is already acquired, we check if it is owned by the current thread group. In this case, any other thread of the thread group may simply continue as the lock has already been acquired. If we find that the lock is owned by a different thread group (in case of a contended lock), we yield the CPU and retry acquisition later.

A special case arises if the lock is owned by the current thread group, but the current thread's epoch counter is larger than the stored one. This happens if the current thread already released the lock and is now trying to re-acquire it. This case is handled as if the lock was owned by a different thread group in order to prevent one thread from overtaking its other replicas.

Our current implementation makes `pthread` mutexes use busy waiting whereas usual `pthread` implementations only spin for some time and then block until they are woken up by the next release operation. The `cpu_yield` function shown in the listings can be modified to not yield the CPU, but block by raising an externalization event. This event will then be handled by the *RomainMT* master. This solution restores the original wait behavior provided by the `pthread` library. In the benchmarks we conducted for this paper, the native `pthread` functions rarely went to sleep because locks were either lowly contended or critical sections were short enough for the lock to only spin.

*Externalization Events and the LIP* When accessing the LIP, the replication-aware lock functions read and modify shared data instead of private copies in the local replica. At this point the replica's register states may deviate, because shared data is read during the lock acquisition phase. This is a problem if the thread group raises an externalization event (e.g., a page fault) while executing this code. In such case the *RomainMT* master will deem this deviation a hardware fault. We circumvent this problem by pinning the LIP and the thread library's code into each address space during startup. Thereby, lock code will never raise a page fault.

We had to deal with another issue that was caused by compilers' calling conventions. 32-bit x86 considers registers EAX, ECX, and EDX *caller-saved*, which means that a function can overwrite their content at will. If these registers are not used by the caller, GCC omits code to save/restore their contents as a performance optimization. In the case of replication this leads to situations where these unused registers are used inside the lock functions to store volatile LIP data. Execution then returns to the caller with differing register content. If the caller does not restore the original register content and does not use the registers until the thread group raises its next valid event (e.g., a system call),

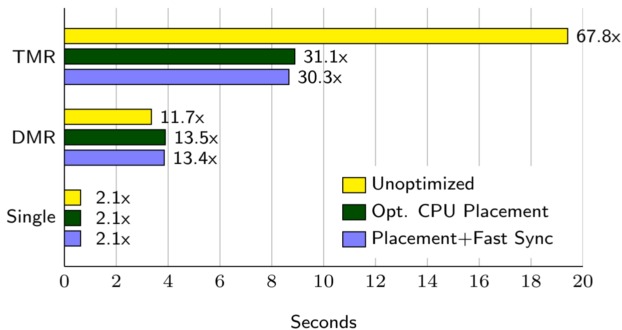


**Figure 6: Cooperative Determinism overhead for single-replica, double-modular (DMR), and triple-modular (TMR) execution. Native execution time: 0.286 seconds.**

the master will detect differing CPU states even though no hardware fault happened. We avoid this problem by manually setting the caller-saved registers to constant 0 before returning from the replica-aware lock and unlock functions (not shown in Listings 3 and 4).

To estimate how cooperative determinism performs in comparison to our enforced determinism implementation, we repeated the microbenchmark described in Section 3.2 and show the results in Figure 6. Again, we show results for unoptimized replica placement and the two performance optimizations discussed in the previous section. Our results show that cooperative determinism significantly reduces replication overhead. Using our optimizations, a lock operation in TMR mode will still be 30 times slower than native, but this is 6 times faster than the best result we were able to achieve using enforced determinism.

## 3.4 Fault Recovery

When an erroneous replica is detected during fault handling, *RomainMT* initiates a recovery routine. For DMR execution, this terminates the application and relies on an external recovery mechanism, such as restarting or rolling back to a previous checkpoint. If three or more replicas are present, *RomainMT* performs forward recovery using majority voting.

First, all thread groups are halted to prevent correct thread groups from obtaining wrong data. Then, the erroneous replica is identified by comparing this thread group's thread states as in original *Romain*. We can only compare the erroneous thread group here, because all other thread groups were stopped at arbitrary points within their execution and are likely to differ at this point. This is no problem if we can assume a single-fault model. In this case we already found the erroneous replica and while other threads in the defective replica may suffer from errors as well, these errors will be corrected by majority voting.

After identifying the mismatching replica, *RomainMT* selects a correct replica for recovery. All thread states of the erroneous replica are set to the state of the corresponding correct thread in their thread group. All memory regions of the erroneous replica are replaced with data from the correct replica. At this point the error is corrected and execution resumes.

