**Diplomarbeit**

# Reducing Resource Consumption of Replication using Dynamic Replicas

Robert Muschner

July 15, 2013

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

| | |
|---|---|
| Betreuender Hochschullehrer: | Prof. Dr. rer. nat. Hermann Härtig |
| Betreuende Mitarbeiter: | Dipl.-Inf. Björn Döbel |
| | Dipl.-Inf. Michael Roitzsch |

## Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 15. Juli 2013

Robert Muschner

# Contents

# List of Figures

# 1 Introduction

Operating systems (OS) manage computer hardware resources and provide services to the user's applications. For reliable execution the OS depends on error free hardware. The semiconductor industry progressively produces smaller and faster processors. The drawback of this development is that these chips are more prone to transient hardware errors. Hardware protection mechanisms are implemented for special use cases but these mechanisms are not available in commercial-of-the-shelf (COTS) systems.

To handle transient hardware errors in COTS systems a software solution is needed. *ASTEROID* [DHAE12] provides this property. The project is based on a microkernel, system management functionality and a replication framework. The latter one is implemented as an operating system service, called *Romain* [DHE12]. Romain redundantly executes an application for transient hardware error detection and recovery. Romain addresses bit flips in the logical units of the central processing unit (CPU) caused by alpha or neutron particle strikes.

Executing one application $N$-times in parallel consumes $N$-times the resources of one application. Applications with high redundancy requirement for resilient execution need a lot of system resources to work properly. Therefore, a high-end system is needed to overcome the resource consumption which limits the applicability.

In this thesis I present dynamic replicas, a mechanism to adjust the number of parallel executions of the application during runtime. One execution of the application is called a replica and it becomes dynamic when the replica can be turned on and off as needed. I added dynamic replicas to Romain. The use case is to provide more replicas when critical code sections are executed and fewer replicas during non-critical parts to save system resources. Additionally, I describe copy on write, a memory saving mechanism, and its implementation in Romain. Copy on write can reduce the number of memory operations by just copying a memory region if a process wants to modify it. Finally, I evaluate my implementation and show that system resources can be saved by using dynamic replicas or copy on write.

# 2 Fundamentals

We are currently living in a time where Moore's Law [Moo65] is still accurate. Computer systems' circuits are getting smaller.

Smaller transistors lead to chips with more functionality without increasing the chip size but induce drawbacks as well. For example, the more dense the transistors are packed the more heat is generated which limits the chip size. This limitation is termed *power wall*. A fast CPU needs fast memory and a fast connection in between, otherwise the CPU cannot be used to capacity. Hence, the number of instructions per second depends on the memory and therefore the limitation is termed *memory wall*.

Beside these two effects, radiation-induced transient hardware faults caused by alpha particles and neutrons are becoming more likely with decreasing transistor size. Radioactive impurities used in the industrial chip packaging process can produce alpha particles. Neutrons are part of cosmic rays and occur from atom fission. Alpha and neutron strikes interact with silicon crystals in different ways but lead to charge accumulation in memory cells. If a certain threshold is reached, memory cells will flip their charge. This threshold is termed *critical charge* [Muk08, Chapter 1.7.3]. With decreasing transistor size the critical charge decreases as well and therefore the transistors are more vulnerable to alpha and neutron strikes. These bit flips can either result in *Silent Data Corruption* (SDC) or *Detected Unrecoverable Errors* (DUE) [Muk08, Chapter 1.1.2]. These faults are termed *soft errors* because they do not permanently affect the device [Muk08, Chapter 1]. According to the terminology before, this can be termed *soft error wall*. If a soft error is caused by alpha or neutron particle strikes then the fault is termed *Single-Event Upset (SEU)* [Muk08]. All in all, the "wall"-problems resulting from shrinking circuits can lead to error prone hardware.

To face these impacts, different solutions have been researched. The "memory wall" led to faster memory technology, for instance *Phase-change RAM* (PRAM) [BLL$^+$11]. Likewise, higher level architectural solutions such as prefetching and multithreading reduce the impact of longer latency memory operations. The "power wall" led to research about energy efficient computing or more general *Green IT* [Mur08].

To face the "soft error wall", error detection and recovery can built into hardware, software or both of them. Hardware techniques are fast on execution but in comparison to software techniques their implementation is expensive. Imagine a system that provides error detection for memory but no such mechanism for CPU registers. To extend the error detection in hardware new physical components are required. To change or improve a software solution, no modifications to the physical components of a system are needed. Therefore, software is more cost and time efficient due to producing and applying the same functionality. This thesis focuses on error detection and recovery in software.

*Replication* is a software approach to detect errors by comparing the execution states of a redundantly running program. Each replica begins in the same state, executes the same instructions in the same order and produces the same output. In his paper [Sch90], Fred B. Schneider assumes that at most one processor can be affected by a single fault but multiple faults can occur at the same time. Therefore, at least $2t + 1$ replicas are needed to determine the flawed replicas when $t$ faults occurred [Sch90]. The majority of the replicas' states are correct hence the flawed replicas can be identified and recovered.

An implementation of the replication approach is *Romain* [DHE12] an error detection and recovery framework that focuses on SEUs in logical units of the CPU. Romain runs redundant copies of the same program on different physical CPUs to detect faults. At certain points during the execution (e.g. system calls) Romain compares the running replicas' state. An error is recovered by replacing the flawed replica by a proper one.

## 2.1 Romain

The resilient operating system (OS) architecture *ASTEROID* [DHAE12] uses a software stack based on the L4 microkernel Fiasco.OC [TD13a], the L4 Runtime Environment (L4Re) [TD13b] and Romain. The combination of Fiasco.OC and L4Re forms the operating system we are using - TUD:OS. Together with Romain they form the *Reliable Computing Base* (RCB) [DHE12, 4.1] as depicted in Figure 2.1.

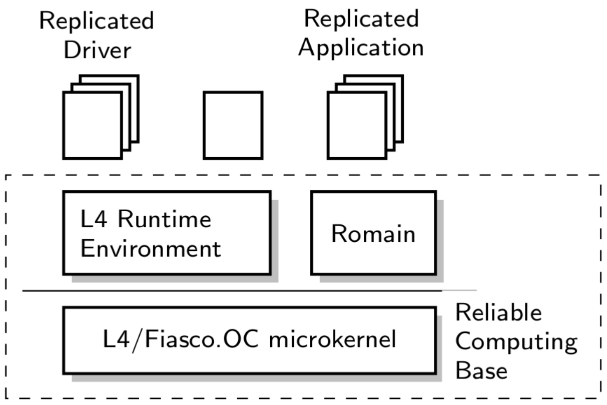

Figure 2.1: ASTEROID overview [DHAE12]

In this section, I shortly describe the RCB and Romain's internal architecture and functionality because this knowledge is crucial for the implementation details described from Chapter 3 on.

### 2.1.1 Reliable Computing Base

Romain's replicas do not rely on the registers of the CPU but need to rely on Romain's software stack. The term Reliable Computing Base is derived from the term *Trusted Computing Base* (TCB) [Tan09, Chapter 9.3.5] and describes the minimal reliable soft- and hardware stack. System components that need to be trusted to achieve a certain security goal are part of the TCB. For example, a malicious hard disk driver can overwrite important data which can lead to corrupted backups. Reliable computing base refers to all system components that need to work reliably with respect to hardware errors [DH12]. In Romain's case we do not trust the CPU but expect error-correcting code (ECC) memory. On the software side, Romain relies on Fiasco.OC, L4Re and the correct execution of itself.

One possibility to protect the software part of the RCB is to physically separate its execution. ASTERIOD's RCB can run on a resilient core and the redundant application on non-resilient cores. A resilient core contains error correcting mechanisms in contrast to non-resilient cores. Due to reduced functionality on the chips, the production costs of non-resilient cores are cheaper compared to resilient cores [DH12].

### 2.1.2 An Overview

Romain terms each redundantly executing copy of the user's application a *replica*. Each replica consists of a vCPU and a separate address space. A *vCPU* (virtual CPU) is an abstraction of a real processor. It was designed to provide OS rehosting, a virtualization technique, on Fiasco.OC [LWP10].

