Diplomarbeit

ELK Herder

Replicating Linux Processes with Virtual Machines

Florian Pester

February 15th, 2014

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

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

Technische Universität Dresden
Fakultät Informatik

Dresden, 01.08.2013

AUFGABENSTELLUNG FÜR DIE DIPLOMARBEIT

Name des Studenten: Pester, Florian

Studiengang: Diplom Informatik

Thema: *Romain on Linux*

As the rate of transient and permanent hardware errors increases, system designers need to implement fault tolerance mechanisms that allow applications to detect and correct those errors to avoid system failures. Commercial-off-the-shelf (COTS) computers make up the the majority of all end-user electronics and are unlikely to include efficient hardware-assisted fault tolerance any time soon. Furthermore, a large amount of existing software is unlikely to be rewritten using fault tolerant programming techniques or recompiled with a compiler generating resilient machine code.

The Romain replication framework leverages the increased available hardware resources in modern COTS systems and provides fault tolerance using replicated execution as an operating system service in Fiasco.OC/L4Re. By supporting binary-only applications, Romain allows to improve fault tolerance for unmodified legacy applications.

The goal of this thesis is to evaluate whether and how the Romain concept can be applied to Linux-based operating systems. This will demonstrate the suitability of OS-assisted replication in another operating system environment. The assignment will have to answer three main questions:

1. Which operating system or hardware mechanisms can aid replication by isolating replicas and allowing strict control over their execution? Linux ptrace mechanism as well as virtual machines based on KVM appear to be feasible starting points, but the student may explore alternatives as well.

2. How can Linux process resources (file descriptors, sockets, memory, signals etc.) be managed so that replication is transparent to the application and how can we transparently proxy Linux system calls for replicated applications? Existing experiences from Romain on L4Re and Valgrind may prove useful.

3. How do the applied measures affect performance, resource consumption, and error coverage?

The assignment shall focus on single-threaded applications first. The SPEC CPU 2006 or MiBench benchmark suites shall be used for evaluation purposes.

verantwortlicher Hochschullehrer:	Prof. Dr. Hermann Härtig
Betreuer:	Dipl.-Inf. Marcus Hähnel
Institut:	Systemarchitektur
Beginn:	15. 08. 2013
einzureichen:	15. 02. 2014

Unterschrift des betreuenden Hochschullehrers

Erklärung

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

Declaration

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Dresden, den 15.02.2014

Florian Pester

Acknowledgements

I want to thank Björn, my advisor, for a really great job. Thank you for countless pieces of advice, lots of valuable input and helpful criticism. I want to thank Professor Hermann Härtig for giving me the possibility to write this thesis. Many thanks to everyone in the OS student lab, who I could always discuss ideas with.

Furthermore I want to thank my parents, for making it all possible. And Maria, for everything.

Abstract

Today's software is prone to faults in the underlying hardware. We need fast and flexible solutions that deal with these faults in order to keep software reliable, even on commercial-off-the-shelf systems. Replication in virtual machines with a minimal proxy operating system provides for flexibility, because it lets users choose, which applications to replicate. Modern COTS hardware provides multiple CPU cores and virtualization support, minimizing replication overhead. I implemented ELKVM, a library that runs applications in a virtual machine with 2.2% runtime overhead. I present ELK Herder a replication manager, which runs these applications in a triple-modular-redundant fashion, providing fault tolerance and recovery, with 16.3% total runtime overhead. These developments make fault tolerance possible and affordable on low-end servers, and most consumer devices.

Contents

List of Figures

1 Introduction

During the last few decades, decreasing transistor sizes enabled faster and smaller devices [M+65]. Modern CPUs, even in mobile phones, have multiple independent CPU cores and modern systems have gigabytes of main memory. Unfortunately this development has a down-side as well. Smaller transistors are more susceptible to faults [SKK+02, HBD+13]. At the same time society's dependence on the systems built from these transistors increases rapidly. Therefore we need reliable and dependable systems. This can be achieved by fault tolerant systems.

Hardware assisted fault tolerance is expensive in both development and production, and therefore seldom built into commercial-off-the-shelf hardware, for which cost and time-to-market play an important factor for commercial success [Sie95]. Adding hardware-based fault tolerance to a system typically adds transistors to the system, therefore systems using hardware fault tolerance consume more energy. As these factors are unlikely to change in the future, it is best to improve the reliability of these systems using software techniques for fault tolerance.

I focus on a software-based detection method for transient hardware faults. These faults, which are also known as soft faults, are short-lived errors on a transistor [MAAB11, WEMR04]. Transient hardware faults may arise from cosmic rays or unsteady power supplies and usually result in bit-flips in either memory, or a processing unit [WEMR04]. Aging hardware components may also trigger these faults [AN97].

I present a solution for discovering transient hardware faults and recovering from them using the state machine approach presented by Schneider [Sch93]. In order to do so, I replicate applications in virtual machines and compare their states on a regular basis. The replicas run in parallel on different CPU cores in order to minimize runtime overhead. I present ELK Herder, a prototype to evaluate the performance of this approach for the 64-bit variant of the x86 architecture.

I evaluated the runtime overhead of ELK Herder with the SPEC CPU 2006 benchmarks. My evaluation shows a geometric-mean runtime overhead of 16.3% for the SPEC benchmarks, running with triple-modular redundancy. In order to show, that ELK Herder reaches its goal of detecting and recovering from transient hardware faults, I have added fault-injection capabilities to ELK Herder. My experiments with fault-injection into replicas show ELK Herder is able to detect and recover from all faults injected into a test binary.

Replication can only work, if the replicas behave deterministically. Deterministic execution of the replicas can be achieved by making sure that all replicas get exactly the same input at all times. On Linux, inputs reach applications via system calls, via signals or via shared memory. In order to intercept all system calls and signals for a given application I present a library called ELKVM. Replicating applications, which use shared memory is not a goal of this work. However, support for interception of shared

memory accesses can be added to ELKVM with the approach described by Mushtaq et al. [MAAB12].

ELKVM runs applications in virtual machines without a full-scale operating system inside and intercepts the application's system calls. I discuss the design and implementation of ELKVM. For system calls not related to memory management, system call interception with ELKVM is an order of magnitude faster than with the `ptrace()` [manc] system call found in Linux. For the evaluation I ran the SPEC CPU 2006 benchmarks with ELKVM, which showed a typical runtime overhead of 2.2%. ELK Herder uses ELKVM to intercept the replica's system calls.

Although I used ELKVM primarily for replication, the library can be used for a variety of other applications. Examples include memory-leak detection, system call tracing, debugging software and checkpointing and migration of applications.

I discuss a variety of preceding software-based approaches in Chapter 2, before discussing the benefit of my own approach in Chapter 3. Chapter 4 covers issues, that became apparent in the implementation phase of this work, and how I resolved these issues. An evaluation of the work is given in Chapter 5, before I conclude my work in Chapter 6.

2 Related Work

In this Chapter I introduce a classification of faults, errors and failures. I also introduce the differentiation between fault prevention, fault tolerance, fault removal and fault forecasting described by Avizienis et al. [ALRL04]. I then give an overview of other fault tolerance works.

The approach to fault tolerance I introduce in this work uses process replication in virtual machines. I chose KVM, the Linux Kernel Virtual Machine, as a hypervisor, because it can be controlled by Linux user-space applications. I conclude this Chapter with a short description of KVM.

2.1 Faults, Errors and Failures

According to the definition by Avizienis et al. "Dependable systems deliver service that can be justifiably trusted" [ALRL04]. Alternate definitions for dependability state, that dependability is the ability to avoid service failures [ALRL04]. A service failure happens, when the delivered service deviates from the correct or specified service. An error is "the part of the total state of the system, that may lead to its subsequent service failure". A fault is what causes the error [ALRL04].

DeKruijf et al. [dKNS10] describe the reasons for transient faults. Smaller devices comprise of less atoms and their transistors are therefore harder to "control in terms of their individual power and performance characteristics". As a result devices are more susceptible to transient faults due to particle strikes, old age or unstable power supplies [dKNS10, SKK+02].

Avizienis et al. describe three approaches to achieve a dependable system [ALRL04], despite the existence of faults:

1. Prevent faults from occurring at all. The influence of cosmic rays can be mitigated by several centimeters of concrete. "Unfortunately, this width is significantly greater than normal computer room roofs or walls." [RM00]. Thus prevention of transient faults due to cosmic rays potentially involves costly rebuilding.

2. A system may avoid service failures, in case faults happen. This is called fault tolerance. Fault tolerant systems provide a viable alternative to fault prevention.

3. Detect and remove the faults themselves, for instance by applying workarounds or patches.

Weaver et al. [WEMR04] classify the effects of transient faults in three different categories depicted in Figure 2.1.

Benign faults occur when transient faults do not affect the correctness of an application, i.e. do not trigger an error. Benign faults include faults in idle function units or pre-fetching instructions. Fault recovery always introduces overhead to a system, therefore the fault tolerance mechanism should ignore benign faults.

When transient faults modify the system in such a way, that a program produces false results but does not crash, a *silent data corruption* - SDC - occurs. Without a fault tolerance mechanism the program continues execution in a false state after the SDC. A fault tolerance mechanism, that supports only *fault detection*, terminates the program and presents a failure to the user after a silent data corruption. Fault tolerance mechanisms that support *fault recovery* can even correct SDCs and continue program execution.

Detected unrecoverable errors - DUEs - are transient faults, that have been detected by the fault tolerance mechanism but the mechanism cannot recover from the fault. The running program can only be terminated by the fault tolerance mechanism. For a fault tolerance mechanism that does not support fault recovery, SDCs are DUEs as well.

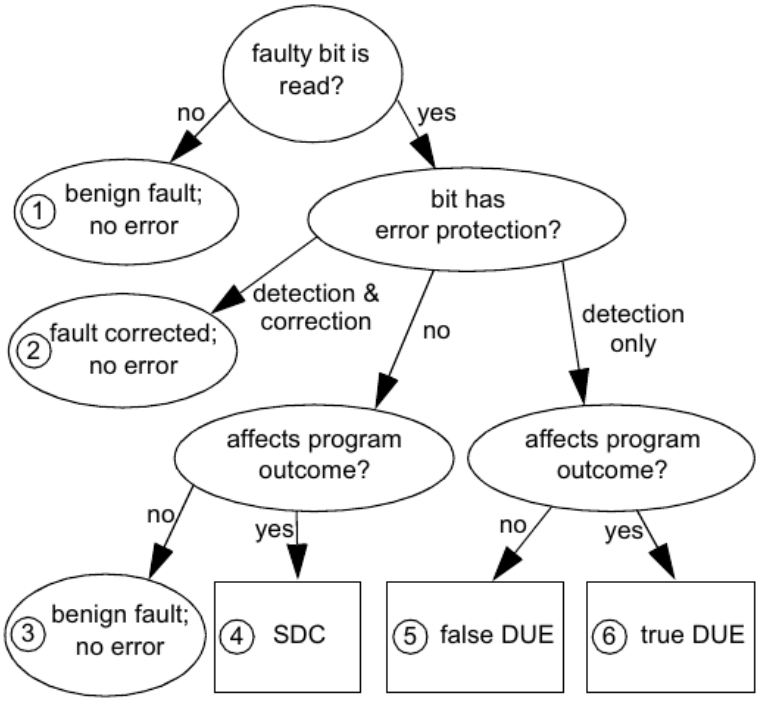


Figure 2.1: Classification of transient faults by Weaver et al. [WEMR04]

2.2 Fault Tolerance

A fault tolerant system is a system that can tolerate a number of faults during a specified interval. If the system can tolerate t faults, it is called t *fault tolerant* [Sch90]. Even

if faults trigger an error, the error must not trigger a service failure. According to Gray [Gra86] modularity and redundancy can be used to allow a module of a system to fail without affecting the availability of that system.

Redundant execution of software, also called replication, is a well researched topic. Schneider [Sch90] established that each replica of a software does the same thing as long as it starts in the same state, runs on a non-faulty processor and executes the same set of instructions. By comparing the output of the replicas it is possible to obtain the correct output of the software.

Bartlett [Bar81] describes a fault-tolerant, distributed system. The system is built from the ground up, with specialized hardware and software. He uses process pairs for fault detection and check-pointing to achieve fault recovery. Regardless of the underlying system, checkpointing is a good way to save and compare an application's state.

Reis et al. [RCV+05] discuss detecting silent data corruptions at the software level. The authors present SWIFT, a compiler which introduces redundant instructions into an application's binary code. Values are stored in multiple registers on the CPU and are compared to their counterparts whenever they are used. SWIFT requires a recompilation of applications before they can be used in a fault tolerant environment, which is impossible, if the source code for an application is not available. However SWIFT can be combined with other fault tolerance mechanisms, in order to achieve a fully protected system.

Fault tolerance mechanisms, which use replication only replicate well-defined parts of a system. The replicated part is called a "Sphere of Replication" by Reinhardt et al. [RM00]. Inside the Sphere of Replication, all data is protected by replication. All data entering the sphere must be replicated to ensure correctness. Inside the sphere additional measures like error-checking codes (ECC) [MP86] can be taken, but are not necessary. Upon leaving the Sphere of Replication, data is compared in order to be able to detect errors. Inside the sphere no comparisons are necessary, because errors are detected by output comparison upon leaving the sphere. To have a fully protected system it is important that all parts of the system outside the Sphere of Replication are protected by other fault tolerance mechanisms.

Rotenberg uses replication of processes to detect transient faults [Rot99]. His approach uses two replicas running on the same CPU with simultaneous multi-threading [EEL+97]. Modern COTS systems do not only feature simultaneous multi-threading, but also multiple cores on one CPU, which makes it possible to run multiple replicas in parallel on one system. This increases the speed of replica comparisons in contrast to comparison over a network.

Shye et al. introduce the use of replicated processes to discover faults at the software level [SBM+09]. State changing system calls are executed by only one of the process replicas, all other replicas only emulate the system call. Their fault tolerance software, PLR, compares the replica's outputs and register contents at each system call in order to detect faults.

According to the authors fault detection at the hardware level produces *false* DUEs, which are unrecoverable errors at the hardware level, but from process level, they are benign faults and can be ignored by a process-level fault tolerance mechanism. If faults are detected at the software level, rather than at the hardware level, false DUEs do not exist, because the state of the software is checked when the software communicates to the outside world, therefore only data relevant to the execution of the application is compared.