The recovery implementation above succeeds if the replica deviation was caused by a transient hardware error, because this error is now overwritten. If the underlying hardware suffers from a permanent error, this error will lead to another deviation in the near future. To handle permanent errors, *RomainMT* would need to collect statistics about where an error occurred in order to determine that one replica is raising errors originating from the same resource over and over. Based on these statistics we could then migrate the affected replica threads to another CPU. We did not implement such a mechanism yet.

# 4 Evaluation

To evaluate *RomainMT*'s impact on real-life applications, we evaluate the replication-induced overheads for the Modified SPLASH2 [44] benchmark suite[4] in Sections 4.1 and 4.2. These benchmarks were also used by Kendo's authors [33]. As before, all tests were carried out on a machine with 12 physical Xeon X5650 CPU cores on two sockets, running at 2.67 GHz. Hyperthreading, frequency scaling, and TurboBoost were turned off for our experiments. The software stack consists of an unmodified L4 kernel, the required user space components, and our implementation of *RomainMT*.

Previous research found that some of the the SPLASH2 benchmarks contain data races [31] and thereby violate our assumption about race-freedom. We analyzed these races using Valgrind's ThreadSanitizer race detector [39] and removed those races that prevented deterministic execution.[5] To facilitate reproduction of our results, we make our patches available to the community at `http://tudos.org/~doebel/emsoft14`.

## 4.1 Memory Replication Overhead

In our previous work [12] we identified memory management as a main contributor to replication overhead, because running multiple replicas requires memory regions to be replicated leading to allocation and copying overhead. The SPLASH2 benchmarks consist of an initialization phase (allocating and initializing potentially large chunks of memory) and a compute phase. While only the latter phase is relevant to evaluate the overhead caused by our locking modifications, we saw large replication overheads for the initialization phases and attributed them to memory management.

As an example, Figure 7 shows the overhead for running the FMM benchmark from the SPLASH2 suite in single, double and triple modular redundancy using cooperative determinism. We see that the compute phase overhead is only 2% for TMR execution, whereas the total runtime overhead for TMR is 107%!

We investigated this issue and found that *RomainMT* manages memory by mapping a 4 KiB page to each replica upon a page fault. The SPLASH2 benchmarks use hundreds of megabytes of memory (330 MiB in the FMM example), which means that even a single replica will raise several thousand page faults, which need to be handled by the master. The resulting overhead is amplified for DMR and TMR because in this case the master needs to create copies of all memory regions during page fault handling.

We applied two modifications to improve memory management: first, we adapted the master process to allocate memory in 4-MiB superpages wherever suitable. Second, we allow the master's page fault handling code to handle a page fault by immediately mapping more than a single 4 KiB page (up to one 4-MiB page). The resulting decrease in overhead can be seen in the *Total.opt* line of Figure 7. We see that

[4] `http://www.capsl.udel.edu/splash/`

[5] We modified the Barnes, FMM, Ocean, and Radiosity benchmarks.

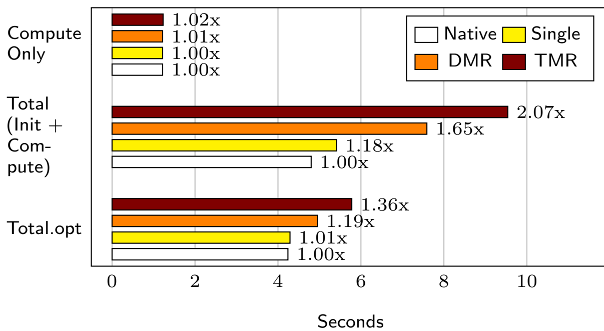


**Figure 7: Overhead for SPLASH2's FMM benchmark running in *RomainMT*, split up for compute phase overhead (Compute), full runtime overhead (Total), and full run overhead after applying memory management optimizations (Total.opt).**

the combined optimizations lead to a significant decrease in memory-related replication overhead.

While it may appear insignificant to optimize initialization overheads, we believe that this optimization addresses an inherent replication problem. Any application that dynamically acquires and releases memory (and hence raises many page faults) will benefit from these improvements. In the remainder of this paper we show results using the described memory management, as well as the previously introduced CPU assignment optimizations.

## 4.2 Application-Level Evaluation

Figures 8 and 9 show the normalized runtime overheads for the SPLASH2 benchmarks using cooperative determinism and running with two and four application threads, respectively. As observed in Section 4.1, initialization times and their replication-related memory overhead is significant. Therefore, we show the initialization time overheads along with the pure compute time overheads for those SPLASH2 benchmarks that measure both times separately.