The management of the replicas is done by Romain's *master*. Figure 2.2 shows the core part of Romain and the redundant application on top - the master and the replicas. Each replica runs on a different physical CPU. The code of the application runs inside a single thread, the vCPU, and in a separate address space to isolate the replicas of each other.

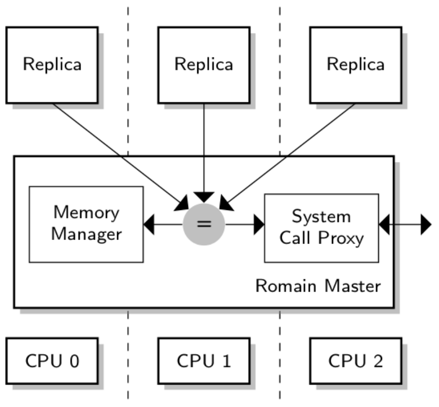

Figure 2.2: Romain's architecture [DH12]

Romain uses Fiasco's vCPU feature to handle replicas and intercept their actions. The vCPU's owner is in control of the vCPU's registers and trap handling. During runtime the user's application executes instructions that result in kernel traps such as page faults. The vCPU then invokes a pre-defined handler - Romain's master. It handles the trap and validates the replica states. After making sure that the replica states match, the master performs trap handling on behalf of the replicas. In most cases this simply means redirecting/proxying a system call. However, for some operations, such as memory management, Romain interrupts and handles them directly in order to maintain control over replicas.

Therefore, Romain controls the replica states every time the virtual CPUs trap.

### 2.1.3 The Master

As you can see from Figure 2.2 Romain's master can be divided in three parts - replica memory management, redundancy and trap handling. In general, the master performs three tasks: *error detection*, *recovery* and *trap handling.*

Romain uses statically configured options to control its behavior. These options include, which trap handlers and how many replicas are used. Depending on the options, the master starts replicas by creating a vCPU and a task for each replica and loading/preparing the application for execution.

The master assigns vCPUs to different physical CPUs to prevent mutual interference through soft errors. A redundancy manager keeps track of each replica during trap handling. The manager performs error detection by checking the replica states and synchronizes the replicas. Meaning, the replicas wait for the master to finish the trap handling, to resume in the same state together.

When dealing with an SEU fault model, we assume a single error to be present at a time. Following the $2t+1$ rule, we hence need to run three replicas to detect and recover an SEU. Therefore, *triple-modular redundancy* (TMR) is Romain's most common mode of execution.

#### 2.1.3.1 Memory Management

Memory is managed as shown in Figure 2.3. In this example Romain replicates an application three times. The big rectangles depict the tasks for each replica A, B and C and the master's task. The smaller rectangles inside symbolize memory regions. The master is in control of all the replicas' memory to handle page faults. Therefore, the replicas' tasks only contain mappings to these regions which is depicted by the grey connection lines.

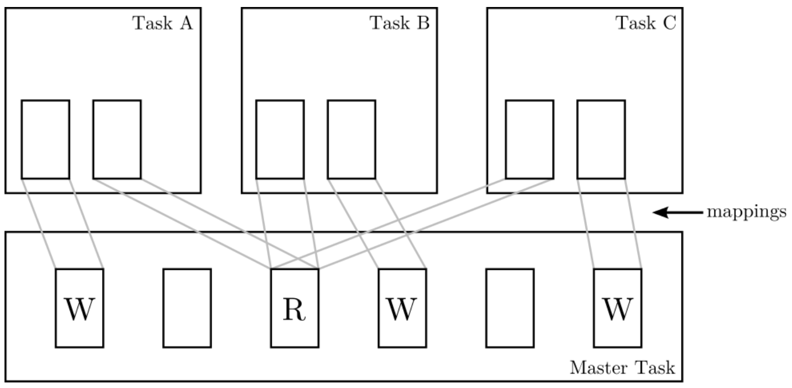

Figure 2.3: Replicas' memory management

If a write page fault occurs, the master tries to map the region into all replicas. If the page is not present in its address space as well the master's memory manager is invoked. It looks up the memory region that causes the page fault, requests a page mapping to the master's task and the master adjusts its page table. Afterwards, the master creates a copy for each replica and maps the page into the replicas' tasks. Each replica needs its own writable memory regions because during a write operation an SEU in the logical units of the CPU cannot affect all replicas at the same time. In contrast, read-only pages are shared across all replicas and therefore a single copy of read-only pages is available in the master's task.

If the application wants to allocate one megabyte of memory, the master allocates one megabyte for each replica in its task and maps the new regions into the replicas' address spaces. The other way around, if the application wants to deallocate a memory region, the master unmaps the region of each replica and deallocates the region from its task.

#### 2.1.3.2 Trap Handling

The runtime loop is waiting for vCPU traps until the application terminates. Specifically, each replica executes the application in its own vCPU. When the vCPU cannot process an instruction on its own, it generates a trap and therefore invokes the trap handling as depicted in Figure 2.4. For instance, the application wants to access a memory page which is not mapped in its address space. This leads to a page fault which causes the vCPU to trap. Romain then handles this page fault by making the requested memory pages available to each replica. To do so, the master maintains and enforces a view of each replica's address space by intercepting and handling all memory management calls. Other reasons are system calls where the kernel gets invoked as well. For example, printing text on the screen or reading data from a network device generates a context switch into the kernel. The trap handler for each replica is Romain.

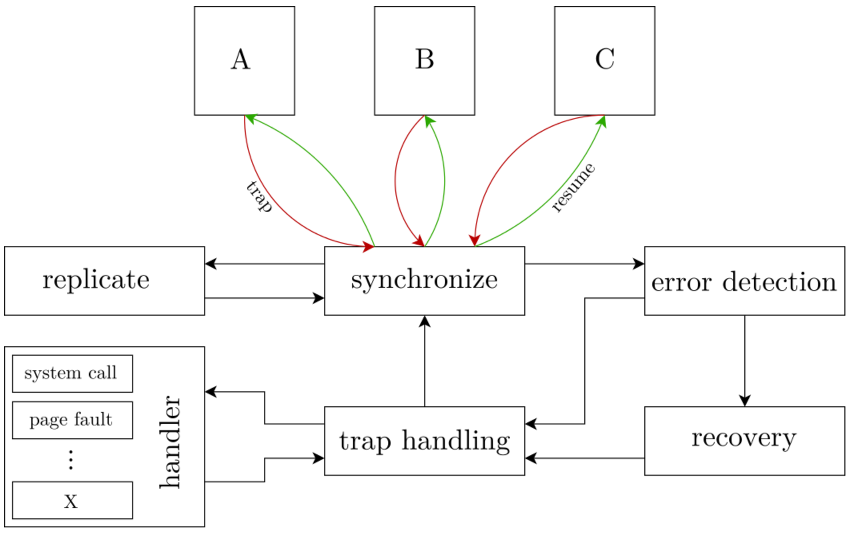

Figure 2.4: Trap handling

After all replicas (A, B and C) trap, Romain checks the replicas' states for possible errors. A *replica state* is a checksum of the vCPU's x86/32 registers. The states have to be equal across all replicas. In that case, trap handling proceeds, otherwise Romain needs to detect the flawed replica and start recovery. This is done by overwriting the replica's state with the state of a valid replica. However, Romain's master handles traps in different ways depending on the trap's type.

In the next chapter I present dynamic replicas and therefore it is necessary to know the trap handling internals.

A single replica handles the trap. Have a look at Figure 2.4 and imagine the replicas execute a "Hello, World!"-application. During the execution each replica wants to write to the screen. The replicas execute a system call and its' vCPU generate a trap. In the synchronization phase all replicas need to wait for each other to be sure that each replica is in a fixed state and waiting for the trap handling results. Thereafter, one replica checks the replicas' state while the others are waiting in the synchronization phase. Depending on the error detection phase, the one replica continues the trap handling or invokes the recovery mechanism described before. In the first case, it calls the list of trap handlers and handles the trap for itself. If memory allocation/deallocation is necessary to handle the trap it can take an arbitrary amount of time depending on the number of replicas. In this thesis, I term this part of the trap handling time *memory management delay*. Next, all other replicas apply the trap handling results to themselves in the replicate phase. Therefore, the replicas adapt their states to the trap handling one. The synchronization phase is passed twice during the trap handling. At first, all replicas wait for the handling replica and thereafter the handling replica waits for the others.