The ability to ignore as many benign faults as possible is interesting, because a replica reinitialization always introduces overhead. Therefore, if a fault tolerance software can ignore more benign faults, it introduces less overhead. Another benefit of this approach is, that replication at the process level does not need the source code of the replicated application. Due to these benefits the findings of Shye et al. are important for my work.

In their publication, Shye et al. examined, which properties are important to make sure the replicated applications behave the same way, as the applications would behave without replication. They identified the following six properties:

1. Modification or recompilation of the underlying hardware, operating system or the application itself is not necessary.
2. Replication is transparent.
3. Replicas maintain process semantics expected by the user.
4. The Sphere of Replication should be around the user-space application and its libraries.
5. Faults shall only be detected if incorrect data exits user space.
6. Replicas replicate the entire process virtual address space and process meta-data.

The first property is necessary for PLR to run on commercial-off-the-shelf hardware, without the need for the source code of the replicated application or the operating system. Properties 2 and 3 ensure, that replicated applications can use signals for communication with the rest of the system. Properties 4-6 allow PLR to ignore many benign faults, while still detecting all other faults.

PLR does not replicate the operating system. However, if OS protection is required, it is possible to protect the operating system with other fault tolerance mechanisms, such as SWIFT.

In order to create the replicas, the authors use binary recompilation and `fork()` the applications. This recompilation is not modification in the sense of property 1, because binary recompilation does not require the source code of the replicated application.

Replicas are compared when they do system calls. At least two replicas are needed to detect faults, while three replicas allow for fault recovery by majority vote.

One of the replicas becomes the master process. Only this replica is allowed to execute system calls, while system calls for all other processes are emulated. The authors mention that in cases where system calls do not “modify any system state”, the system call is not be emulated for the slave processes but done by all processes. I find this

behavior problematic, because the reply from the system call might change, if a completely independent process happens to change system state between system calls made by two replicas.

For instance if the replicas execute the `stat()` system call[1] and one of the replicas is delayed until after another, independent process executes a `write()` on the same file, the file's size has changed for the last replica.

Due to a transient fault, a replica might stop executing system calls. In order to recover from this case PLR uses a watchdog timer, which flags an error once a timeout has been reached.

After the replicas have started execution, the original process waits for the replicas to finish and is responsible for signal handling. The `SIGKILL` and `SIGSTOP` signals do not allow a process to define a handler routine, but terminate a process instantly. This leaves no way to kill the replicas once the original process received one of these signals. In order to be able to react to the `SIGKILL` and `SIGSTOP` signals a monitor process is introduced, which kills replicas in case the original process does not exist any more.

Bressoud and Schneider first describe a replica-based approach to fault tolerance, which uses virtual machines [BS96]. However they use a full-scale operating system inside their virtual machines and need a network to communicate between the VMs. They also only support two replicas, limiting them to fault detection. For I/O intensive operations their hypervisor takes twice the time than running the application natively, even on two physical CPUs.

Jeffery and Figueiredo propose the use of a hypervisor to implement fault tolerance in PCs and low-end servers with replication [JF07]. Their replicas run in parallel on the same machine. Again, the authors use full-scale operating systems inside the VMs, which complicates the elimination of nondeterminism in the inputs.

A combination of hardware-assisted virtualization and replication on the process level promises good performance. Hardware-assisted virtualization keeps the virtualization overhead low and multiple cores or CPUs in one system enable replicas to run in parallel. Most modern COTS hardware provides for both. On the software side, the KVM hypervisor, present in the Linux kernel, provides a well documented API, which allows to control the execution of virtual machines.

2.3 KVM

In order to implement replication for Linux, I use KVM as a hypervisor. In this section I introduce KVM and how KVM is used. Further information can be found in the KVM Documentation [kvma].

KVM is the Linux Kernel Virtual Machine, a hypervisor built into the Linux kernel. It executes virtual machines as normal Linux processes and provides an API to control the virtual machines from the host's user space.

[1] The stat system call is not explicitly mentioned by Shye et al., but to my knowledge changes no system state, and would therefore be executed by all replicas

```
1 int fd = open("/dev/kvm", O_RDWR);
2 assert(fd > 0);
3
4 int vmfd = ioctl(fd, KVM_VM_CREATE, 0);
5 assert(vmfd > 0);
6
7 int vcpufd = ioctl(vmfd, KVM_VCPU_CREATE, 0);
8 assert(vcpufd > 0);
```

Listing 2.1: Creating a virtual machine

KVM adds the guest execution mode to Linux processes and uses that mode to execute all non-I/O guest code. Kernel mode is used to switch between host and guest. KVM consists of a device driver to manage virtualization hardware such as virtual CPUs and memory. A user-space component, most commonly QEmu [Bel05], performs I/O.

Virtual machines managed by KVM are scheduled by the standard Linux scheduling mechanism and memory is managed by the Linux memory allocator. All Virtual Machines belong to the user that started them, they can be viewed with `top` and destroyed with the `kill` command [kvmb].

For x86 CPUs KVM requires virtualization extensions like Intel's VMX [man13] or AMD's SMX [man11].

In Listing 2.1 a KVM virtual machine, with one virtual CPU is created. As a first step, the virtual machine monitor (VMM) opens the character device, provided by KVM, for reading and writing. The VMM can use the file descriptor returned by the open call, to make requests via the `ioctl()` system call. As a second step the VMM creates a new virtual machine with an `ioctl()` call on the file descriptor and `KVM_VM_CREATE` as an argument.

This call returns a new file descriptor, which is used to control the virtual machine. In line 7 of the listing, the VMM calls another `ioctl()` to create a new virtual CPU with the virtual machine file descriptor. Again this `ioctl()` call returns a file descriptor. This file descriptor is used to control the VCPU.

Before the virtual machine can run, it needs some memory. Listing 2.2 shows how 1 MiB of memory is mapped to a virtual machine. First, the VMM acquires the memory to map to the virtual machine. KVM requires this memory to be aligned to the page size of the host machine. Second, the VMM, creates a `kvm_userspace_memory_region` structure. This structure holds information about the memory's host address, its guest address, size and a slot number. The slot number serves as an identifier and must be unique to the memory region. In order to finally map the memory region, the VMM passes the VM file descriptor, `KVM_SET_USER_MEMORY_REGION` and a pointer to the `kvm_userspace_memory_region` in an `ioctl()` call to the virtual machine.

KVM can only handle 128 memory regions, called slots, in Linux 3.11. As a further limitation, memory regions passed to KVM cannot be resized.

After the memory is mapped, the VMM retains access to the memory. The VMM can now load a binary into the memory. Once this is done, the virtual machine needs

```
1  void *ptr = NULL;
2  uint64_t memsize = 1 * 1024 * 1024;
3  int err = posix_memalign(&ptr, PAGESIZE, memsize);
4  assert(err == 0);
5
6  struct kvm_userspace_memory_region mem;
7  mem.slot = 0; //region id, must be unique to this region
8  mem.flags = 0;
9  mem.guest_phys_addr = 0x0;
10 mem.memory_size = memsize;
11 mem.userspace_addr = (uint64_t)ptr;
12
13 err = ioctl(vmfd, KVM_SET_USER_MEMORY_REGION, &mem);
14 assert(err == 0);
```

Listing 2.2: Mapping memory to a virtual machine

```
1  struct kvm_regs regs;
2  int err = ioctl(vcpufd, KVM_GET_REGS, &regs);
3  assert(err == 0);
4
5  regs.rip = 0xc0ffee;
6  /* modify other registers here */
7
8  err = ioctl(vcpufd, KVM_SET_REGS, &regs);
9  assert(err == 0);
```

Listing 2.3: Access to the VCPU registers

to know, where to start running. The virtual CPU's registers can be manipulated to provide that information.

In Listing 2.3 the VMM creates a `kvm_regs` structure, and calls `KVM_GET_REGS` on the VCPU file descriptor with a pointer to the structure to get the current register values of the virtual CPU. Then the VMM needs to set the guest's `rip` register to the entry address for the guest binary. For this example the entry point is `0xc0ffee`, which is set in line 5 of the listing. The new register values are written with a call to `KVM_SET_REGS`.

KVM also provides access to special registers like `CR0` through `CR4` and the segmentation registers. The `kvm_sregs` structure and `KVM_GET_SREGS` substitute the arguments to the `ioctl()` call in this case.

Now that the virtual machine has a VCPU, memory for a binary and an entry point into the binary, the VMM can start the virtual CPU. Listing 2.4 shows an example run-loop for a virtual CPU. In the first lines, the VMM maps the `kvm_run` structure, which is used by KVM to provide information about the VCPU. Note, that the VCPU file descriptor is used as an argument to the `mmap()` call.

The VMM creates a flag to indicate, if the VCPU shall run again. The `ioctl()` call inside the loop starts the virtual CPU. This `ioctl()` takes the VCPU file descriptor and `KVM_RUN` as arguments. KVM stores VCPU's exit reason in the `kvm_run` structure. This information can be used to handle the VCPU exit.

```
struct kvm_run vcpu_run;
vcpu_run = mmap(NULL, sizeof(struct kvm_run), PROT_READ | PROT_WRITE,
  MAP_SHARED, vcpufd, 0);

bool run_again = true;
while(run_again) {
  int err = ioctl(vcpufd, KVM_RUN, 0);
  if(err && errno == EINTR) {
    continue;
  }
  assert(err == 0);

  switch(vcpu_run->exit_reason) {
    case KVM_EXIT_HYPERCALL:
      handle_vm_hypercall();
      break;
    case KVM_EXIT_DEBUG:
      dump_vm();
      break;
    case KVM_EXIT_UNKNOWN:
      /* fall-through */
    case KVM_EXIT_SHUTDOWN:
      run_again = false;
      break;
    default:
      run_again = false;
  }
}
```

Listing 2.4: Running a VCPU

3 Design

My main goal is to design and implement a software to detect soft errors, originating from transient hardware faults, during application run-time. Once an error has been detected it should also be possible to recover from the error and continue program execution without a failure. The runtime overhead in the common case - that is no soft errors - should be as small as possible.

Additionally, the following three features are requirements for this work:

1. It shall be possible to run binaries without access to their source code.
2. No hardware modifications shall be required.
3. The software shall support recovery from transient hardware errors.

The first requirement rules out compiler-based error-detection techniques like SWIFT [RCV+05], as explained in Chapter 2.

Requirement 2 enables the use of commercial-off-the-shelf hardware and directly rules out any hardware-based approaches. Hardware-based fault tolerance mechanisms are usually designed for special purposes, such as airplanes [Yeh96]. In particular, the x86 and ARM architectures, used in most desktop and laptop and mobile systems, have no hardware-based tolerance against transient faults [man13, man11, Sea00].

For detection of transient hardware errors I rely on process replication, similar to PLR [SBM+09], explained in Chapter 2, because replicas yield only a small amount of benign faults and should perform reasonably fast on modern multi-core CPUs. Given at least three replicas, this technique allows for fault recovery by majority vote - which meets requirement 3.

Replication-based fault tolerance adds three new requirements:

4. The Sphere of Replication [RM00] shall be optimal with regard to the protection of the system, while adding as little runtime overhead as possible.
5. Replicas must be isolated from each other.
6. In order to keep the replicas comparable, they need to behave deterministically.

In case of an error in one of the replicas, their behavior is undefined. Requirement 5 ensures, that it is not possible for one replica to overwrite the state of another replica or the fault tolerance software itself.

Requirement 6 can be achieved by making sure the replicas get the same inputs at all times. Inputs to a program mainly arrive as results of system calls or as signals. Additional methods include shared memory and the `rdtsc` instruction.

The fault tolerance software needs a way to intercept the system calls made by each replica. The system calls can then be modified and the results copied, such that all replicas get the same results. Upon system call entry, all replicas are supposed to be in the same state, therefore system calls can be used for comparing the replicas.

Linux offers the `ptrace()` system call, which is used by one process to observe the behavior of another process. With `ptrace()` the tracing process can inspect registers and memory of the traced process. `ptrace()` "is primarily used to implement breakpoint debugging and system call tracing" [manc].

The dynamic linker can be used to load user-specified shared libraries with the `LD_PRELOAD` environment variable. Usually this is "used to selectively override functions in other shared libraries" [manb].

Operating-system level virtualization, provided by the `jail()` system call [KW00] in FreeBSD, or OpenVZ [Kol06] on Linux, provides isolation between different parts of a system. Multiple applications and a file system run in separate containers, with no access to the rest of the system.

The Linux Kernel Virtual Machine (KVM), explained in Section 2.3, is a hypervisor, running in the Linux kernel. Together with a suitable virtual machine monitor (VMM) KVM can be used to create a paravirtualization-based solution.

Requirements 1 and 2 can be addressed with the help of `ptrace()` [manc] or with `LD_PRELOAD` [manb]. The different levels of virtualization can provide for requirements 1, 2 and 5.

In the following sections I investigate which of these tools best meets requirements 4-6 and present ELKVM, a library to run unmodified ELF binaries inside a virtual machine without a full-scale operating system. The final section discusses ELK Herder, my fault tolerance software, built on top of ELKVM.

3.1 Fault Detection and Recovery

In order to detect faults reliably it is important to compare the replicas' states regularly. Unfortunately each comparison takes time and thereby introduces runtime overhead. Additionally, the replicas can only be compared, when they are supposed to be in the same state. A replica's state is defined by its register and memory contents.

At system calls the replicas' states are supposed to be the same, therefore system calls provide an inherent comparison point. A `ReplicaManager` compares the replicas, therefore the `ReplicaManager` needs access to the replicas' registers and their memory. Consider the design depicted in Figure 3.1. The `ReplicaManager` runs between the replicas and the Linux kernel and intercepts the replicas' system calls and compares the replicas' states at each system call.

The `ReplicaManager` starts the replicas when execution begins. Once all replicas started running, the `ReplicaManager` waits for the replicas to perform system calls. Only one replica is allowed to actually perform the system call, for all other replicas the `ReplicaManager` copies the result of the system call after the system call has finished. Once this is done the replicas continue execution.