For two application threads, TMR execution causes a geometric mean overhead of 24%. When running four application threads, TMR overhead increases to 65%. These overheads are acceptable compared to alternatives such as executing the application multiple times and comparing their outputs. Replication provides the advantage of repairing errors at runtime while multiple re-executions require instances to finish before their results can be compared. Hence, replication yields lower error detection latencies.

Replication overhead results from two sources: system calls and synchronization. The Water and Ocean benchmarks perform a significant number of system calls, which restricts independent execution of replicas and hence explains their overheads.

All other overheads are dominated by the programs' lock densities. Any lock operation requires synchronization among otherwise concurrently executing replica threads. Hence, the more lock operations an application performs, the more constrained it becomes in terms of parallelism. To demonstrate this, we order the benchmarks by their lock density from high (Radiosity, 6 million lock operations per second) to low (Radix, 9 lock operations per second). We see that especially with four application threads, those benchmarks with high lock densities have higher overheads as well.

We conclude that replication becomes infeasible for applications with high lock density and large thread counts. This is not a direct problem of replicated execution. Instead, locking is known to be a scalability bottleneck [24]. This effect is merely amplified by running multiple replica threads in our experiments.

We also measured the overheads for enforced determinism, but do not show them for space reasons. Enforced determinism induces significantly higher runtime overheads (geom. mean DMR: 92%, TMR: 113%) as we expected after our initial microbenchmarks in Section 3.

In our design we sacrificed the requirement to produce the same deterministic event order in every application run and focused on ensuring determinism within a single replication run. As a result, our overhead for single-replica execution is close to zero as opposed to Kendo [33], but does not provide completely deterministic thread scheduling in this case.

# 5 Limitations

*RomainMT* instruments `pthread` lock operations to achieve deterministic multithreaded replication and assumes the replicated application to be data-race free. This in turn poses a limitation on our work, because it prevents *RomainMT* from replicating applications that use ad-hoc synchronization (spinlocks) or lock-free data structures [18]. This problem can be solved by applying strongly deterministic multithreading approaches, such as DThreads [27], but will lead to higher resource consumption and runtime overheads as discussed before.

In line with the previous work on *Romain*, the *RomainMT* master and the underlying operating system kernel (the Reliable Computing Base) remain unprotected against hardware errors. Our work does not address this issue. We believe that fault-tolerant compiler techniques [36] or heterogeneous hardware with differing resilience properties [10,25] may help protecting the RCB.

The replication-aware `pthread` library described in Section 3.3 allows for minimizing runtime overheads. However, from a fault tolerance point of view it also adds a single point of failure. The lock info page is mapped into all replicas and can be overwritten due to a hardware fault before this fault is detected. Such a fault will modify application behavior and may result in undetected or unrecoverable application errors. We believe that this can be mitigated by applying compiler-level redundancy [36] to `libpthread_rep`, but the resulting performance implications remain to be investigated. For first ideas on this please see our estimation of the effects of compiler-based RCB hardening [11].

# 6 Conclusion

In this paper we presented *RomainMT*, an operating system service that allows transparent replication of multithreaded binary-only applications in order to tolerate transient hardware faults. While many existing SIFT methods only support single-threaded applications, *RomainMT* solves the non-determinism problems that make multithreaded replication difficult by reusing ideas from the field of deterministic multithreading.

We presented two mechanisms to achieve lock-based determinism by instrumenting synchronization functions in a `pthread` library: enforced determinism transforms these operations into traps handled by the *RomainMT* master. Trap-based determinism incurs a high runtime overhead. Cooperative determinism avoids expensive trapping by sharing a lock info page among all replicas. Replicas use the LIP to agree on lock ordering.