The list of trap handlers is implemented with the observer design pattern [GHJV94, Page 293]. Consequently, adding and removing new trap handlers is straightforward and does not require a re-implementation of the master. In general a trap handler implements a startup function and a notify function. For example, the breakpoint handler's startup function sets preconfigured breakpoints in each replica and the notify function checks if the breakpoints were hit. The startup function is called once and the notify function on every trap. If the trap handler can process the trap then it returns *success* or *ignored* otherwise. The former terminates the trap handling and the latter enables the opportunity to implement additional features. For example, the trap handler can count the number of traps or gather information about the instruction that causes the trap but most of the time it simply ignores the trap.

An essential trap handler is the *page fault handler*. It obtains the region and the region handler related to the faulting address, calculates the offset in this region and tries to map as large memory regions as possible for each replica.

Another important trap handler is the *system call handler*. Most of the time it forwards calls by invoking a real system call using the replica's vCPU registers. All memory management calls are handled by the master itself to ensure that these calls are applied to all replica address spaces.

In summary, for one trap the master compares the replica states for error detection, recovers a flawed replica by applying a correct state and then handles the trap for all replicas.

## 2.2 What's Bad about Multiple Replicas

For $N$ running replicas, $N$-times the amount of resources of one replica is required by Romain.

In a worst-case scenario, memory space in the order of gigabytes is used writable. For instance, if the application wants to allocate one gigabytes of memory the operation results in allocating $N$-times one gigabyte of memory. If the system doesn't run out of memory it will massively lack in performance due to the *memory management delay* as described in the section before. This delay costs execution time and therefore energy.

To save system resources (memory, execution time and energy) we want to run as few replicas as possible but as many replicas as needed to resiliently execute the application.

## 2.3 Program Vulnerability

In this section I describe an analysis tool to approximate program vulnerability [SK09] and its impact for this thesis.

The idea is based on the AVF (architectural vulnerability factor) [MWE$^+$03]. The PVF analysis tool [DSE13] is able to analyze x86 binaries by their instruction trace. It iterates over the instructions and evaluates them by a given fault model. The authors laid the focus on possible bit flips in CPU registers. Hence, the analysis generates a sequence of states for each register. For example, a state changes if an instruction writes this register. An instruction is classified by its state as vulnerable or invulnerable. Finally, "the register PVF for a given trace is computed as the ratio of vulnerable instructions compared to trace's instruction count" [DSE13].

The PVF authors' experiments with the analysis tool finished in minutes but lacked in precision compared to fault injection experiments [HANR12]. However, such an analysis can hence identify program phases with different vulnerabilities and aid Romain in adjusting the number of replicas accordingly. In my thesis, I aim to determine what needs to be done to let Romain use a dynamic number of replicas, and how expensive this adaptation is. The induced overhead for this operation may be a limitation with respect to the frequency of the adaptation.

# 3 Dynamic Replicas

For $N$ running replicas, $N$ times the resources of one replica are needed during runtime but the amount of system's resources is limited. We can reduce the number of replicas but this results in reducing the application's resilience as well. For instance, if we execute a single replica instead of three then the application runs without redundancy. Executing two replicas results in detecting the fault but lacks in recovery because no majority decision is possible to determine the correct replica. We want more replicas to detect the flawed replica and to guarantee resilient execution of the user's application. With the help of the PVF analysis we can determine critical parts of the program. We want to use more replicas when entering a critical code part and fewer replicas when leaving. Therefore, a mechanism is needed which turns replicas on and off on demand.

In this chapter I present *dynamic replicas*, their design and the implementation in Romain.

## 3.1 Who Is in Control of the Switch?

Depending on the application scenario the control over turning replicas on and off is implemented in the application or in Romain.

An application built for execution with Romain could implement the control itself - application-aware. The knowledge about critical code parts of the application is considered during development or the application is instrumented afterwards. In this case the application notifies Romain to increase or decrease the amount of replicas.

Another use case is to apply PVF analysis. Thereby, an application binary is statically analyzed and modified. Special instructions are inserted that command Romain to adjust the number of replicas. These steps can be applied by a PVF-aware compiler as well. After the modification, critical parts of the program are surrounded by an increase and decrease instruction.

When monitoring for example energy consumption or ambient temperature, the application does not need to be modified or adjusted to its execution in a multiple replica environment. It is Romain's task to evaluate the sensors' values and act accordingly. Therefore, the switch is placed in Romain or an external application. Monitoring helps to resiliently execute an application when environmental factors have an effect on the system's execution.

For a prototypical implementation, I decided to implement the switch inside the application due to the fact that I can easily test my implementation. Consequently, I wrote test applications containing the following calls to increase and decrease the number of replicas.

```
        asm volatile("int␣$0x41"); /* increase number of replicas */
        asm volatile("int␣$0x40"); /* decrease number of replicas */
```

Listing 3.1: Notify Romain

The interrupt instructions invoke the trap handling as described in Section 2.1.3. To process the notifications I implemented my own trap handler within Romain's trap handler chain. Because each trap handler is an observer I needed to add a new one. I didn't have to change Romain's software architecture, I just added new logic for dynamic replicas.

## 3.2 Decreasing the Number of Replicas

The decrease mechanism can be implemented in two different ways.

The replica can be set to sleep until it will be resumed. There is nothing to do except stopping the vCPU. No objects are freed and the replica retains its memory. Another option is to completely tear down the replica. This includes freeing all memory and terminating the vCPU.

If the application's execution switches rapidly from critical to non-critical code parts, the latter variant is slow compared to the first due to steady allocation and deallocation of memory. On the other hand, if for example the execution time of the critical code part is one percent of the total time it is worthwhile to implement the decrease mechanism by completely tearing down the replica. In this case we additionally save memory. How much time, memory and energy is saved depends on the execution time to completely disable a replica and on the rate of decrease and increase instructions.

| Resource usage | set replica to sleep | completely destroy replica |
|---|---|---|
| memory | retaining memory | freeing memory |
| execution | fast | slow |

Table 3.1: General assumptions about the decrease design decision

As you can see in Table 3.1, setting a replica to sleep implies retaining memory but the decrease operation is executed fast. If a replica is completely destroyed, its memory is returned to the system for further use, but decreasing takes more time to execute due to the free operation.

I decided to implement the decrease operation by setting a replica to sleep due to a simple and quick prototype implementation. That means, there is no resource deallocation at all and therefore I expected a speed-up compared to the second approach.

When an interrupt `0x40` occurs, the observer handles the decrease request. The number of currently running replicas is decreased by one and the replica which is going to sleep is executing an `l4_ipc_wait(utcb, label, L4_IPC_NEVER)` instruction. The crucial argument is `L4_IPC_NEVER` which forces the replica to wait until the master wakes it up.

## 3.3 Increasing Replicas

Increasing the number of replicas at runtime is the same procedure as creating a replica at start. The mechanism includes creating the following necessary objects: vCPU and task. The main difference is to bring the new replica into the same state such as all already running replicas.

During runtime the application's interim results and variables are stored in memory. Hence, we have to copy *all* memory regions from an existing replica into the new replica's address space. This includes for example the whole stack and the vCPU registers.

For the creation part I could reuse almost all of the existing startup code. However, I had to add code that takes care of adjusting the new replica's state.

The dynamic replica observer waits for the interrupt `0x41`. If there is no sleeping replica available the observer creates a new one. It allocates a new vCPU and task which are the components of a replica. The vCPU x86/32 registers are filled from a running replica. The vCPU's fault handling function is set to Romain's master which is called for any trap.

The master has access to every memory region of all replicas' address space and is responsible for the mappings as described in Section 2.1.3.1. To get the new replica into the same state the observer iterates over all mappings of an existing replica. The master creates a new dedicated copy of every memory region and maps them into the replicas' address space. All none-writeable memory regions are shared and hence no copy is required.