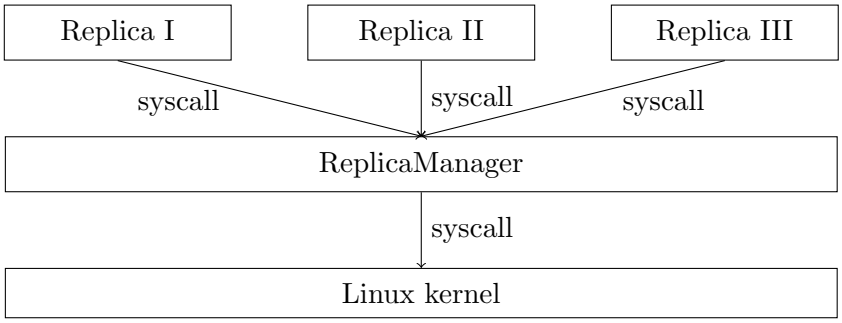


Figure 3.1: The ReplicaManager intercepts and forwards system calls

Recovery from transient hardware faults, which resulted in an error, can be achieved with a majority vote, because all correct replicas have the same state. All incorrect replicas can simply be overwritten with state from one of the correct replicas. Therefore read access to registers and memory suffices for fault detection, but recovery from faults needs write access.

The `ptrace()` system call provides options to read and modify the registers and memory of a traced process. With `LD_PRELOAD` memory access is not an issue, as long as the replicas run in the same address space. In KVM a user-space VMM maps all memory into the virtual machines, therefore this process has access to the replicas' memory. KVM also provides a call to read the registers of a virtual machine.

The `jail()` system call itself does not provide access to memory and registers, therefore `jail()` needs to be combined with other techniques, like `ptrace()`, for replica comparison.

3.2 Sphere of Replication

The decision on the size of the Sphere of Replication [RM00] affects, which parts of the system the fault tolerance software protects.

Protecting the operating system requires replication at a very low level. Replication at a low level makes it impossible to ignore many false DUEs[1]. Some benign faults can also only be recognized as such at higher levels. Therefore replication on a low level adds unnecessary overhead to the system, because the error recovery mechanism needs to be executed more often.

A large Sphere of Replication tends to introduce more overhead, because more code is executed with replication. On the other hand, less output comparisons are necessary, which allows to ignore benign faults and reduces overhead.

A large SoR decreases flexibility. For instance if the SoR includes the whole system, the OS and every application running on the system is replicated. While this will protect the whole system against hardware faults. It will also introduce the replication overhead for all applications running on the system.

[1] false detected unrecoverable errors, introduced in Chapter 2

Some applications may not need fault tolerance, think of an audio player or a CPU usage widget. The replication overhead for these applications can be avoided with a smaller, more flexible Sphere of Replication.

A small Sphere of Replication reduces resource overhead, especially memory usage, because less data needs to be replicated. However, a small Sphere of Replication also reduces error coverage.

Adding replication at the process level, allows for fine-grained control over which applications shall be replicated and allows the fault tolerance system to ignore false DUEs, thereby reducing overhead. For these reasons protecting the operating system kernel is not a design goal of ELK Herder and the Sphere of Replication can be placed around the replicated binary and its libraries. However if the need arises, the operating system can be protected with other fault tolerance mechanisms, such as ECC [MP86] or SWIFT [RCV+05]. Another possibilty is to replicate the operating system kernel itself.

With `ptrace()` and KVM the Sphere of Replication comprises of the whole process and its libraries, while with `LD_PRELOAD` it contains only a part of the process. The C library itself is not part of the Sphere of Replication with the `LD_PRELOAD` method, leaving a large part of each replicated application unprotected. For `jail()`, the Sphere of Replication comprises of multiple applications and their libraries.

3.3 Isolation of Replicas

The `ptrace()` system call uses standard Linux processes, which means the replicas cannot access each other's memory or memory in the fault tolerance software itself. However, memory access by the `ReplicaManager` in the replicas is slow.

For `LD_PRELOAD` it is possible to use processes as well by `fork()`ing the application. However this method complicates the communication to the `ReplicaManager` and eliminates one benefit against the `ptrace()` method. On the other hand, if `LD_PRELOAD` is used with threads, instead of processes, `LD_PRELOAD` provides no isolation between the replicas.

In the KVM case, each replica runs in its own virtual machine, which use virtual memory to provide isolation between the replicas. The `ReplicaManager` maps memory regions to the replicas. Figure 3.2 shows the situation if each replica has two memory regions mapped to it. Only these memory regions are accessible by the replicas. The `ReplicaManager` runs in a host address space and has access to the memory of the virtual machines to make memory management easier.

3.4 Input Interception

One of the design goals is minimizing runtime overhead, therefore execution of the replicas shall be done in parallel as long as the underlying hardware and operating system support this. The `ReplicaManager`, which intercepts inputs and compares replicas runs single-threaded. Therefore input interception and replica comparison should be as fast as possible.

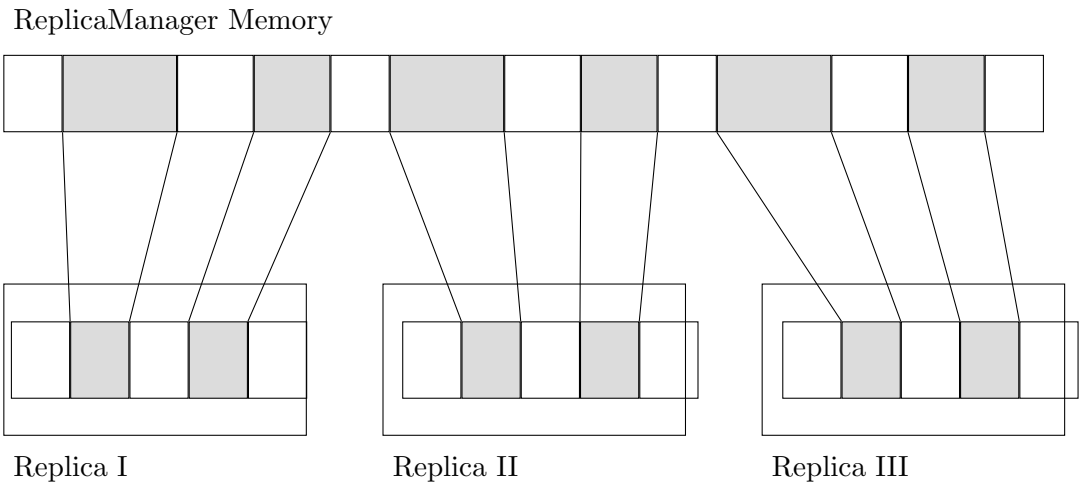


Figure 3.2: Virtual memory mappings inside the replicas provide isolation

```
1 entry:
2   vmcall
3   sysretq
```

Listing 3.1: Calling the host from a guest syscall handler

Input interception with `LD_PRELOAD` can be done at any API level, as long as the binary is dynamically linked. For instance if the chosen level is the C library, input can be intercepted by redefining all system calls in a wrapper library.

The `jail()` system call is not meant for system call interception, but `jail()` could be patched to do this. This patch needs to be applied in the kernel directly.

Table 3.1 summarizes the situation. So far, `jail()` cannot be used stand-alone and requires patches to kernel code. For `LD_PRELOAD` a huge isolation problem is foreseeable, as described in Section 3.3. These two methods are therefore unsuitable. The remaining methods are the `ptrace()` system call and KVM.

Modern x86 CPUs with virtualization support, ship with the `vmcall` [man13] or `vmmcall` [man11] instructions, which provide a way for a guest to make a call to the host. Instead of loading a fully-featured operating system into the guest, it is possible to load a short system call handler, shown in Listing 3.1, that calls the hypervisor inside the host for each system call or interrupt.

The hypervisor running in the host can then inspect the guest's registers and memory and handle the system call, for instance by forwarding it to the host's operating system. When the system call returns, the host writes the result back into the replica und enters the virtual machine again.

The `ptrace()` system call has the function to "observe and control the execution of another process and examine and change the tracee's memory and registers." [manc]. Therefore one could argue, that `ptrace()` is the obvious choice for the task.

```
1    uint64_t cycles[iterations];
2    struct stat buf;
3    int res;
4    for(int i = 0; i < iterations; i++) {
5      cycles[i] = RDTSC();
6      res = fstat(fd, &buf);
7      assert(res == 0);
8    }
```

Listing 3.2: This micro-benchmark measures the overhead of intercepting the fstat() system call

As I described above, system calls are in the critical path of each replicated application and the system call interception method should introduce minimal run-time overhead. Table 3.1 shows that `ptrace()` and KVM offer the same features with regard to all other requirements, therefore the speed of system call interception is critical to the decision between `ptrace()` and KVM.

In order to measure the run-time overhead of system call interception I implemented micro-benchmarks for the `brk()`, `fstat()`, `gettimeofday()`, `mmap()`, `read()` and `write()` system calls, as well as the C library's `malloc()` function. Listing 3.2 shows the micro-benchmark for the `fstat()` system call exemplary. The micro-benchmark executes each system call $10,000$ times in a loop, with the CPUs time-stamp counters read at the beginning of each iteration of the loop.

To measure the overhead of the `ptrace()` system call I wrote a small prototype, that intercepts each system call of the micro-benchmark, but does nothing apart from the interception.

Measuring the overhead introduced by KVM, required more work. I wrote a library, which loads an ELF binary - in this case the micro-benchmark - and the system call handler shown in Listing 3.1 into a KVM virtual machine. This library is called ELKVM and I describe its design in detail in Section 3.5.

My measurements show, that `ptrace()` introduces high overhead for system call interception. Figure 3.3 shows the cycle counts for `fstat()`, `gettimeofday()`, `read()`, `write()`, `mmap()` and `brk()`, when executed natively, with system call interception with ELKVM and when executed with system call interception with `ptrace()`. The x-axis shows the different system calls, while the y-axis shows the number of cycles divided by $1,000$.

Intercepting the `fstat()` system call with ELKVM results in ten times as many cycles for the whole call, intercepting the same system call with `ptrace()` takes 100 times as many cycles.

Figure 3.3 shows that the ELKVM overhead remains at $7,000$ to $10,000$ cycles for the `fstat()`, `gettimeofday()`, `read()` and `write()` system calls. The overhead for the `mmap()` and `brk()` system calls is about $70,000$ cycles for ELKVM and `ptrace()` respectively.

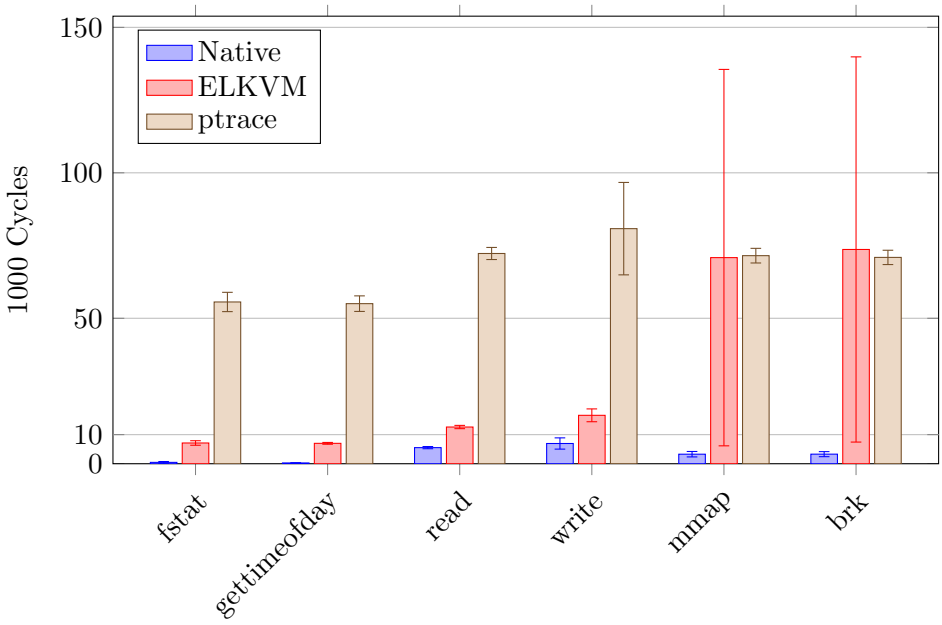


Figure 3.3: ELKVM adds overhead to various benchmarks

Due to the fact that memory mappings in KVM cannot be resized[2] in the worst case ELKVM needs to obtain a large new memory mapping from the Linux kernel, therefore ELKVM's maximum for the `brk()` system call, measured with the micro-benchmark, is at 3,671,112 cycles. In the best case, that is no new memory mapping is needed, ELKVM uses 15,560 cycles for the `brk()` system call. In the `ptrace()` case, the minmum and maximum are much closer, with 63,698 and 129,921 cycles respectively. The `mmap()` and `brk()` system calls are the system calls with the most overhead induced by ELKVM.

Requirement	ptrace	LD_PRELOAD	KVM	jail
Binary only	✓	✓ if dynamically linked	✓	✓
COTS Hardware	✓	✓	✓	✓
Majority voting	✓	✓	✓	X
Isolation	✓	X	✓	✓
Input interception	slow	✓ at any API level	fast	difficult
Sphere of Rep.	process	part of process	process / VM	mult. processes

Table 3.1: How various approaches address the requirements

[2] see Section 2.3

In conclusion these numbers, and the facts summarized in Table 3.1, motivate my decision to use the KVM method for the rest of this work. Unfortunately some work needs to be done before it is possible to run normal processes inside a virtual machine without a full-scale operating system.

3.5 ELKVM

ELKVM is a library that abstracts the KVM interface and allows execution of an ELF binary inside a virtual machine without a full-scale operating system. ELKVM initializes the virtual machine architecture to match the host machine and intercepts system calls. The library provides an API to handle intercepted system calls. This API is designed to match the Linux system call API as closely as possible.

Unfortunately the x86 architecture cannot run without an operating system at all. The architecture requires an Interrupt Descriptor Table to provide entry points for hardware and software interrupts and a Global Descriptor Table provides information about memory segments [man13]. Whenever a `syscall` instruction is executed, the hardware loads a pre-defined memory segment and jumps to a pre-defined address, the system call handler. In long mode, the architecture also mandates the use of virtual memory, also called paging [man11]. The operating system is responsible for setting up the required page tables.