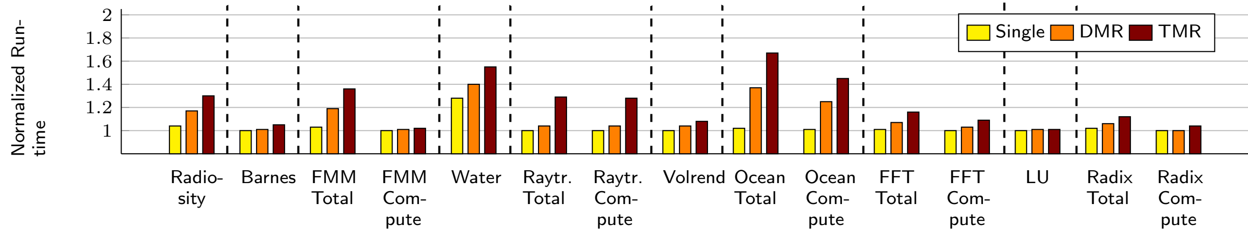


**Figure 8: SPLASH2 replication overheads for cooperative determinism using two application threads. Benchmarks ordered by decreasing lock density. Geometric Means: DMR = 13%, TMR = 24%. (Means computed over full benchmark runtimes.)**

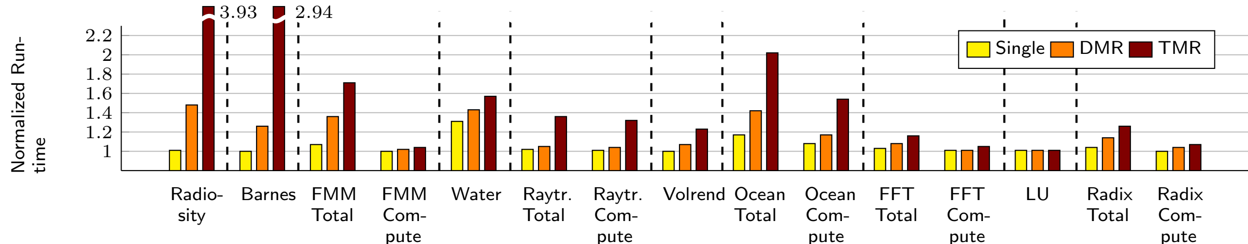


**Figure 9: SPLASH2 replication overheads for cooperative determinism using four application threads. Benchmarks ordered by decreasing lock density. Geometric Means: DMR = 22%, TMR = 65%. (Means computed over full benchmark runtimes.**

We evaluated *RomainMT* using the SPLASH2 suite of benchmarks and showed that we achieve geometric mean overheads 24% for TMR with two threads and 65% for TMR with four threads. This overhead is lower than the overhead for previous compiler-level methods that only protect single-threaded execution [16]. Compared to Mushtaq's multithreading work [30], *RomainMT* requires a slightly higher overhead, but provides fault tolerance even in the presence of memory faults.

## Acknowledgment

We would like to thank the anonymous reviewers as well as Thomas Knauth, Michael Roitzsch, Stephan Diestelhorst, and Markus Partheymüller for their feedback that helped to improve this paper.

This work was supported by the German Research Foundation (DFG) as part of the priority program "Dependable Embedded Systems" (SPP 1500 – `http://spp1500.itec.kit.edu`) and by the European Social Fund and the Free State of Saxony within the project Secure Remote Execution (SREX, Nr. 100111037).

## 7 References

[1] 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.

[2] ARM. ARM11 MPCore Processor Technical Reference Manual. Technical Documentation at `http://infocenter.arm.com`, 2008.

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

[4] Aviram, A., Weng, S.-C., Hu, S., and Ford, B. Efficient system-enforced deterministic parallelism. In *Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–16.

[5] Bergan, T., Hunt, N., Ceze, L., and Gribble, S. D. Deterministic Process Groups in dOS. In *Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–16.

[6] Bernick, D., Bruckert, B., Vigna, P., Garcia, D., Jardine, R., Klecka, J., and Smullen, J. Nonstop: Advanced architecture. In *International Conference on Dependable Systems and Networks* (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] Corp., I. Intel64 and IA-32 Architectures Software Developer's Manual. Technical Documentation at `http://www.intel.com`, 2013.

[9] Cui, H., Wu, J., Gallagher, J., Guo, H., and Yang, J. Efficient deterministic multithreading through schedule relaxation. In *SOSP* (2011), T. Wobber and P. Druschel, Eds., ACM, pp. 337–351.

[10] Döbel, B., and Härtig, H. Who watches the watchmen? – protecting operating system reliability mechanisms. In *International Workshop on Hot Topics in System Dependability (HotDep)* (2012).

[11] Döbel, B., and Härtig, H. Where have all the cycles gone? – investigating runtime overheads of os-assisted replication. In *Workshop on Software-Based Methods for Robust Embedded Systems* (2013), SOBRES'13.

[12] Döbel, B., Härtig, H., and Engel, M. Operating system support for redundant multithreading. In *12th International Conference on Embedded Software (EMSOFT)* (2012).