The new replica is registered in the master. When creating a replica we need to notify each fault handler using the previously mentioned startup function.

The last step is to synchronize the new replica with all the others. This is important otherwise, for instance, one replica resumes the execution and traps again while another replica is still in the trap handling and hence the replicas' states diverge.

### 3.3.1 Waking up Replicas

In the case we need to add a new replica and we already decreased the number of replicas before, Romain can take advantage of sleeping replicas.

Romain is aware of all sleeping replicas and their associated objects and memory locations. Now we need to wakeup a single sleeping replica and adjust its state to the currently running replicas. At this point, we can implement the following different solutions.

During the replica's sleep phase the application could allocate/free/modify any memory region. Therefore, we have two options. First, overwrite the whole address space of the wakeup-replica with the currently running replica's address space. Alternatively, we keep track of each modified memory region and only copy these regions to the wakeup-replica.

The wakeup operation is slow when the number of replicas frequently increase/decrease and the whole address space is copied compared to only copying the changes. On the other hand, if there are small changes in nearly all memory regions we need to execute a lot of *memcpy()* operations compared to copying large chunks of memory.

Additionally, saving the changes leads to a worst-case scenario. Imagine sleeping replicas which never resume. In that case, tracking all memory operations for these replicas consume system's resources for no reward.

I decided to not track the memory modifications due to a simpler implementation compared to implementing a tracking mechanism. When at least one sleeping replica is available the dynamic replica trap handler wakes it up. The observer iterates over the mappings list of a running replica, deallocates every memory region and maps/copies all regions of a running replica. As a result, we save time for allocating a replica because it already exists.

# 4 Copy-on-write

In the fundamentals chapter, I pointed out that the increase in resource consumption makes replication infeasible in use cases where physical resources are scarce, such as embedded systems. Dynamic replicas improve on that by reducing resource consumption in non-critical program phases. Still, the number of resources increases with every running replica.

As described in Section 3.3, during a replica's creation, readable memory regions are shared and writable regions are copied. Many applications have write-once, read later behavior hence we can improve our memory management. Imagine an application which implements AES encryption [AES01]. With a certain number of rounds a secret message is encrypted. Each round executes the same operations with a different key. Those round keys are derived from the cipher key during the key expansion phase. From the memory management point of view, the round keys are written in the key expansion phase and only read during the encryption rounds. Let us extend the example by executing this application inside Romain with dynamic replicas. For each replica which is created after the key expansion phase (write operations) the dynamic increase operation makes a copy for each generated round key. These writeable memory regions are accessed read only until the application terminates. Therefore, the copies are not necessary. SEUs in the CPU will have no effect on these regions and SEUs in memory are detected and recovered by ECC. Hence, an improvement is to not copy the regions until a write page fault occurs.

In this chapter, I describe how a copy on write mechanism can be integrated into Romain's memory management to address such situations.

Copy on write [Tan09, Page 221] improves performance by reducing copying such as in the example before. However, if a replica wants to write a memory region, which was not replicated due to the copy on write mechanism, we need a copy. To illustrate the situation, suppose two replicas A and B working on the same data D. Replica A reads D, adding ten and storing the new value D* in the same location. When B executes the instructions afterwards it operates on D* instead of D. Hence, the replicas need an extra copy of D to reach the same state.

The copy on write mechanism optimizes memory allocation by making a copy only when it is needed. A copy is necessary when a replica wants to write a memory region as explained in the example before. To prevent a replica from overwriting another replica's result we need to intercept the write operation. Therefore, the copy on write mechanism sets these memory regions to read-only to enforce a page fault. When the write page fault occurs, each replica obtains its own copy of the region. In the best case, neither replica needs to modify any data and no memory copy operation is executed.

Figure 4.1 shows the transition that happens in the case where two replicas executing with a shared page and a write page fault occurs. The rectangles inside the replicas'

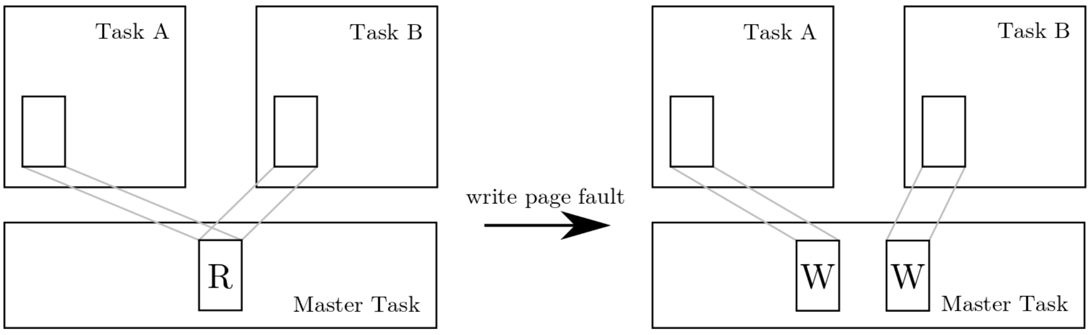

Figure 4.1: Copy on write with 2 replicas

task depict a memory region. As you can see, each replica receives its own copy after copy on write handles the write page fault. The scenario changes when adding dynamic replicas.

## 4.1 Dynamic Replicas add Complexity

As described in Section 3.2, there is no resource deallocation for a sleeping replica. Consequently, the scenario above changes such as depicted in Figure 4.2.

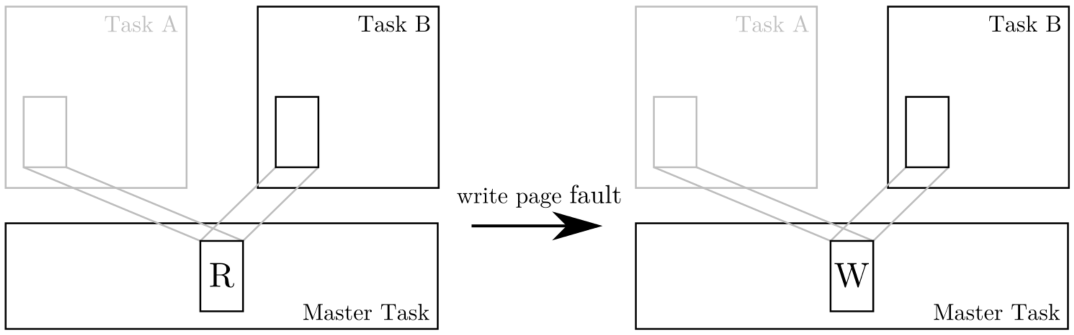

Figure 4.2: Copy on write with a single replica sleeping

The task showed greyed out symbolizes a sleeping replica A. The page's mapping into task A is still there but the copy on write mechanism does not need to consider replica A because it is set to sleep. Therefore, giving replica B the right to write into this page resolves the fault. If there were more than one running replica, Romain needs to copy such as described in the section before but can ignore all sleeping replicas. Once replica A wakes up no assumptions are made about such a modification hence the master adjusts A's task by copying the memory region's content from an active replica as described in Section 3.3.1.

## 4.2 Implementation

I extended Romain's memory manager to support copy on write.

Read page faults are resolved as described before. The master thread obtains the missing page if not already there and maps it into each *active* (non-sleeping) replica. Write page faults are resolved as described in Section 4.1.

A special situation comes up when a replica is created and at least two replicas are already running. In this case, the increase operation needs to handle the copy on write mechanism. The scenario is illustrated in Figure 4.3.