In the host user-space ELKVM sets up a virtual machine, including a virtual CPU and initial memory. For the guest ELKVM works as a minimal proxy OS, replacing the operating system inside the guest. The proxy OS forwards all system calls to the host via hypercall.

Figure 3.4 shows the different parts involved in virtualizing an application with ELKVM. The ELKVM library loads the proxy OS and the guest binary into the virtual machine. The library uses KVM as a hypervisor, but abstracts from the KVM interface. The monitor is an application that implements the ELKVM API and handles system calls for the guest binary.

Before execution can begin, ELKVM loads the ELF binary into the virtual machine and sets up the environment pointer and the stack.

3.5.1 ELF Loading

Unix and Unix-like systems use the ,,Executable and Linkable Format" (ELF) [rep97] as the standard format for binary files. An ELF binary consists of an executable's text and data sections. Each section has a header, that contains information about virtual memory mappings for this section. The ELF header also contains information about segments, debug information and the program's entry address.

3.5.2 Resource Management

The x86 architecture requires applications to use virtual memory in long mode [man13, man11]. Therefore ELKVM needs to manage virtual memory for its guests. The amount of memory the guest binary requires during execution cannot be determined in advance,

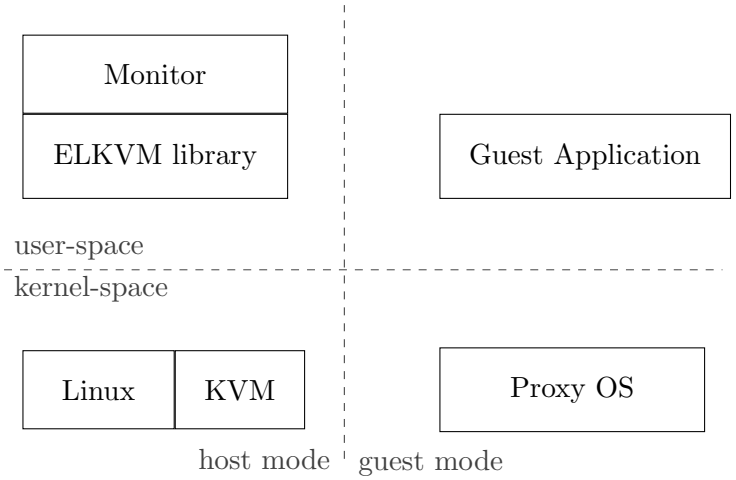


Figure 3.4: Units involved, when running an application with ELKVM

because of dynamic memory allocations. KVM does not allow memory mappings to be resized. Therefore ELKVM allocates more memory than necessary for the virtual machine and manages this memory with a buddy allocator [Kno65, Knu73].

A buddy allocator splits memory into two equally sized regions until a further split would make the region smaller than the memory needed by the binary. One of the regions is now marked as used and a pointer to this region is returned to the binary. The other region is kept as free memory. If the binary needs more memory, the other region is split to a fitting size. With the buddy allocator ELKVM manages a tree of regions with variable size.

Consider the example in Figure 3.5. A guest binary requests 8 MiB of memory, with the `mmap()` system call and there is no suitable memory region available. ELKVM allocates 32 MiB of memory inside the host and maps the whole memory region to the KVM guest with a call to `KVM_SET_USER_MEMORY_REGION`. The memory allocator splits the memory region in two adjacent 16 MiB regions. One of the two regions is split into two additional 8 MiB regions. One of those two regions is now marked as used. The resulting tree structure is searched for each consecutive request. For instance, if the guest binary requests additional 16 MiB of memory, the allocator searches the tree and finds the right region from the first split as free.

The smallest supported size is the page size. ELKVM only requests additional memory from the host Linux kernel, if no suitable memory region is found. If the amount of memory requested by the guest is larger than ELKVM's chunk size, ELKVM requests a larger chunk instead.

Memory that has been mapped to KVM, but is not used by the guest binary is not available to the guest binary, because there are no page table entries for this memory. If the guest binary requests more memory, a free memory region inside existing mappings is searched. If a suitable region is found, ELKVM creates the page table entries inside the guest without requesting more memory from the host Linux kernel.

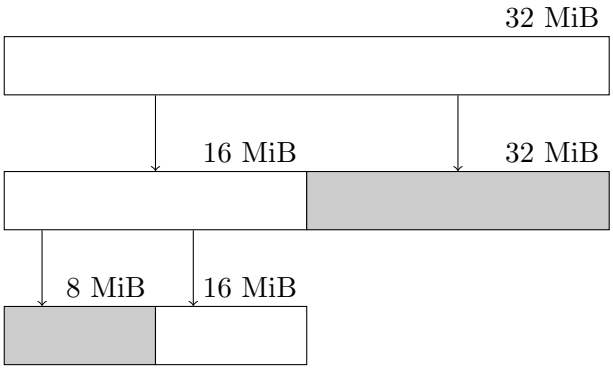


Figure 3.5: The Buddy allocator used to manage memory in the virtual machines

ELKVM keeps track of the memory mapped to the guest, by saving the host virtual address, the guest virtual address and the mapping's size for each mapping.

The `brk()` and `mmap()` system calls are used for memory management under Linux and require special handling by ELKVM. The `brk()` system call extends or shrinks the heap. A call to `brk()` cannot simply be forwarded to the host Linux kernel, because that would modify ELKVM's own heap size. Therefore ELKVM handles the `brk()` system call, by keeping track of the guest's heap size. The memory needed for the heap is managed by the buddy allocator described above.

Memory mappings created by the `mmap()` system call work in a similar way. The buddy allocator is used to find a suitable region for the mapping and the necessary page table entries are created eagerly. For file-based mappings ELKVM copies the file contents into the guest's memory from the specified offset.

ELKVM creates an `elkvm_mapping` object for each memory mapping created by `mmap()`. The `elkvm_mapping` object keeps a count of the mapped pages. This is necessary, because it is possible to partially unmap a mapping, by calling `munmap()` with a smaller size, than the preceding call to `mmap()`. ELKVM can only mark the connected memory region as free, if the original mapping is fully unmapped.

The files backing file-based mappings are updated on `munmap()` or by calling `msync()`. For a call to `msync()` ELKVM translates the buffer address from guest virtual to host virtual address and passes the call to the monitor application.

For the `mremap()` system call the buddy allocator needs to get a new region with the new size. If the neighboring memory region in the buddy allocator happens to be free and the two regions together are large enough, the two regions can simply be merged into one larger region. Otherwise ELKVM must copy the memory contents of the old region into a newly allocated region.

3.5.3 Intercepting System Calls

System calls provide a pre-defined entry point into the operating system kernel. For the x86-64 architecture, the Linux kernel defines more than 300 system calls, each identified

by an individual system call number. Up to six arguments for a system call are passed in the CPU registers. System calls with more arguments do not exist.

System call handling is illustrated in Figure 3.6. ELKVM loads the handler in Listing 3.1 into the virtual machine, and sets the CPU to jump to this handler on the `syscall` instruction.

System call interception is best described with an example. The guest executes the `fstat()` system call, which takes two arguments. The first argument is a file descriptor, the second argument is a pointer to a `struct stat` buffer. On execution of the system call, the operating system fills the buffer with meta-data from the file, represented by the file descriptor, and returns.

For an ELF binary running in ELKVM, the `syscall` instruction jumps to the proxy OS system call handler, which in turn executes the `vmcall` instruction. On execution of this instruction the physical CPU leaves guest mode and returns to the host hypervisor. The KVM module in the Linux kernel takes control and returns to the ELKVM library running in user-space.

The ELKVM library's system call handler function reads the system call arguments, including the system call number, from the registers, with `KVM_GET_REGS`. In the case of `fstat()` we now have three arguments, the system call number 5 in the `rax` register of the guest, the file descriptor in `rdi` and the pointer to the `stat` buffer in `rsi`.

The pointer is represented as the virtual guest address and is therefore invalid in the host. ELKVM uses the guest's page table to translate the pointer from guest virtual address to host virtual address, in order to make sure the correct buffers are read and written.

ELKVM now calls the `fstat()` handler function registered by the monitor. The monitor handles the system call, either by forwarding the call to the host operating system, or by fulfilling the request itself.

After the monitor finished handling the system call it returns the system call result to ELKVM. If the call was successful ELKVM writes the system call result into the `rax` register of the guest, otherwise ELKVM writes the error number, `errno`, into the same register.

System calls related to resource management are an exception to this behavior. Memory is managed in terms of KVM memory, for the reasons described in Section 3.5.2, therefore the ELKVM library emulates their behavior.

3.5.4 Signals

All system calls related to signals cannot simply be forwarded to the host operating system, because then the Linux Kernel would call the registered function in the ELKVM library, instead of the guest binary.

In order to forward signals to guest binaries ELKVM has its own signal handler, which is registered whenever a virtual machine registers a signal handler with the `sigaction()` system call. When a registered signal arrives, the ELKVM signal handler notes the arrival of the signal. ELKVM then waits for the next system call by the guest binary to deliver the signal. ELKVM calls the guest's signal handler after the system call. This

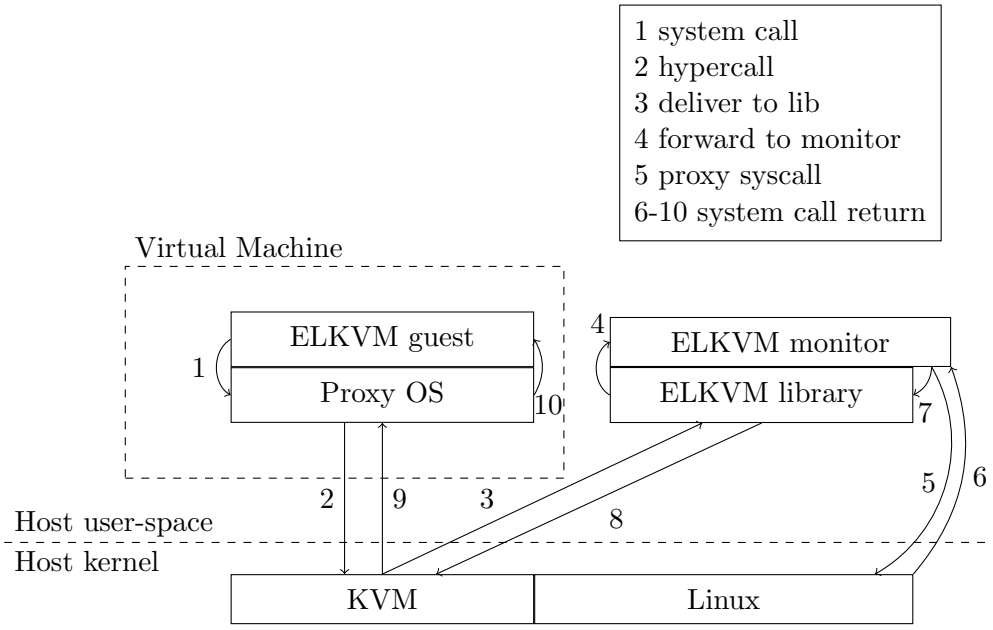


Figure 3.6: ELKVM forwards a system call to the monitor

way guest execution can continue after the signal handler, without an additional call to the host.

Injecting signals at system calls is necessary to keep replicas deterministic, which is a requirement for the `ReplicaManager` described in the beginning of this chapter. Unfortunately, waiting for a system call may introduce large delays to signal delivery. This may not be compatible with other uses of ELKVM, therefore it should be possible to configure signal delivery in future versions of ELKVM.

3.5.5 Currently Unsupported System Calls

In Section 3.5.3 I explained the large number of system calls in the Linux kernel. Because of amount of system calls I had to focus on implementing the system calls necessary to run the SPEC CPU 2006 benchmarks, used for the evaluation. In this section I give an overview on ideas to implement some of the system calls, that are not yet supported in ELKVM.

The `fork()` system call is used to copy an address space on Linux, for ELKVM this requires a copy of the whole VM, including the CPU state and the VM's memory. None of the benchmarks used to evaluate ELKVM and ELK Herder make use of the `fork()` system call and therefore an implementation for `fork()` has not been a priority. Similarly `execve()` requires ELKVM to load a new binary into the virtual machine and adjust the VM's memory mappings accordingly. `execve()` is also not supported at the moment.

`clone()` is very similar to the `fork()` system call, but "allows the child process to share parts of its execution context with the calling process" [mana]. This behavior can

be implemented by creating a new virtual machine in the same address space as the calling process, but with its own execution thread. The ELKVM library inside the host has to manage the parts, which are to be shared by the processes.

The `ioctl()` system call consists of three parameters, the first one is a file descriptor, the second one is an integer, that identifies an operation. This operation varies between different targets of the `ioctl()`, which cannot be identified by the file descriptor. The type of the third parameter depends on the value of the second parameter and on the target of the call. While it could be anything, in most cases it is either an integer or a pointer. There is no way for ELKVM to know about all possible `ioctl()` combinations.

A possible, albeit not very nice solution to this problem is to keep a configuration file, from which ELKVM reads the operation / parameter combination needed for certain `ioctl()`s. It is also be possible to keep the information in a header file, requiring a recompilation of ELKVM whenever the `ioctl()` interface changes. The performance and memory impacts of the two possibilities need to be explored accurately to make a well-founded decision. The SPEC CPU 2006 benchmarks used to evaluate ELKVM do not use the `ioctl()` system call, therefore the implementation of support for `ioctl()` is left as further work.

3.6 ELK Herder

ELK Herder is a `ReplicaManager` running on Linux. ELK Herder uses ELKVM to run unmodified ELF binaries inside a virtual machine and intercept their system calls. This way, each of the virtual machines becomes a replica. The replicas run independently from each other as guests in user mode, until they execute a system call or interrupt.

3.6.1 System Calls

Once a replica executes a system call, ELKVM translates guest addresses to host addresses and provides up-to-date register information for ELK Herder. ELK Herder first collects the system call's arguments and calculates a checksum over the replica's registers. The first replica to arrive creates a shared system call object that saves the state of each replica, the arguments to the system call and the call's result.