[13] Elliott, J., Kharbas, K., Fiala, D., Mueller, F., Ferreira, K., and Engelmann, C. Combining Partial Redundancy and Checkpointing for HPC. In *Conference on Distributed Computing Systems* (Macau, SAR, China, June 18-21, 2012), ICDCS '12, IEEE, pp. 615–626. Acceptance rate 13% (71/515).

[14] Ernst, D., Kim, N. S., Das, S., Pant, S., Rao, R., Pham, T., Ziesler, C., Blaauw, D., Austin, T., Flautner, K., and Mudge, T. Razor: a low-power

pipeline based on circuit-level timing speculation. In *Annual International Symposium on Microarchitecture* (dec. 2003), pp. 7–18.

[15] Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D. Dark Silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture* (New York, NY, USA, 2011), ISCA '11, ACM, pp. 365–376.

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

[17] Fiala, D., Mueller, F., Engelmann, C., Riesen, R., Ferreira, K., and Brightwell, R. Detection and correction of silent data corruption for large-scale high-performance computing. In *International Conference on High Performance Computing, Networking, Storage and Analysis* (Los Alamitos, CA, USA, 2012), SC '12, IEEE Computer Society Press, pp. 78:1–78:12.

[18] Herlihy, M. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst. 15*, 5 (Nov. 1993), 745–770.

[19] Hwang, A. A., Stefanovici, I. A., and Schroeder, B. Cosmic rays don't strike twice: understanding the nature of dram errors and the implications for system design. *SIGARCH Comput. Archit. News 40*, 1 (Mar. 2012), 111–122.

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

[21] Intel. Thread building blocks (TBB). `http://www.threadbuildingblocks.org`, 2013.

[22] Kapitza, R., Schunter, M., Cachin, C., Stengel, K., and Distler, T. Storyboard: optimistic deterministic multithreading. In *International Conference on Hot Topics in System Dependability* (Berkeley, CA, USA, 2010), HotDep'10, USENIX Association, pp. 1–8.

[23] Kaptritsos, M., Wang, Y., Quema, V., Clement, A., Alvisi, L., and Dahlin, M. Eve: Execute-verify replication for multi-core servers. In *OSDI 2012* (Oct 2012).

[24] Kleen, A. Linux multi-core scalability. Tech. rep., 2009.

[25] Leem, L., Cho, H., Bau, J., Jacobson, Q., and Mitra, S. ERSA: Error Resilient System Architecture for Probabilistic Applications. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010* (March 2010), pp. 1560–1565.

[26] Lewis, B., and Berg, D. J. *Multithreaded programming with Pthreads*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

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

[28] Merrifield, T., and Eriksson, J. Conversion: Multi-version concurrency control for main memory segments. In *Proc. of EuroSys 2013* (2013).

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

[30] Mushtaq, H., Al-Ars, Z., and Bertels, K. Efficient software based fault tolerance approach on multicore platforms. In *Proc. Design, Automation & Test in Europe Conference* (Grenoble, France, March 2013).

[31] Nistor, A., Marinov, D., and Torrellas, J. Light64: lightweight hardware support for data race detection during systematic testing of parallel programs. In *International Symposium on Microarchitecture* (New York, NY, USA, 2009), MICRO 42, ACM, pp. 541–552.

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

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

[34] Pinheiro, E., Weber, W.-D., and Barroso, L. A. Failure trends in a large disk drive population. In *5th USENIX Conference on File and Storage Technologies (FAST 2007)* (2007), pp. 17–29.

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

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

[37] 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.

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

[39] Serebryany, K., and Iskhodzhanov, T. Threadsanitizer: data race detection in practice. In *Workshop on Binary Instrumentation and Applications* (New York, NY, USA, 2009), WBIA '09, ACM, pp. 62–71.

[40] 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 *International Conference on Dependable Systems and Networks* (Washington, DC, USA, 2007), DSN '07, IEEE Computer Society, pp. 297–306.

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

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

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

[44] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. The SPLASH-2 programs: characterization and methodological considerations. In *International Symposium on Computer Architecture* (New York, NY, USA, 1995), ISCA '95, ACM, pp. 24–36.

[45] Yoshida, J. Toyota Case: The Single Bit Flip That Killed. `http://www.eetimes.com/document.asp?doc_id=1319903`, Oct. 2013.

[46] Zhang, Y., Lee, J. W., Johnson, N. P., and August, D. I. Daft: decoupled acyclic fault tolerance. In *International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2010), PACT '10, ACM, pp. 87–98.