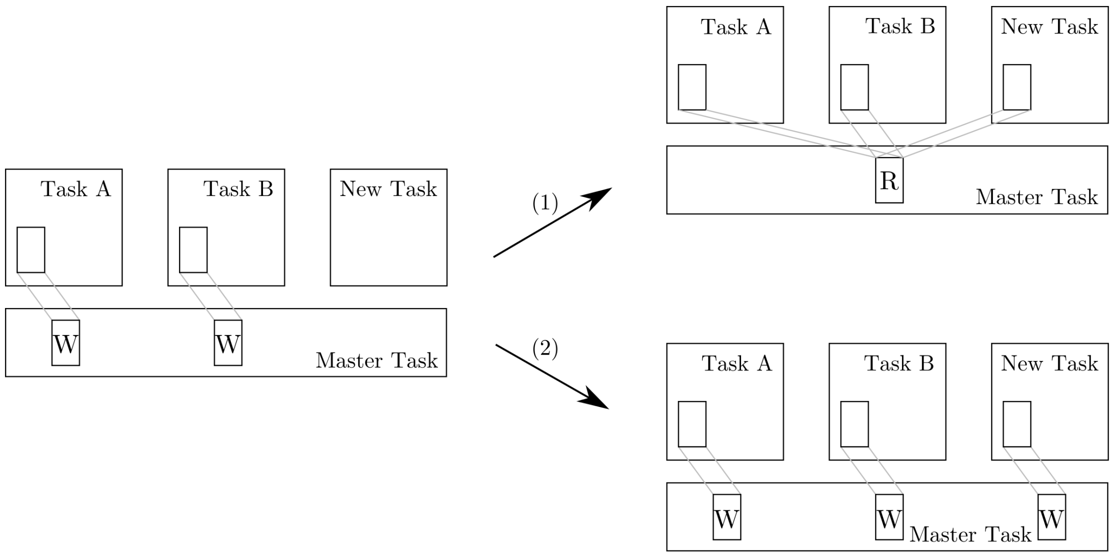

Figure 4.3: We can (1) make a shared read-only page or (2) make a copy of the existing page

In this example, we got two existing replicas and creating a new one. Both existing replicas have their own copies of the same page. When adding a third replica we can either chose to create one read-only mapping (1) or create a third writeable copy of the respective region (2). If the new replica obtains its own copy, Romain never has to run the copy on write mechanism except if more than one replica was created on start with shared writable pages. There is a worst-case scenario, suppose that multiple replicas share a page and a write page fault occurs. The trap handling makes a copy of this page for each replica. Thereafter, the number of replicas increases, therefore the replicas share the page and all copies are freed. Depending on the rate of such trap sequences, there can be a considerable execution time overhead due to the allocate/deallocate operation compared to memory management without copy on write.

I implemented scenario (1) from Figure 4.3 due to the properties of the copy on write mechanism as described in the beginning of this chapter.

### 4.2.1 More Optimizations

There are writeable memory regions that are worthless to set to read-only such as the stack of each replica. When processing a write page fault, the page fault handler could compare the faulting address with the current replicas' stack/base pointer. This information can be used by the memory manager to copy the stack or any other memory region wherein writing happens frequently. Alternatively, Romain can enforce to never set these regions to read-only. Therefore, an optimized version of the implementation could be a hybrid solution between variant (1) and (2) of Figure 4.3.

Romain assumes memory with ECC but state-of-the-art ECC does not detect/correct 100% of memory errors[HSS12]. Therefore, we need to replicate all memory to detect and recover all errors. An improvement could be to preallocate a big chunk of memory for each replica. This mechanism can speed up write page fault handling because memory is already allocated and only mapping is required.

# 5 Evaluation

In this chapter, I describe the performance tests I executed to evaluate my implementation. The test machine was based on an Intel i5-3550 Quad-Core-CPU (3.3 GHz) with 4 GB of RAM.

At first, I evaluate the dynamic replica trap handler and thereafter the copy on write mechanism. I begin with dynamic replicas and examine the three operations described in Chapter 3 - *decrease*, *increase* and *wakeup*.

## 5.1 Dynamic Replicas

As described in Section 3.1, I designed the notification mechanism to be controlled by the application itself. To test my implementation I wrote an application with for-loops and increase/decrease instructions such as in the following code.

```
/* [ ... ] */
for (i=0; i<100; ++i) {
        asm volatile("int␣$0x41"); /* increase number of replicas */
}
for (i=0; i<100; ++i) {
        asm volatile("int␣$0x40"); /* decrease number of replicas */
}
/* [ ... ] */
```

Figure 5.1: Test application

I measured the number of cycles the operations need to finish inside the dynamic replica trap handler. For the decrease and increase operations the test application looks like the code in Figure 5.1. One hundred replicas are created and set to sleep afterwards. I expected faster trap handling while the number of replicas decreases due to less overhead during the synchronization and replication phase. Therefore, the greater the control variable $i$ in the for-loop the faster the decrease-operation test and the slower the increase-operation test. In the case of the wakeup operation, I added another for-loop with `asm volatile("int␣$0x41");` inside. The for-loop was placed at the end of Figure 5.1 to trigger the wakeup code. As well, I expected slower execution due to the increasing load. I measured one hundred times the execution time of each operation inside the trap handling. The results are depicted in Figure 5.2.

Except for the wakeup operation, the tests went off as expected with stray values included. The wakeup operation's execution time increases with an increasing number of replicas as well, but this implementation seems like far from a predictable execution

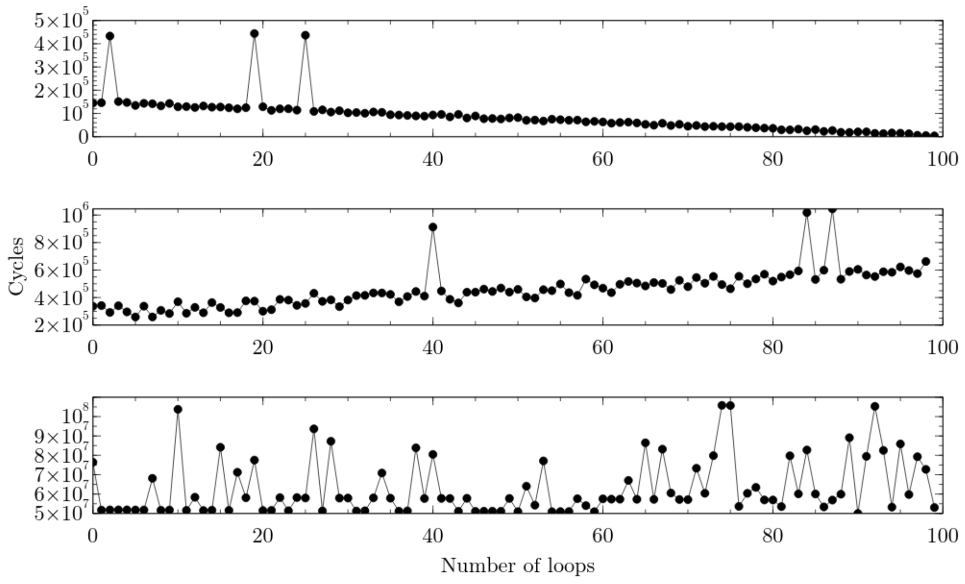

Figure 5.2: One hundred dynamic replicas: decreasing(top) increasing(middle) waking up(bottom)

time. The average of the first fifty measurements of the wakeup operation are lower than the second fifty. The top graph indicates, to decrease one hundred replicas to ninety-nine takes 144565 cycles and decrease two replicas to one replica takes 3628 cycles. Increasing one replica to two replicas takes 336812 cycles and increasing ninety-nine to one hundred replicas takes 662288 cycles as the middle graph indicates.

To examine the causes for the stray values in all operations I conducted a more detailed test. The test application remained the same but I inserted more measurement points to obtain the inconsistent code parts. Therefore, I tested each operation one hundred times with measurements for their code parts. The results are depicted in Figure 5.3 (decrease), Figure 5.4 (increase) and Figure 5.5 (wakeup). Each figure shows the code parts the operation needs to pass in ascending order. To get an overview, the code parts' proportion of the overall execution time is depicted as well.

The decrease operation's execution time is dominated by the resume part. In there, the trap handling replica signals all waiting replicas to replicate the results of the trap handling. Afterwards, the trap handling replica sets itself to sleep. The three stray values from the top of Figure 5.2 belong to the three in the 5th program part.

Figure 5.4 shows that the most time of the increase operation is spent in code part three. This part allocates memory for the vCPU and aligns the vCPU's handler stack to 4KB. The *memalign()* function varies almost every execution and is the reason why the middle of Figure 5.2 is not a straight line such as in the top graph.