Until at least half of the replicas have arrived, each replica waits for the other replicas to arrive. After more than half of the replicas reported the same state, the last replica[3] to report this state executes the system call on behalf of all other replicas. The replica, which executed the system call, becomes the reference replica. All replicas that arrive after the reference replica must wait for the reference replica to finish the system call. When all replicas arrived and the system call has finished, the `ReplicaManager` is called and reinitializes all replicas which reported a false state.

3.6.2 Replica Comparison & Reinitialization

The `ReplicaManager` compares each replica's checksum to the reference replica's checksum. If the checksums match, the `ReplicaManager` copies the result of the system call

[3]For n replicas, that is replica $\lfloor (n/2) \rfloor + 1$

into the replica. If a replica has a different checksum, this replica had a hardware fault. For a transient hardware fault, the replica can be reinitialized and continue execution.

For reinitialization, the reference replica's virtual address space and virtual CPU registers are copied into a new replica and the old replica is stopped. Figure 3.7 illustrates the steps involved in reinitializing a replica.

First the `ReplicaManager` copies the register contents of the virtual CPU. Afterwards the `ReplicaManager` copies all memory regions, because their contents may be affected by the fault.

When this procedure is finished, the replica is in the same state as all others right before the replicas executed the system call. At this point the new replica is allowed to run, which means it executes ELKVM's system call handler and ends up in exactly the same state as all other replicas, unless there is a new error. In this unlikely case the procedure is repeated until the checksums match.

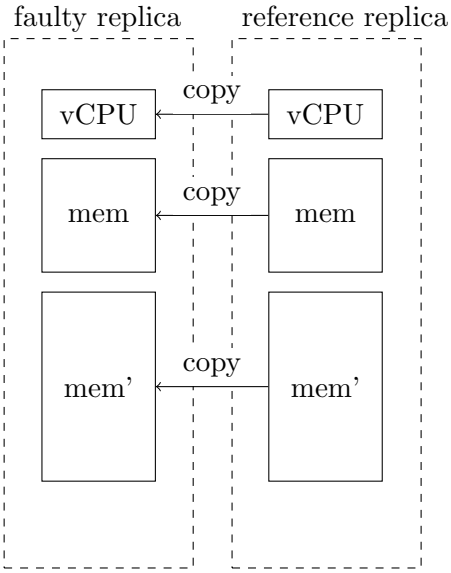


Figure 3.7: Restart of a faulty replica

The `ReplicaManager` compares each reinitialized replica to the reference replica again, because the fault may still be present. If the states of the reinitialized replicas and the reference replica match, the replicas are allowed to resume.

If the reinitialized replica experienced the same fault again, the fault is likely permanent. In this case, the hardware component needs to be marked as faulty and binaries shall be kept away from the component by ELK Herder.

Compute-bound applications tend to do few system calls. This may result in the time between two system calls becoming more than twice as long as the mean-time between faults. Longer execution time between two system calls, increases the possibility of than two transient hardware faults arising between two system calls. Therefore it may be necessary to compare the replicas in between system calls for compute-bound applications, to make sure less than half of the replicas experience transient hardware faults in

between two comparisons. Kriegel [Kri13] described an instruction based watchdog to solve this problem. I plan to use the results from Kriegel's work in a future version of ELK Herder.

3.6.3 Signals & Other Sources of Input

When the replicas finish the execution of a system call, ELK Herder copies the system call result from the reference replica to the other replicas. If the ELK Herder process received any signals since the last system call, ELKVM injects these signals into each of the replicas after the results have been copied. This way ELK Herder can be certain that each replica receives the signal at exactly the same instruction - i.e. the `vmcall` instruction in ELKVM's system call handler. Signal injection itself is not modified from the raw ELKVM case.

Unfortunately compute-bound applications do system calls sparsely. In order to deliver systems timely it may be necessary to inject signals between system calls. Instruction-based watchdogs, described by Kriegel [Kri13], may provide more frequent opportunities for signal injections in those kinds of applications.

The `rdtsc` instruction is an additional source of input into the replicas. ELK Herder can intercept this instruction easily by setting the Time Stamp Disable bit in the `CR4` register of each virtual CPU in each of the replicas. This way the guests receive a general protection fault on execution of the `rdtsc` instruction. ELK Herder can handle this fault the same way system calls are handled.

Shared memory occupies an important role in multi-threaded applications. Although the main goal for this work is single-threaded applications, multi-threading is a feature, which I intend to add in the future.

Mushtaq et al. [MAAB12] describe a method to support shared memory in replicated applications. Their basic idea is to map memory read-only at first, to detect dirtied pages. This idea can be adapted easily by ELK Herder, by mapping the same memory region to all replicas read-only. This way a write to the memory produces a page fault, which triggers the replica comparison mechanism. When a majority of replicas arrived at that page fault the write can be executed.

In a follow-up publication [MAAB13] Mushtaq et al. propose a version of their algorithm, which does not need read-only mappings to make sure shared memory accesses are deterministic. This version uses clocks inside a mutex to make sure memory accesses are performed in the same order. However this algorithm is not secure against transient hardware errors flipping a bit in the clock data structure.

3.6.4 Error Handling

Comparing the replicas with the methodology described above only works if two assumptions hold:

1. All replicas must arrive in the system call handler, which means they must execute *any* system call in their lifetime.

2. More than half of the replicas must execute the same system call with the same arguments.

ELK Herder is only able to determine a clear, absolute majority of the replicas' states if the second assumption holds. If the `ReplicaManager` cannot determine an absolute majority ELK Herder raises an error and stops execution. This way ELK Herder provides safety against silent data corruptions. In the future this can be extended by keeping checkpoints of the replicas and reverting to known correct state if there is no absolute majority of replica states.

If a replica is hit by a transient hardware error, the behavior of that replica is undefined. This means it might stop issuing system calls altogether. Therefore the first assumption does not always hold. Similar to the method described by Shye et al. [SBM+09] and by Kriegel [Kri13] the `ReplicaManager` starts a watchdog timer after the first replica issues a system call.

If the watchdog timer runs out for a system call, and a majority of replicas arrived at the call, before the timer ran out, all remaining replicas are reinitialized with data from the reference replica. If only a minority of replicas arrived at the system call when the timer runs out, these replicas are killed and reinitialized at the next system call.

If ELK Herder is running with three replicas, then one replica is the largest minority, that can execute a false system call. This replica is killed, leaving two replicas running. If there is another transient fault, ELK Herder is not able to recover from this fault. At the next system call, the replica is reinitialized and recovery is provided again. The same principle applies for higher replica counts as well. Detection of silent data corruptions always stays intact, because at least two replicas always continue execution.

The problem is the same, if a replica is hit by a transient hardware fault and subsequently enters an endless loop. When running with triple-modular-redundancy, only two replicas remain in the correct state, which means an additional transient hardware fault in one of the correct replicas can only be detected, but not recovered from. As a consequence, killing replicas, that execute a false system call does not diminish ELK Herder's fault tolerance.

3.6.5 Memory

All data that enters the Sphere of Replication must be replicated for error detection to work correctly. If the fault tolerance mechanism keeps only one copy of specific data inside the SoR, a transient fault may affect this data and cause the error detection to fail.

Data can enter the Sphere of Replication inside memory mapped to an application with the `mmap()` system call. Consequently this data must be replicated by the `ReplicaManager`, before it is mapped into each replica.

For read-only memory the copy could be avoided with error-checking codes for the mapped memory. However only small parts of an application are actually mapped read-only, compared to the total memory used by an application. For typical applications, the text section is mapped read only and a small part of the data section. Dynamic memory mappings are almost never read-only. Therefore the implementation overhead for such a scheme is not justified, because the performance benefit is small.

Addresses of memory mappings are also an input into an application. When the replicas' states are compared, these addresses are part of the state and must therefore be the same for all replicas. As a consequence, all replicas must receive their memory mappings at the same virtual (guest) address.

The host address for the mappings must necessarily differ, because each replica has an exclusive copy of the mapping. Each of these copies must exist in a different host memory location. Therefore the mappings from guest virtual address to host virtual address are different for each replica, as shown in Figure 3.8.

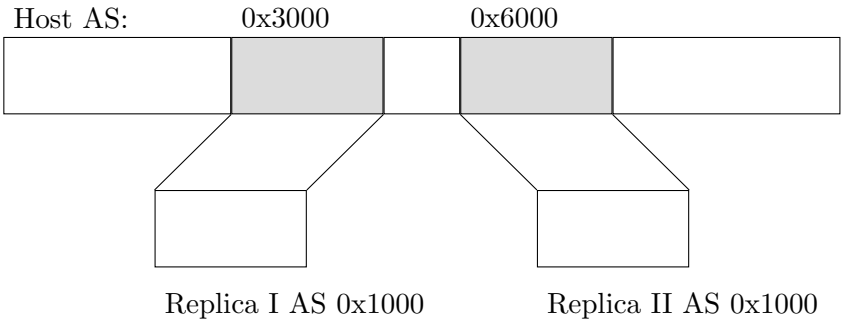


Figure 3.8: Data replication for mmap

ELKVM needs the information about this mapping from ELK Herder. I solved this problem by adding an additional parameter to `mmap()` in ELKVM's API. This parameter contains the guest virtual address, the host virtual address and the size of the mapping.

4 Implementation

In the following sections I present details on the implementation of ELKVM and ELK Herder. ELKVM is implemented as a dynamically-linkable library in C, while ELK Herder is implemented in C++.

Although I implemented ELKVM as a general-purpose library to run ELF binaries with a minimal proxy operating system in KVM, some design and implementation decisions were taken with the ELK Herder use case in mind.

4.1 ELKVM

ELKVM abstracts from the KVM API with the help of the data structures found in Figure 4.1. The `elkvm_vm` structure manages a virtual machine. It provides the `elkvm_vm_create()` method that creates and initializes a virtual machine.

Each virtual machine has a list of virtual CPUs, abstracted by the `elkvm_vcpu` structure. An `elkvm_vcpu` consists of the `kvm_regs` and `kvm_sregs` structures provided by KVM, the file descriptor to the VCPU and the `kvm_run` structure, that KVM uses to provide information about the result of the last guest run.

An ELKVM monitor application must implement the system call handler functions, provided in a structure of function pointers. As Listing 4.1 shows, I kept the function signatures close to the Linux API. A function pointer is provided for each system call, that exists in the x86-64 Linux kernel.

The `elkvm_pager` structure handles all virtual memory inside the virtual machine. It is responsible for page table setup, address translation and the heap size. The `elkvm_pager` abstracts memory mappings to KVM and keeps track of the total memory size mapped to the virtual machine.

```
1  struct elkvm_handlers {
2    long (*read) (int fd, void *buf, size_t count);
3    long (*write) (int fd, void *buf, size_t count);
4    long (*open) (const char *pathname, int flags, mode_t mode);
5    long (*close) (int fd);
6    long (*stat) (const char *path, struct stat *buf);
7    long (*fstat) (int fd, struct stat *buf);
8    /* ... */
9    long (*mmap) (void *addr, size_t length, int prot, int flags, int fd,
       off_t offset, struct region_mapping *);
10   /* ... */
11 }
```

Listing 4.1: The ELKVM system call handlers

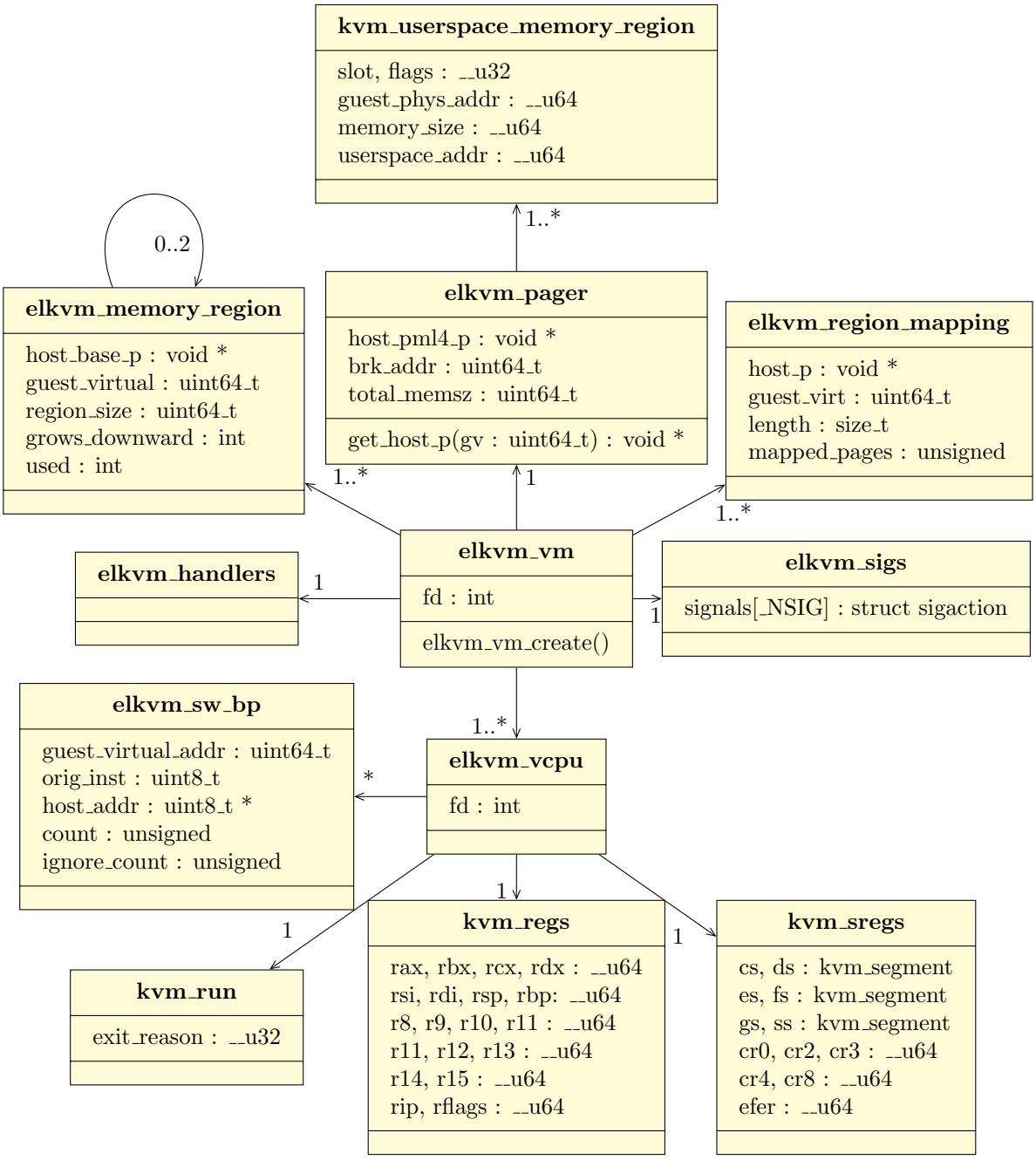


Figure 4.1: Components of ELKVM

4.1.1 Memory Management

In Section 2.3 I explained, that KVM only allows a user-space binary to map fixed-size memory regions. A guest binary can request arbitrarily sized memory mappings with the `mmap()` or `brk()` system calls dynamically. In order to allow these dynamic memory mappings, in spite of the fixed-size limitation, ELKVM implements an additional memory manager. As described in Section 3.5.2, the ELKVM memory manager uses a buddy allocator [Kno65, Knu73] to map smaller parts of these regions into the virtual machine's page table. The `elkvm_memory_region` structure represents the tree structure underneath the memory allocator.

The `munmap()` system call allows parts of a memory mapping created by `mmap()` to be unmapped. ELKVM can only mark an `elkvm_memory_region` as free, once the whole mapping is unmapped. The `elkvm_region_mapping` structure keeps track of the number of mapped pages for each mapping as well as the location and size of each mapping. This structure is also used to communicate information about mappings between the monitor application and the ELKVM library.

The proxy OS requires initial memory for the guest binary's text and data sections, the initial page tables and the minimal operating system structures described in Section 3.5. ELKVM's virtual memory layout is the same layout as on Linux, with the lower addresses for user-space memory, including the heap and the stack, and the top 64 GiB of the virtual address space used for kernel memory. For system call and interrupt handling ELKVM installs short assembly routines inside the guest's kernel memory region.

While the buddy allocator, described in Chapter 3, is reasonably fast and simple in its design, there is one important drawback. If the guest application regularly decreases and increases its heap size, the heap might not be contiguous in the host's memory any more. Therefore ELKVM needs to check large buffers received from the guest for contiguity in the host address space. The problem is depicted in Figure 4.2. If the buffer received, e.g. for a `read()` is not contiguous, ELKVM splits that system call into multiple calls to ensure that memory is not corrupted. This only affects `read()`s with buffers, which overlap the border of two `elkvm_memory_region`s.

My profiling showed, that most of the time spent in the ELKVM library, is used for address translation. Every buffer handed to ELKVM by a system call has to be translated from guest virtual address to host virtual address by ELKVM. The same is true every time the library accesses or modifies the guest's stack.

In this translation, the host virtual address serves as the guest's physical address and the translation is done by walking the page table. Hereby every entry in the page table contains the guest's hardware address of the next table and part of the guest virtual address is used as an offset into that table.

I assumed, that implementing a cache to directly return the host virtual address would improve performance of these translations. Therefore I implemented a software translation-lookaside buffer (TLB). Unfortunately a soft-TLB did not bring any measurable performance improvement.

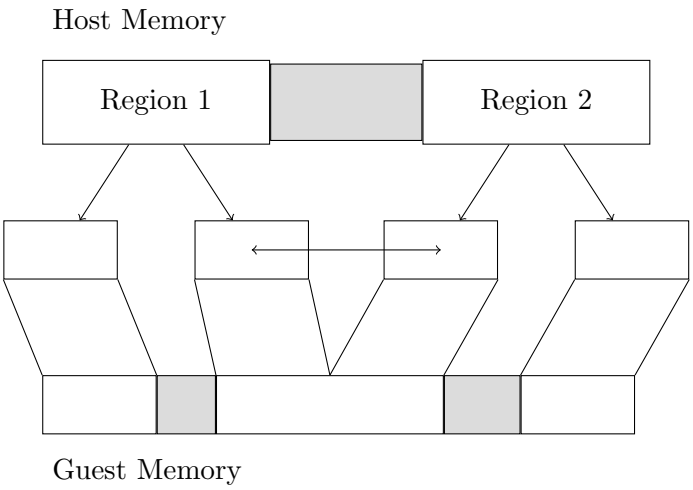


Figure 4.2: A non-contiguous guest buffer

In case of an interrupt the virtual CPU pushes further information about the interrupt onto the stack [man13, man11]. If the user-space stack were used for this push, entering the interrupt handler could result in a page fault, which at this point would mean a double fault. In order to prevent page faults, when entering interrupt handlers the proxy OS uses a kernel stack for interrupt handlers. A single page is always mapped for this stack.

4.1.2 System Call and Interrupt Handling

After a system call or interrupt the `vmcall` instruction is used to signal the hypervisor that an action by the host is necessary. The proxy OS pushes the hypercall type to the kernel stack, before executing the `vmcall` instruction. Inside the host the ELKVM library pops the type from the stack and handles the hypercall accordingly.

All further system call and interrupt handling is done inside the host. For most system calls, handling the call means getting the system call number and arguments from the corresponding registers, translating any pointers from guest virtual address to host virtual address and calling the appropriate handler function, defined by the monitor through the `elkvm_handlers` structure. Because all buffer addresses have been translated to host addresses, the monitor can write to buffers provided by the guest binary directly, without the need for ELKVM to copy any data. After the monitor has handled the system call, ELKVM writes the result into the `rax` register and returns control to the guest.

Usually ELKVM guests only get interrupts if the stack for the binary grows beyond the initial 2 MiB of memory mapped for it. In this case, the guest gets a page fault and the ELKVM library maps additional memory for the stack. Additionally ELK Herder replicas get interrupts, for instance, if a fault invalidates memory accesses. An ELKVM monitor application may register a handler function for interrupts. If no handler function

```
1   static int handle_vmcall(struct kvm_vcpu *vcpu)
2   {
3 -         skip_emulated_instruction(vcpu);
4 -         kvm_emulate_hypercall(vcpu);
5 -         return 1;
6 +         vcpu->run->exit_reason = KVM_EXIT_HYPERCALL;
7 +         return 0;
8   }
```

Listing 4.2: The patch to handle_vmcall

is registered, the ELKVM library stops execution of the guest binary, when the guest receives an any interrupt, that is not a stack expansion.

Vanilla KVM does not forward the `vmcall` instruction to clients, but rather handles it inside the Linux kernel and immediately returns to the guest. Therefore ELKVM depends on a patch in `arch/x86/kvm/vmx.c` to KVM's `handle_vmcall()` routine. The patch is shown in Listing 4.2.

AMD CPUs do not support the `vmcall` instruction but rather use the `vmmcall` instruction [man11]. The `vmmcall` instruction has a different opcode, than the `vmcall` instruction, but the same behavior. In order to run ELKVM on an AMD CPU one needs to change system call and instruction handlers to reflect that behavior. I did not test ELKVM on AMD CPUs, yet I expect ELKVM to run on any 64-bit x86 CPU capable of virtualization, in spite of this incompatibility.

4.1.3 Signals

System calls, which deal with signal handling are handled by ELKVM itself. ELKVM intercepts any signals for a guest binary and delivers these signals at system calls.

If a guest binary calls the `sigaction()` system call to register a signal handler, ELKVM saves the guest binary's signal handler address in its own `elkvm_sigs` data structures. The ELKVM library then registers a standard system call handler with the Linux kernel. This handler saves the signal for later delivery.

In Section 3.5.4 I mentioned, that it is possible to inject an interrupt into the guest in order to deliver the signal the moment it arrived. Currently ELKVM waits until the next system call made by the guest to inject the signal. Replicated guests need to receive the signal at the exact same instruction - and in loops in the same iteration of the loop - in order for inputs into the replica to remain deterministic.

Figure 4.3 shows a sequence diagram of the `open()` system call, with a signal injected after the system call. The binary's call to `open()` is redirected to the ELKVM library, which forwards the call to the Linux kernel. When the call returns, ELKVM checks for pending signals. If there are no pending signals, ELKVM resumes execution of the guest.

If a signal exists, ELKVM modifies the guest binary's stack to return to the pre-registered signal handler. The signal handler registered by the guest is a normal function,

therefore it might clobber any registers, including the `rax` register, containing the system call result. The signal handler itself is executed in guest user-mode, which means it is not an option to restore the system call result in ELKVM inside the host, because returning to ELKVM introduces the overhead of another hypercall. Additionally, the signal handler has to be modified to perform a hypercall once it finished. Instead, the ELKVM library modifies the guest binary's stack, in a way that the guest binary returns to a short system call restore routine, running in guest mode, after the signal handler.

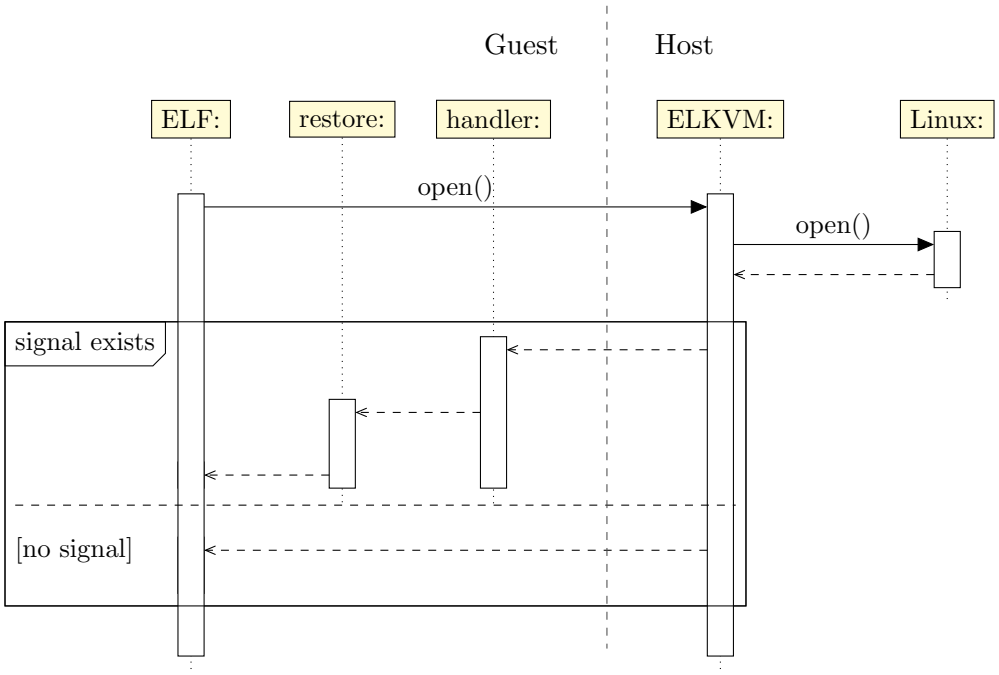


Figure 4.3: ELKVM injects signals into the VM

4.1.4 Debugging and Single-stepping

In order to allow debugging the ELKVM library, ELKVM guests and ELKVM monitor applications, ELKVM supports a debugging mode, which prints all system calls made by a guest application, as well as their arguments and the system call results to `stderr`. ELKVM also supports single-stepping the guest application and comes with a simple debugging shell, that allows to inspect the guest's registers, stack and the next instructions.

Breakpoints are helpful in debugging ELKVM, as well as guest binaries. Breakpoints can also be used to inject faults into a guest binary at a specific instruction. ELKVM abstracts breakpoints with the help of the `elkvm_sw_bp` structure found in Figure 4.1. Additionally ELKVM provides a breakpoint handler function in its API, that is called, whenever a breakpoint is hit. If the client did not set a breakpoint handler, ELKVM calls its own debug shell.

4.2 ELK Herder

ELK Herder is a replica-manager implemented in C++. The class diagram in Figure 4.4 provides an overview over the parts involved.

The `ReplicaManager` class is responsible for starting all replicas, and handles all replicas' system calls. The `ReplicaManager` also reinitializes faulty replicas.

The `Replica` class provides a common interface for the `ReplicaManager` to work with. For error injection and testing, `Replica` is a virtual class and all functionality is provided by the `ElkvmReplica` class. `ElkvmReplica`s use ELKVM to execute an ELF binary with replication. `ElkvmReplicas` provide handler functions for all system calls, which call the `ReplicaManager` for state comparison.

4.2.1 System Call Handling

The `SharedSyscall` class keeps track of the replicas that issued the current system call and the system call's state. A system call can be in the *pending* state, meaning that no clear majority has been determined and no action has been taken, yet.

Once a `SharedSyscall` object has established an absolute majority of replicas with the same state, one of the replicas issues the system call to the Linux kernel, making the system call enter the *called* state. When the replica returns from the system call, the call enters the *done* state. The `syscall_state_t` enumeration is used to name these states. The replica, which issued the system call becomes the reference replica.

In order to determine the state of a replica, a CRC checksum is calculated over the values of all registers of the replica's virtual CPU. The memory contents are only included in the checksum if the system call uses an output buffer. This is done for performance reasons and does not affect error coverage, because all data, that leaves the Sphere of Replication, is still compared. The calculation of the checksum is done hardware-supported with SSE4 instructions.

For ELK Herder handling a system call and obtaining the results is largely the same procedure for almost all of over 300 system calls existent in Linux 3.11 for the x86-64 architecture. In order to avert repeating the code for this procedure, all replicas prepare a `Syscall` object before registering for a system call. This object contains the system call number, the number of arguments for the system call and the size and location of each buffer, provided as an argument. The actual system call is not called through the C library wrappers but rather through the `syscall()` method. After the system call, each argument with a specified size is filled with the values from the reference replica's `Syscall` object.