In Figure 5.5, part three and two need the most execution time and they are the one with the most inconsistent results. Both parts working with the C++ data structure *std::map* with an integer as key and value. The map indicates mapped pages and in part two these pages are unmapped and freed if the replica exclusively owns this page.

The 3rd part fills the map again with valid entries from a running replica and therefore attaches/allocates and maps the pages. Memory management again is the reason for the stray values. As you can see in Figure 5.2, each value in the wakeup graph (bottom) is higher than each value in the increase graph (middle).

It turns out that waking up a replica costs more than creating a new one due to the fact that we need to remap almost all of the replicas memory. I loose a substantial fraction of the benefit I intended to gain by not freeing the memory inside the decrease operation. Therefore, I took the wrong design decision to place the replica's clean up code inside the wakeup operation. If we want to speed-up the wakeup mechanism we have to put the clean operations somewhere else. We can place them inside an extra garbage collector which is independent of the increase and decrease operations. Another option is to completely destroy a replica inside the decrease operation but then no wakeup functionality is needed anymore.

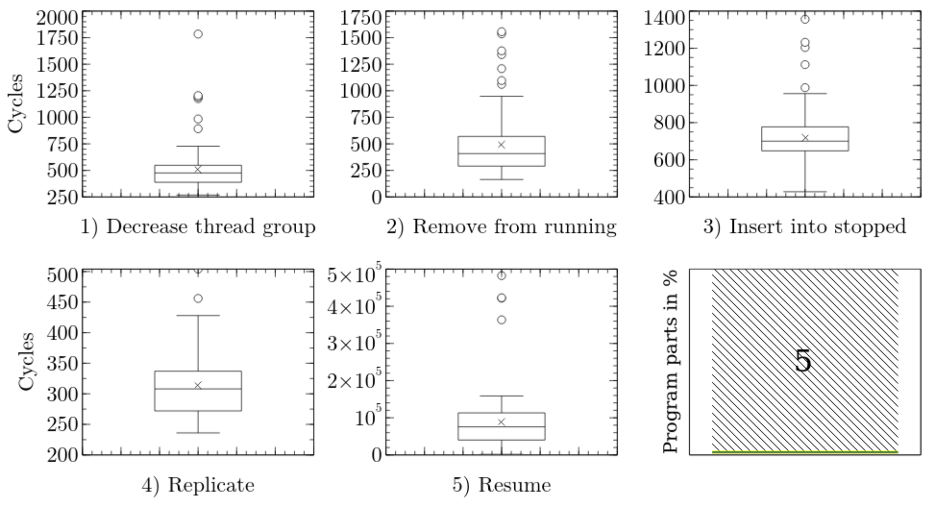

Figure 5.3: Decrease-operation's program parts execution time in cycles

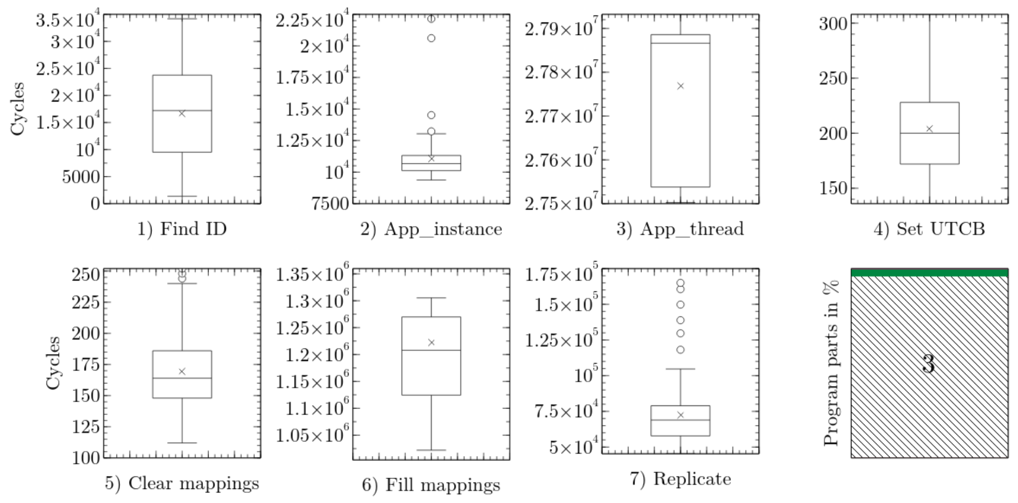

Figure 5.4: Increase-operation's program parts execution time in cycles

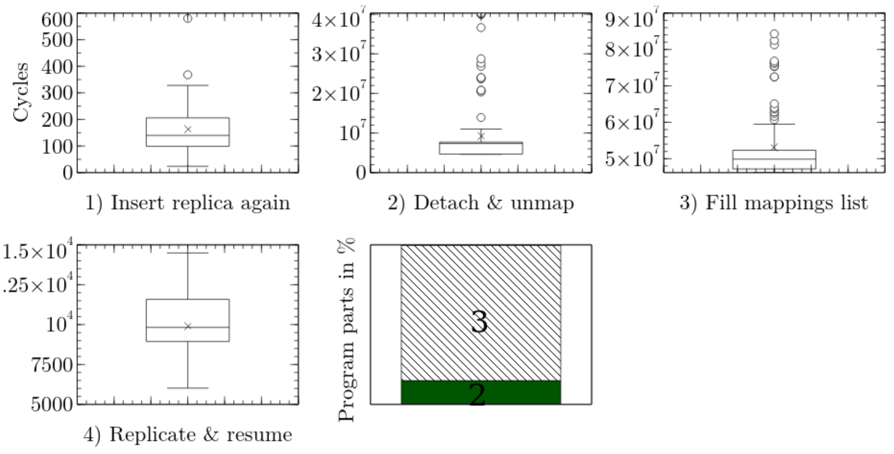

Figure 5.5: Wakeup-operation's program parts execution time in cycles

### 5.1.1 Micro Benchmarks

After examining the operations in detail, I tested dynamic replicas with a part of the MiBench benchmark suite [GRE+01]. I needed to modify the benchmarks because Romain waits for an external notification to adjust the number of replicas. The problem is the placement of the increase and decrease notifications inside the benchmarks. I ended up with two versions per benchmark. One increases the number of replicas at the benchmarks' start and decreases at the benchmarks' end (trivial dynamic replicas). These benchmarks allow me to compare their execution time with and without dynamic replicas. I expected a little overhead with dynamic replicas due to the additionally inserted interrupt calls. In the second version of the benchmarks I inserted the increase and decrease notifications so that the number of replicas is frequently/non-frequently adjusted to demonstrate how important the placement of the notifications is. Therefore, the interrupts are added inside (frequent) and outside (non-frequent) loops. For each replica I will mention the placement of the interrupts.

| | MiBench | | Trivial DR | | Dynamic Replicas | |
|---|---|---|---|---|---|---|
| auto_bitcnt | 0.01600 | 0.00462 | 0.01620 | 0.00039 | 0.16200 | 0.00900 |
| auto_qsort | 0.45644 | 0.03005 | 0.48000 | 0.03481 | 0.45854 | 0.02339 |
| net_dijkstra | 0.04554 | 0.00910 | 0.05927 | 0.01102 | 0.05327 | 0.00899 |
| auto_susan | 0.16072 | 0.01453 | 0.17630 | 0.01110 | 0.67040 | 0.02575 |
| auto_basicmath | 10.4820 | 0.01350 | 10.5034 | 0.34181 | 10.1702 | 0.29832 |

Table 5.1: MiBench tests: Arithmetic average and standard deviation in seconds

In Table 5.1, the columns describe the placement of the interrupts inside the benchmarks. The left value inside the columns is the average of ten runs of each benchmark and the right value is the standard deviation. In column one there is no modification at all and I run the test by using triple modular redundancy from the beginning. In column two and three the benchmarks initially started with a single replica and got more replicas dynamically added. I called column two *trivial dynamic replicas* because in that case you could have initially started with three replicas instead of increasing the number at start and decreasing it at the end. In the last column are the results for the benchmarks which I modified such as shown in Figure 5.6 to show some use cases.

```
/* [ ... ] */
int i;
int result = 0;
File* f = fopen(argv[1]);

asm volatile("int␣$0x41"); /* bitcnt, qsort, dijkstra, basicmath */
for (i=0; i<100; ++i) {
        printf("Starting␣loop␣#%i\n", i);
        asm volatile("int␣$0x41"); /* susan */

        /* calculation */

        asm volatile("int␣$0x40");
        printf("Intermediate␣result␣=␣%i\n", result);
}
asm volatile("int␣$0x40");
/* [ ... ] */
```

Figure 5.6: Example benchmark

Starting with *bitcnt*, *qsort* and *dijkstra*, all these tests show the same behavior. The unmodified benchmarks execute faster than the benchmarks with dynamic replicas. The longest execution times were measured by the trivial dynamic replica benchmarks.