4.2.2 Reinitialization of Replicas

If a replica has a different state than the reference replica, the `ReplicaManager` reinitializes this replica. For the reinitialization each memory chunk of that replica is overwritten with data from the reference replica. If the memory chunk differs in size, the `ReplicaManager` unmaps it and maps a new chunk, with the the correct size, into the replica. The `ReplicaManager` also overwrites all register contents of the reinitialized

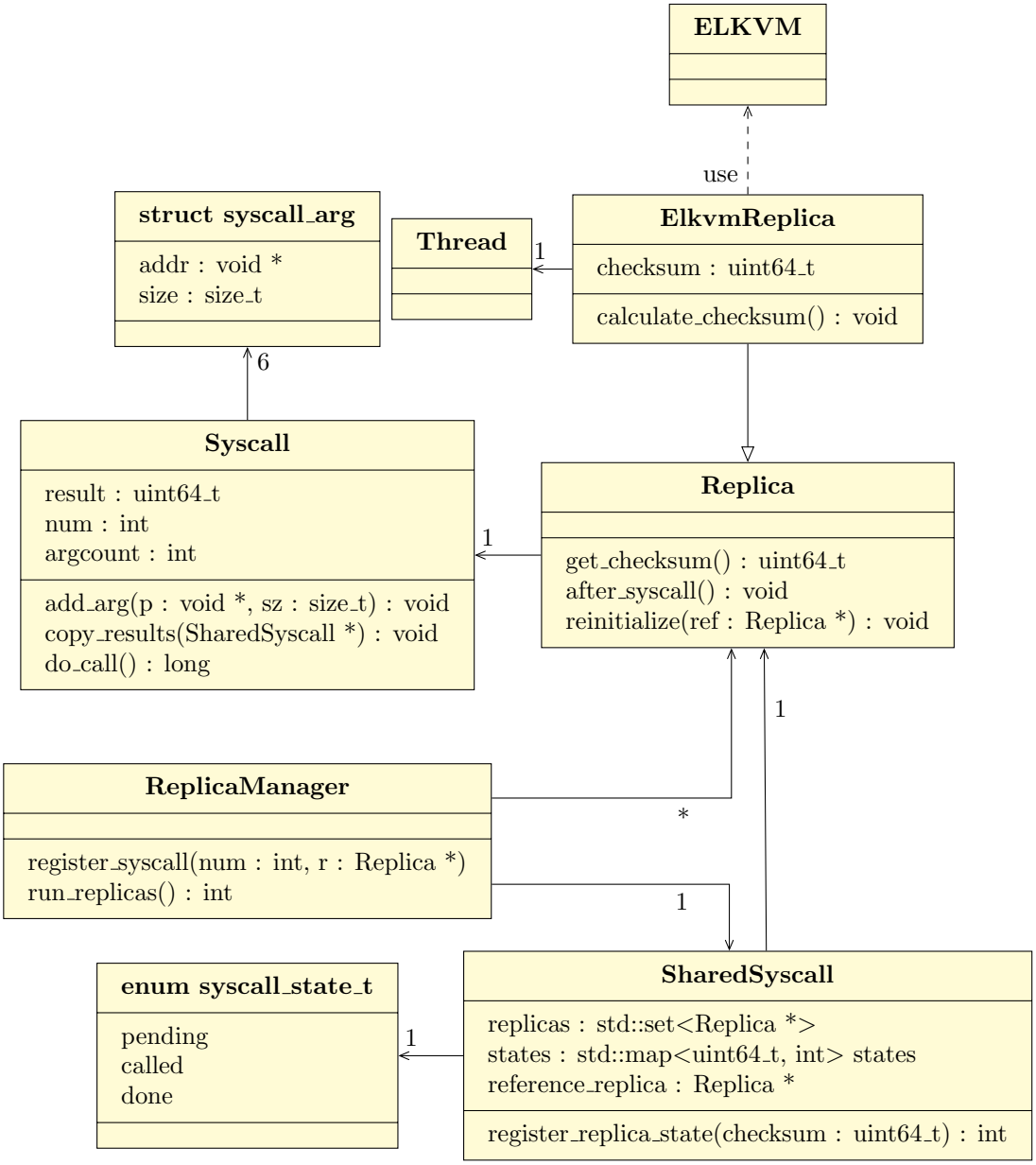


Figure 4.4: ELK Herder class diagram

replica with values from the reference replica. This way the reinitialized replica has the same state as the reference replica, before the reference replica executed the `vmcall` instruction.

For the reinitialization of the replica itself, the `ReplicaManager` creates a new `Thread` object and calls ELKVM to enter guest mode and execute the `vmcall` instruction.

When the new `Thread` enters the system call handler again it should have the same state as the reference replica. If this is not the case and the states differ again, this procedure is repeated until the states are the same. The old `Thread` executes the `exit()` system call in the host.

In case of a fault, the behavior of a replica is unknown. The replica may enter an endless loop and stop issuing system calls altogether. In order to handle these cases, ELK Herder starts a watchdog timer on the reference replica. When the timer fires, the `ReplicaManager` sends a `SIGUSR2` signal to all replicas that have not yet arrived at the system call. In Section 2.3 I explained, that an `ioctl()` with the virtual CPU file descriptor and `KVM_RUN` is used to execute the virtual machine. Delivery of the signal interrupts this `ioctl()` and thereby the virtual machine.

The drawback for this solution is, that ELK Herder does not support registration of the `SIGUSR2` signal any more. In my case, this is not an issue, because the SPEC benchmarks I use for the evaluation do not register this signal. For applications, which use the `SIGUSR2` signal for communication, another approach has to be found. An idea is to switch to any other unused signal in this case.

4.2.3 Fault Injection

For this work I want to be able to test the error coverage of ELK Herder. Therefore ELK Herder can inject faults into a replica. For this feature, the `ElkvmReplica` class is extended by an `ErrorInjectingReplica`, as shown in Figure 4.5. ELK Herder reads the address of the instruction at which the faults shall occur from a file. ELK Herder then registers a breakpoint handler and the breakpoint with ELKVM. In order to be able to inject breakpoints into exact iterations of loops, ELK Herder can ignore a pre-defined amount of hits for each single breakpoint.

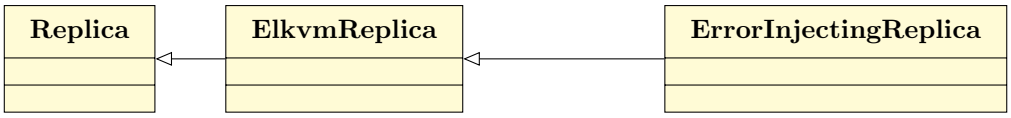


Figure 4.5: Error injection is done by expanding ElkvmReplicas

Once the replica arrives at the breakpoint, ELK Herder flips the bit specified in the file, and resumes the replica. If the fault manifests in an error it should be detected at the next system call.

5 Evaluation

ELK Herder and especially ELKVM have a noticeable implementation effort, as Table 5.1 shows. Together the ELKVM library and the ELK Herder replica manager require more than 7,500 lines of C and C++ code[1]. This Chapter evaluates benefits and drawbacks of using a minimal proxy OS inside virtual machines as replication method.

5.1 Implementation Effort

ELKVM totals in 6,702 lines of C code and 53 lines of Assembly code for the system call and interrupt handlers. Although I used ELKVM primarily for the purpose of replication in my work, it is possible to use ELKVM for any task, which requires the interception of system calls. My measurements in Chapter 3 show that ELKVM intercepts system calls an order of magnitude faster than the `ptrace()` system call.

	Component	LOC
ELKVM		6,702
	ELF Loader	294
	System & VCPU setup	1,045
	Memory mgmt	1,042
	VM mgmt	563
	Syscall handling & API	2,150
	Unit Tests	1,201
	Other	407
ELK Herder		1,050
	Replias	140
	ReplicaManager	138
	Syscall handling	439
	Thread handling	110
	Unit Tests	65
	Other	158
Total		7,752

Table 5.1: LOC spent for ELKVM and ELK Herder

[1] all LOC were generated using David A. Wheeler's 'SLOCCount'.

Table 5.1 summarizes the lines of code spent on the different modules of ELKVM and ELK Herder. The ELF Loader uses 300 lines of C code to parse an ELF header and load the binary into VM memory, as specified in the header.

Creation of the virtual machine, including global descriptor tables, interrupt descriptor tables and the virtual CPU takes 1,045 lines of code. 1,042 lines of code are spent on memory management, this includes the buddy allocator, as well as page table management and API calls to KVM. Together with the 50 lines of assembly needed for the minimal proxy OS inside the virtual machine, these 2,000 lines of code make up the machine dependent code in ELKVM. In order to port ELKVM to another hardware architecture, this code needs to be adapted.

Virtual machine management, for instance, getting and setting register contents uses slightly more than 500 lines of code.

With more than 2,000 lines system call handling and the definition of the API is the largest part of ELKVM. Most of this code is boiler-plate code, that initializes system call arguments. Unfortunately the number and type of the arguments vary between different system calls. Additionally, whenever the Linux API changes, this code needs to be updated to reflect those changes.

In ELK Herder, the `Replica` class takes 140 lines of C++ code. This code includes acquiring a lock for each system call, calculating the checksum and replica reinitialization. The `ReplicaManager` uses 138 line of code for starting and synchronizing the replicas.

System call handling requires more than 400 lines of code in ELK Herder. 89 of these lines are for the `SharedSyscall` class, which synchronizes the replicas at each system call and decides which replica actually executes the system call.

As described in Chapter 4, each replica runs as a separate thread, handling these threads takes 110 lines of code.

5.2 Memory Overhead

I measured ELK Herders memory footprint with the Massif tool, included in the Valgrind [NS07] tool-set. I profiled a C application which just returns from `main()` without any further action. If run on Linux natively the program uses about 1 KiB of heap space. Initially ELKVM maps 16 MiB of memory for each binary, as explained in Chapter 4. ELK Herder maps the initial memory for each of the replicas. When run with three replicas, ELK Herder uses 48 MiB of memory for the replicas. I substracted this memory from the values determined by Massif to get the heap space used for ELKVM's and ELK Herder's data structures. Figure 5.1 shows the heap space for ELK Herder with one to four replicas in blue. As a comparison, the figure shows the heap used by the native binary in red. For each replica ELK Herder uses about 16 KiB for its own data structures.

ELK Herder includes all data of an application in its Sphere of Replication. Therefore total memory overhead is proportional to the number of replicas used by ELK Herder. This means that for dual-modular-redundancy ELK Herder uses twice the memory as

without replication, for triple-modular-redundancy, ELK Herder uses three times as much memory.

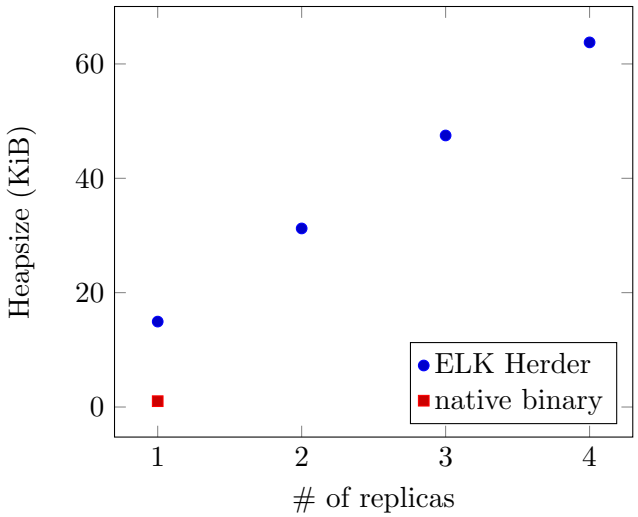


Figure 5.1: Memory required by ELK Herder

5.3 Fault Tolerance

Executing ELK Herder with two replicas, allows for detecting hardware errors, while three or more replicas allow for recovery from hardware errors. The amount of executed replicas can be freely specified with a command-line flag.

ELK Herder's recovery properties depend on the amount and distribution of faults. An arbitrary amount of faults in a single replica can be tolerated, because this affects only the checksum of one replica. The number of replicas in which faults can be tolerated is limited.

For n replicas running, ELK Herder can always tolerate faults in t replicas with $t = \lceil (n/2) \rceil - 1$ as described in Chapter 2. While adding more replicas results in a higher fault tolerance, adding more replicas generally adds synchronization overhead and consumes more resources. Therefore resulting in higher runtime overhead.

In order to show that ELK Herder can detect and recover from transient hardware faults I added the ability to inject faults into arbitrary virtual CPU registers at arbitrary instructions of the replicated binary. A fault-injecting replica, described in Section 4.2.3 sets a breakpoint at the pre-defined location and calls a breakpoint callback defined in ELK Herder. The handler flips the specified bit in the replica and returns control to the guest application.

In order to test error detection and recovery, I inspected a pre-compiled bit-counter benchmark binary and injected one fault during every run of the binary. The injection was repeated for every bit in every general purpose register and the `rflags` register, resulting in 17 registers with 64 bits. In total, I injected $17 * 64 = 1,088$ faults into each

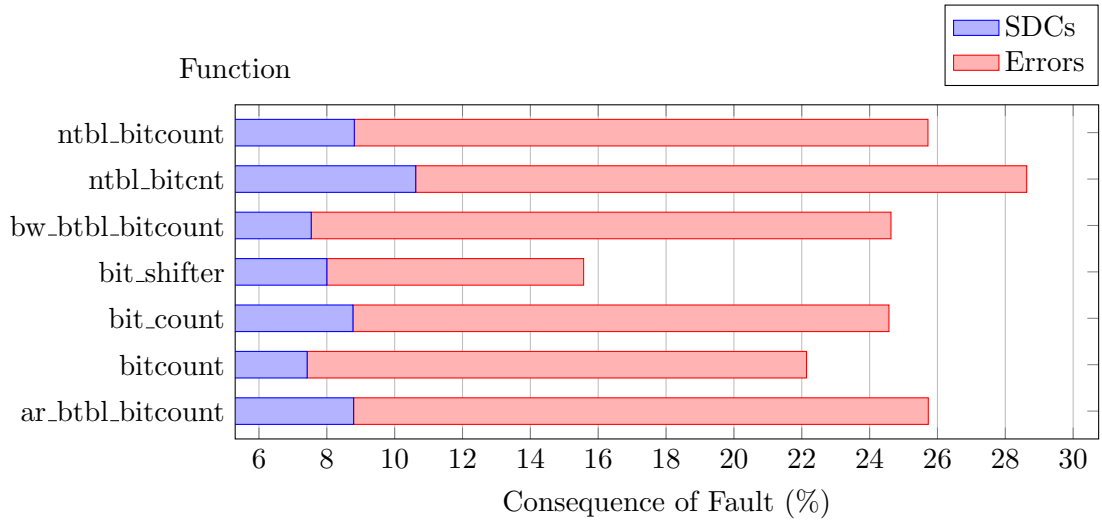


Figure 5.2: Faults, errors and SDCs for bitcount's different methods

instruction. I compared the computational result for each run to a fault-free run of the binary.