The bitcnt benchmark contains one for-loop. I put the interrupts around this loop. Therefore, the difference between the trivial and the non-trivial version of the benchmark is its startup code such as initialising local variables.

In the qsort benchmark, the startup code contains a file open operation. Increasing the number of replicas afterwards is worth due to the fact that the system call is only executed for one replica. This benchmark calls the qsort() function of the system's C library around which I placed the interrupts. The unmodified version is still faster than the version with dynamic replicas because of the varying execution time of the memalign() call inside the increase operation. In the dijkstra benchmark we see the same results but with more diverging arithmetic average.

The modified version of the *susan* benchmark shows that dynamic replicas can slow down the execution when adjusting replicas at the wrong time. In this case, I implemented the interrupts inside a nested long running for-loop. In each loop, the application executes some mathematical operations but in the modified version additionally the benchmark traps two times. Therefore, we see more than 300% overhead between the unmodified and the modified version.

```
/* [ ... ] */

/* calculation one */

/* [ ... ] */

asm volatile("int␣$0x41"); /* susan */
/* longest executing calculation */
asm volatile("int␣$0x40");

/* [ ... ] */

/* calculation nine */
}
/* [ ... ] */
```

Figure 5.7: Example benchmark

The *basicmath* benchmark is implemented with nine mathematical calculations in sequence. For the dynamic replica benchmark, I implemented the interrupts around the calculation with the longest execution time as shown in Figure 5.7. The basicmath benchmark prints its results immediately after the calculation finished. In the case of the unmodified version and the trivial modified version, all system calls are handled for three replicas. In contrast, the dynamic replica benchmark with only three replicas inside the longest executing calculation, executes all other calculations with only a single replica. The basicmath benchmark is the longest running according to the others I tested. Therefore, the varying execution time of the memalign() function is clearly evident by the standard deviation of the basicmath's modified benchmarks.

## 5.2 Copy on Write

I tested the copy on write mechanism with a synthetic benchmark. The application consists of a loop wherein 1MB is allocated, modified and freed and the time for these operations are printed out. The loop runs ten times.

Each test ran with TMR on start except for the tests with dynamic replicas. In this case the application started with a single replica and the number increased to three during runtime. I executed the test application ten times for each of the following four different configurations:

1. Copy on write + dynamic replicas
2. Copy on write
3. Dynamic replicas
4. Without both

Therefore, I can compare copy on write with and without dynamic replicas against Romain without these features. Table 5.2 shows the results. Each row shows the number of system calls and traps, the standard deviation and the arithmetic average of all tests.

| | Average Execution Time | Standard Deviation | Traps | Systems Calls |
|---|---|---|---|---|
| COW + DR | 0.32100 | 0.03107 | 401 | 342 |
| COW | 0.29683 | 0.03682 | 396 | 342 |
| DR | 0.31320 | 0.03524 | 401 | 342 |
| W/O | 0.31535 | 0.02957 | 394 | 342 |

Table 5.2: Copy on write test results with execution time and standard deviation in seconds

The test application with the copy on write mechanism enabled executes most rapidly. The configuration with the original Romain executes almost as fast as the test with the dynamic replicas. The combination of copy on write and dynamic replicas comes last.

The last column shows the number of system call traps which are equal for each test. The trap column values are different because they are calculated from the dynamic replica interrupts and page faults.

The COW+DR and the DR test have the same amount of traps. This results from the fact that the copy on write mechanism is implemented in the replica creation process. Between an increase and decrease operation is no extra page fault caused by copy on write. Therefore, the copy on write is handled during these operations only.

As you can see from the COW test and the W/O test the first is faster but generates 2 more traps. Therefore, the time to handle 2 page faults in the trap handling has no visible impact compared to the benefit gained from the copy on write mechanism.

## 5.3 Resource Consumption

In this section, I evaluate the influence of dynamic replicas on resource consumption (RC). Resources are CPU, energy and memory. Each replica needs a certain amount of these resources. Three replicas consume 50% more resources than two replicas. Moreover, running two replicas requires Romain to consume twice as much resources as running a single replica. I measured an abstract resource consumption with dynamic replicas considering the execution time. The consumption is depicted as red and green areas in Figure 5.8

I performed the tests with the *bitcnt* benchmark of the MiBench suite. I executed the whole content of the benchmark one thousand times. As described in Section 5.1.1, I put the replica increase and decrease operations around the for-loop. In each of the thousand iterations, an additionally added code part decides whether to increase the number of replicas by one. All executions with dynamic replicas start with two replicas and maximally increase to three. In the end of each of the thousand iterations, the benchmark executes the replica decrease operation. I tested with the following different *frequencies*: 2, 4, 10, and 40. Each number implies how often dynamic replicas are increased and decreased. For example, the number 2 stands for using dynamic replicas every other iteration. Figure 5.8 illustrates the abstract resource consumption of these tests.

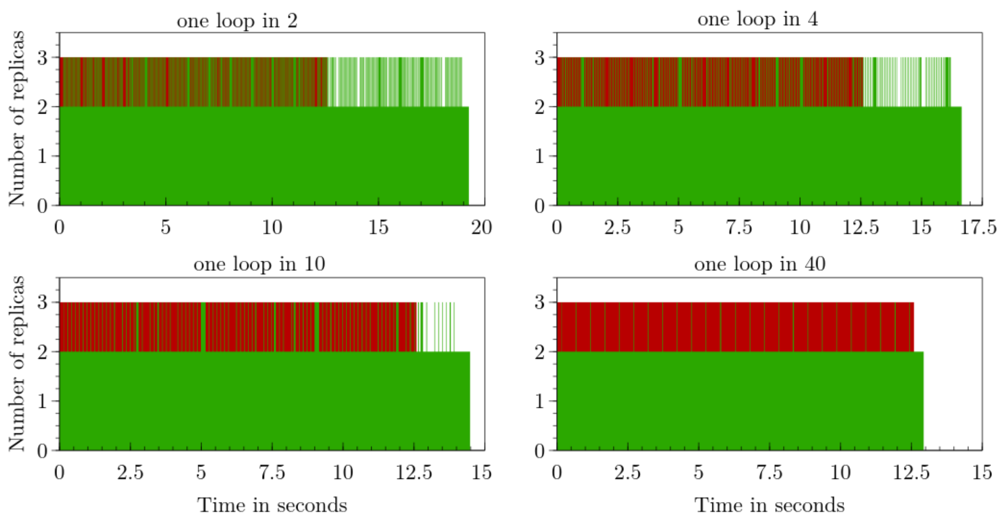

Figure 5.8: Resource consumption with different frequencies of dynamic replicas

The differences between the red and green areas depict the abstract resource consumption ratio between the execution with (green) and without (red) dynamic replicas. The red area is the bitcnt benchmark executed thousand times with triple modular redundancy and its surface area is the same in each graph. The green areas are executions with different frequencies of dynamic replicas. In Table 5.3, I listed the surface areas of the red and green figures. A small number implies less resource consumption compared to a large number.

| Frequency | 2 | 4 | 10 | 40 | Static TMR |
|---|---|---|---|---|---|
| Total RC | 52.071 | 43.607 | 34.43 | 29.436 | 37.77 |
| RC with TMR | 27.567 | 14.994 | 5.811 | 1.638 | 37.77 |
| Proportion of TMR's ET | 48.5% | 30.87% | 13.38% | 4.22% | 100% |

Table 5.3: Abstract resource consumption and the proportion of TMR's execution time

The first four columns show the results for all green graphs from Figure 5.8. The last column indicates the result of the static TMR experiment. The first row shows the total surface area (abstract resource consumption) of all graphs. A fraction of the total resource consumption is shown in the second row. These values represent the total amount of resources that were consumed when triple modular redundancy was turned on. The last row indicates TMR's proportion of the total execution time (ET). As you can see from Table 5.3, using three dynamic replicas every other or every fourth iteration consumes more resources than executing the benchmark with three static replicas. On the other hand, using three dynamic replicas every tenth or every fortieth iteration consumes less resources than the static execution.

For this benchmark and considering system resources, it is worth to use dynamic replicas every tenth iteration. In other words, resources for this execution are saved if the percentage of executing three replicas is less than or equal to 13.38% of the whole execution time. In this benchmark, I decided to turn dynamic replicas on and off for each loop. Therefore, the results include overhead from a lot of increase and decrease operations. I would expect less overhead and therefore less resource consumption with dynamic TMR when three replicas would run for ten consecutive loops or more. In this case, fewer increase and decrease operations would be executed.

This experiment makes no adoption of the benchmark's resilience. For instance, imagine we need to cover at least 30% of the execution time with TMR to guarantee resilience for this benchmark. Consequently, it is worthless to use dynamic replicas because static TMR needs less resources than three dynamic replicas. Therefore, resilient execution and the amount of resources that can be saved depend on the application itself and on the desired resilience.

# 6 Summary and Conclusion

In this thesis, I described my implementation of dynamic replicas for Romain, an error detection and recovery framework. At first, I started with Romain's fault tolerance model and how it is used to provide resilience. Next, I described Romain's internals and why multiple replicas increase the resource consumption of the soft error detection and recovery. In Chapter 3, I introduced dynamic replicas, I described their design and a prototypical implementation. Chapter 4 described another approach to reduce the resource consumption of multiple replicas - copy on write - and its implementation in Romain. Thereafter, I tested my implementation and presented the results in Chapter 5. The experiments showed, that using dynamic replicas can speed up execution time but deployed in the wrong way it can be an obstacle and slows down the execution. Next, I evaluated the copy on write mechanism. The experiment showed, when executing a memory-bound application with copy on write it executes faster than the application without the mechanism. Finally, I measured abstract resource consumption by using dynamic replicas with different frequencies. The results showed, when increasing and decreasing the replicas too often we increase resource consumption, but on the other hand lower frequency reduces the resource consumption.

## 6.1 Conclusion

I implemented dynamic replicas inside Romain, the replication framework of TUD:OS. The number of replicas can be modified by the application. The program notifies Romain to decrease or increase the number of replicas. By decreasing the number of replicas we reduce memory consumption and speed-up the overall execution time. On the other hand, dynamic replicas can slow down Romain's execution when they are frequently used. If we can turn off replicas without loosing resilience then dynamic replicas can reduce system resource consumption.

The copy on write mechanism can be used to improve execution time for applications that heavily allocate memory.

## 6.2 Future Work

It would be interesting to test an implementation where the interrupts are placed inside a program which is not Romain or the redundantly executing application. Therefore, I think about more realistic experiments for example with an ambient temperature sensor which notifies Romain to adjust the number of replicas when a certain threshold is reached.

In addition, replica's memory is allocated when its needed. If we provide each replica with a sufficient chunk of memory on creation, we can speed-up the application's execution time by reducing the amount of memory allocation calls. Hence, the trap handling does not need to allocate memory, which was a problem in the experiments, and therefore a more consistent execution can be expected. The question is how to simplify the memory management itself to improve Romain's performance?

Another improvement would be to extend the area where soft errors could come from, for instance memory without ECC. Hence, instead of comparing the replicas' whole address spaces you could start a separate thread on an extra CPU which continuously compares read-only memory regions. If the thread detects a fault it notifies Romain and the memory region is handled on the next trap. On the other hand, we could replicate memory as well. In this case, Romain can be used on systems without ECC but the drawback is an increased resource consumption.

When waking up a replica, almost all of the replica's memory is cleaned up and remapped. A memory-operation tracking system as suggested in Section 3.3.1 can improve the execution as well. Therefore, the wakeup operation would not need to cope with the whole address space and in collaboration with a garbage collector it just copies the modified memory regions. Also, a garbage collector can simplify the memory management inside the trap handling. Hence, further work could contain the implementation of a memory garbage collector or a memory tracking system to tear down replicas in a more efficient way.

Through time reasons, I could not evaluate the resource consumption in a wider range. Experiments with larger quantity of applications may give an answer to the following question. How much time of the total execution time can be used for dynamic triple modular redundancy and still saving resources compared to an execution with static triple modular redundancy?

# Bibliography

[AES01] Specification for the Advanced Encryption Standard (AES). `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`, 2001. 15

[BLL$^+$11] Matthew J. Breitwish, Chung H. Lam, Hsiang-Lan Lung, Bipin Rajendran, Aljeandro G. Schrott, and Yu Zhu. Phase Change Memory Device and Method of Manufacture, United States Patent 7,868,313 B2, January 2011. 3

[DH12] Björn Döbel and Hermann Härtig. Who Watches the Watchmen? - Protecting Operating System Reliability Mechanisms. *International Workshop on Hot Topics in System Dependability (HotDep 2012) Hollywood, CA, USA*, October 2012. 7, 5

[DHAE12] Björn Döbel, Hermann Härtig, Philip Axer, and Rolf Ernst. ASTEROID - Analyzable, Resilient Real-Time Operating System Design. *DATE 2012, poster session, Dresden*, February 2012. 7, 1, 4

[DHE12] Björn Döbel, Hermann Härtig, and Benjamin Engel. Operating System Support for Redundant Multithreading. *International Conference on Embedded Software (EMSOFT'12), Tampere, Finland*, 2012. 1, 4

[DSE13] Björn Döbel, Horst Schirmeier, and Michael Engel. Can we use PVF Analysis to Quickly Approximate Program Vulnerability? *5th Workshop on Design for Reliability (DFR 2013), Berlin, Germany*, Januar 2013. 9

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns / Elements of Reusable Object-Oriented Software.* Addison-Wesly, 1994. 8

[GRE$^+$01] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. *In Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, 2001. 22

[HANR12] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults. *In Proceedings of ASPLOS, pages 123–134*, 2012. 9

[HSS12] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the

Implications for System Design. *In Proceedings of the 17th international conference on Architectural Support for Programming Languages and Operating Systems* , pages 111–122, 2012. 18

[LWP10] Adam Lackorzynski, Alexander Warg, and Michael Peter. Virtual Processors as Kernel Interface. *Twelfth Real-Time Linux Workshop 2010, Nairobi, Kenya*, October 2010. 5

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 1965. 3

[Muk08] Shubu Mukherjee. *Architecture Design for Soft Errors.* Morgan Kaufmann, 2008. 3

[Mur08] San Murugesan. Harnessing Green IT: Principles and Practices. *IT Pro January/February 2008, Published by the IEEE Computer Society*, 2008. 3

[MWE$^+$03] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. *In the Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO)*, 2003. 9

[Sch90] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990. 4

[SK09] Vilas Sridharan and David R. Kaeli. Eliminating microarchitectural dependency from Architectural Vulnerability. *In HPCA 2009. IEEE 15th International Symposium on High Performance Computer Architecture*, pages 117–128, 2009. 9

[Tan09] Andrew S. Tanenbaum. *Modern Operating Systems 3rd Edition.* Pearson Prentice Hall, 2009. 4, 15

[TD13a] Chair of Operating Systems TU Dresden. Fiasco.OC. `http://os.inf.tu-dresden.de/fiasco/`, April 2013. 4

[TD13b] Chair of Operating Systems TU Dresden. L4Re. `http://os.inf.tu-dresden.de/L4Re`, April 2013. 4