I injected faults into each instruction of the binary's seven different calculation methods. Figure 5.2 shows what percentage of the faults injected into each function resulted in a silent data corruption or an error. In this case an error means, that the binary either crashed, or entered an endless loop and had to be killed.

I injected 11,968 faults into the `bit_count()` method, which resulted in 1,891 errors and 1,049 silent data corruptions, when run without fault tolerance. When run with ELK Herder, there were 2,940 replica reinitializations, but no SDCs and no errors. For the other computing functions, the results are similar. The number of reinitializations, when run with ELK Herder equals the number of SDCs + errors, when run without fault tolerance.

Most of the functions experience errors or SDCs for around 25% of the faults. The `bit_shifter()` method experiences less errors and SDCs, with only 15% and the `ntbl_bitcnt()` method experiences errors and SDCs for almost 30% of the faults.

5.4 Runtime Overhead

I have done some performance measurements, in order to substantiate the decision to use ELKVM, instead of the `ptrace()` system call for system call interception. In order to get a broad overview of the runtime overhead of ELKVM and ELK Herder in more realistic environments, I compare the runtimes of different SPEC CPU 2006 benchmarks, between running them natively, running them with ELKVM and with ELK Herder.

All performance measurements were taken on a quad-core Intel Core i5-3550 with 3.3 GHz, 256 KiB L2 cache for each core and 6 MiB L3 cache. The system is equipped

with 4 GiB main memory, running Linux 3.11 with the KVM kernel module patched to return to user-space for calls to the `vmcall` and `vmmcall` instructions. TurboBoost and power saving features of the CPU were turned off to get reproducible results.

The decision to use ELKVM to intercept system calls for ELK Herder founds on the results of the micro-benchmarks described in Section 3.4, which show that intercepting system calls with ELKVM introduces an order of magnitude less overhead than the `ptrace()` system call, for all system calls not related to memory management. For the `mmap()` and `brk()` system calls, ELKVM introduces about the same overhead as `ptrace()`.

For this evaluation I want to distinguish between the runtime overhead of the ELKVM library and ELK Herder. In order to evaluate the runtime overhead of ELKVM, I implemented an additional ELKVM monitor application. The ELKVM proxy only forwards the system calls to the C library running in the host, this way the overhead of ELK Herder's register checksumming etc. is averted.

Unfortunately a memory corruption bug in ELKVM prevents six of the SPEC CPU 2006 benchmarks from executing with the ELKVM proxy and ELK Herder. The benchmarks which do not work are the 403.gcc, 429.mcf, 464.h264ref, 483.xalancbmk, 447.dealII and 465.tonto benchmarks. I could not find the exact cause of this bug on time for this work.

The remaining 23 benchmarks showed a geometric-mean overhead of 2.2% when run with the ELKVM proxy and 16.3% in ELK Herder with triple-modular redundancy.

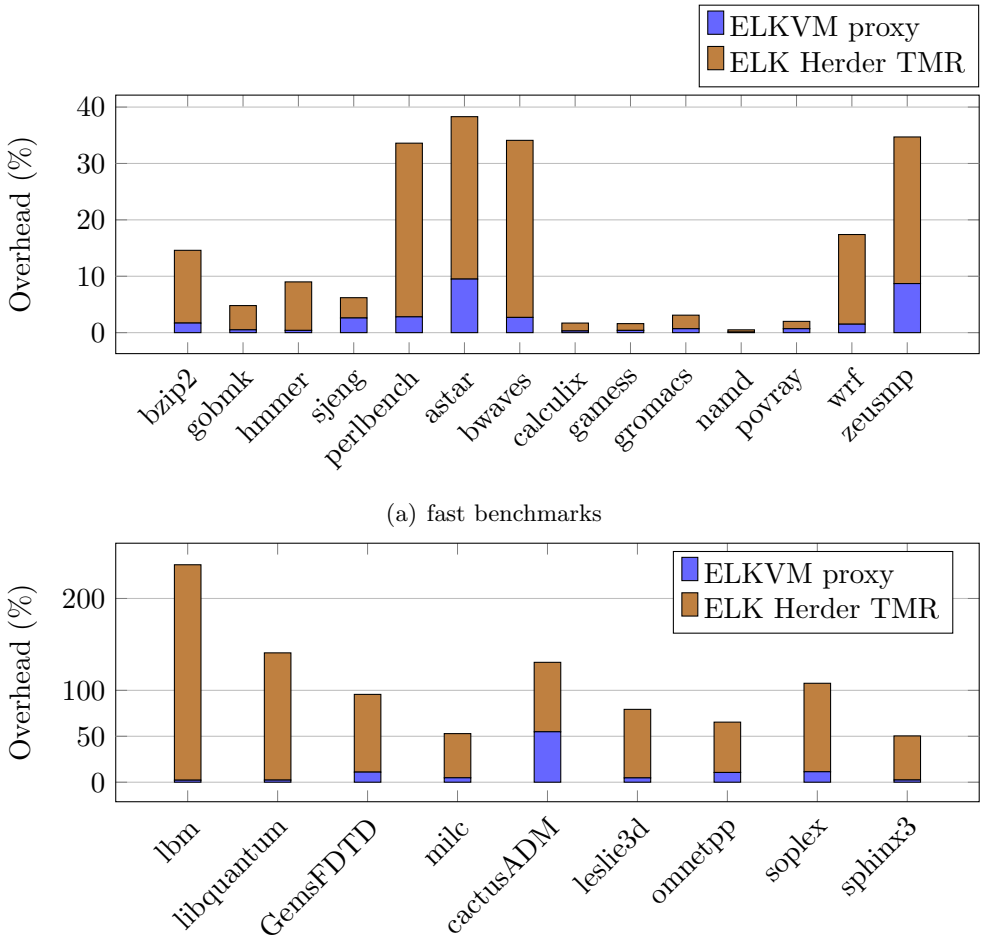
Figure 5.3 shows the overhead of the SPEC CPU 2006 benchmarks for the ELKVM proxy in blue. The maximum overhead induced by ELKVM lies at 55% for the 436.cactusADM benchmark. The 434.zeusmp, 450.soplex, 471.omnetpp and 473.astar benchmarks have an overhead of around 10%. All other benchmarks have less than 5% overhead.

In brown, Figure 5.3 shows the results for ELK Herder with triple-modular redundancy. With TMR, the overhead differs more between the different benchmarks. The 416.gamess, 435.gromacs, 444.namd, 454.calculix and 458.sjeng benchmarks all have less than 5% overhead.

Figure 5.3(b) shows the seven SPEC benchmarks with more than 50% overhead in ELK Herder. Most notable among these is the 470.lbm benchmark with almost 235% overhead. The 410.bwaves, 462.libquantum and 470.lbm benchmarks all have significantly higher overhead in ELK Herder but low overhead in ELKVM. 470.lbm has the most extreme divergence, with only 5% overhead in ELKVM but 235% overhead in ELK Herder.

In order to be able to explain the overheads of the SPEC benchmarks, I used the `perf` [per] tool to measure cache references, cache misses, branch misses and faults and normalized these values to runtime. I also counted the system calls made by each benchmark with the `strace` [str] utility.

Figure 5.4(a) shows cache misses per second on the x-axis and the overhead on the y-axis. Each dot in the figure represents one of the SPEC benchmarks. The figure shows a



(a) fast benchmarks

(b) slow benchmarks

Figure 5.3: Normalized Runtimes of the SPEC benchmarks

small correlation between cache misses and overhead, but does not explain the high overhead of the 470.lbm benchmark, which clearly stands out in the top-center region of the figure. The 462.libquantum benchmark, as well as the 436.cactusADM, 459.GemsFDTD and the 437.leslie3d benchmarks, also show a discrepancy in that regard.

Figure 5.4(b) shows system calls per second on the x-axis and the overhead of the benchmarks on the y-axis. No correlation between the amount of system calls and the overhead of the benchmarks can be seen. Additionally, I measured system calls by different system call types, which did not give any more conclusive results.

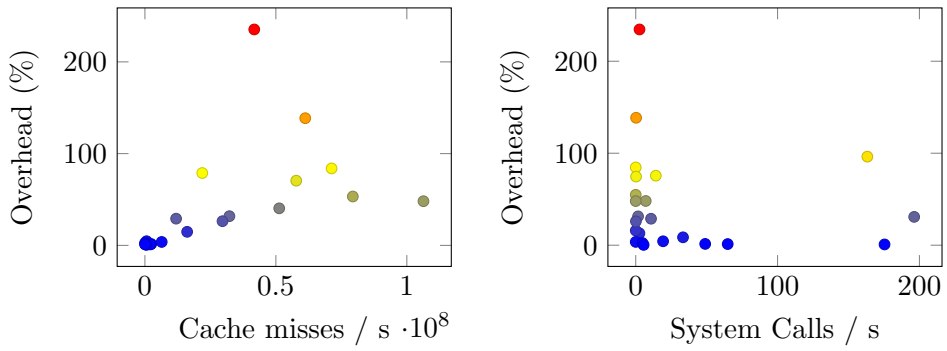


Figure 5.4: Cache misses and System calls per Second in ELK Herder

In an effort to investigate the high overhead of the 470.lbm benchmark, I measured the SPEC benchmarks on a hexa-core Intel Xeon X5650 CPU with 2.67 GHz and 24 GiB of main memory. Putting the replicas on different CPU sockets does not increase but decrease ELK Herder's performance. The added overhead for each benchmark is shown in Figure 5.5 in light brown. The dark brown bars show the overhead of ELK Herder if run on the same socket. The Figure shows that the overhead of the 470.lbm benchmark with TMR is only 63% if all ELK Herder threads run on the same socket, as opposed to 114%, if they run on multiple sockets.

These numbers suggest, that the overhead comes from ELK Herder's synchronization mechanisms, because the only occurrence of communication between the replicas and the `ReplicaManager`, is during a system call or an interrupt. During the other execution phases the replicas execute independently from each other. However, further examination of the 470.lbm benchmark showed that the overhead originates from the `performStreamCollide()` function in the benchmark, which does not execute any system calls. This again suggests memory or cache congestion and not synchronization overhead.

5.5 Replica Reinitialization

In order to evaluate the time ELK Herder takes to reinitialize a replica I implemented a benchmark, that `mmap()`s a specific amount of memory, writes to each mapped page

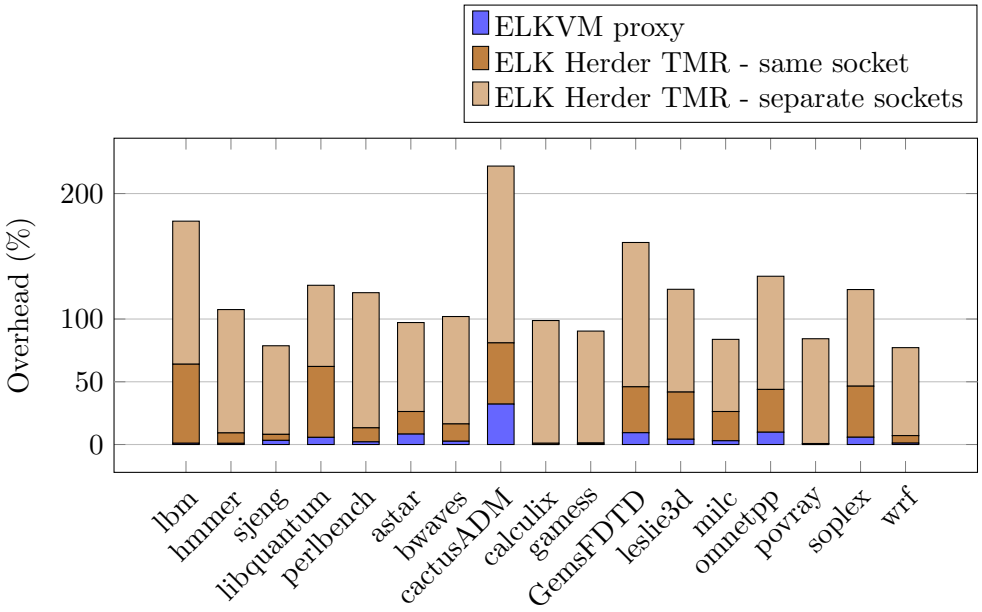


Figure 5.5: Normalized Runtimes of the SPEC benchmarks on separate sockets

once and calls `getpid()` afterwards. The benchmark is executed with three replicas. In order to trigger a restart of one of the replicas I injected a transient fault into one of the replicas at the `syscall` instruction in the C library's `getpid()` wrapper. I measured the total runtime of this binary with and without injecting a fault and determined the time necessary for a restart of a replica by subtracting the total runtime of the binary without a fault from the total runtime of the binary with a fault.

I repeated this experiment, with memory mappings ranging from 25 MiB to 1000 MiB with increases in size of 25 MiB. Figure 5.6 shows the time needed for a replica restart in context of the memory mapped to the replica.

Consider the red plot in the figure. The plot shows the total memory needed for all three replicas as a function of the memory mapped by the replicated binary. ELK Herder's memory usage is constant between 25 MiB and 100 MiB. This is due to the buddy allocator used to manage the replica's memory. Starting at 125 MiB, ELK Herder cannot find a suitable memory region in the first chunk, because some of the memory has been used by the binary's environment and stacks. Therefore ELK Herder requests another memory region from the host's Linux kernel and maps this to the replica. For mappings up to 250 MiB, this region is large enough to fit the memory requested by the guest. Afterwards, the guest binary's request does not fit into the standard 256 MiB mapping, therefore ELK Herder increases the size of the mapping on demand.

The blue plot in Figure 5.6 shows the time ELK Herder needs to reinitialize a faulty replica. The plot shows, an increase in time between 100 MiB and 125 MiB and a gradual increase when more than 250 MiB are mapped to the guest. This is expected behavior, because ELK Herder needs to copy more memory for a replica reinitialization at these data points.

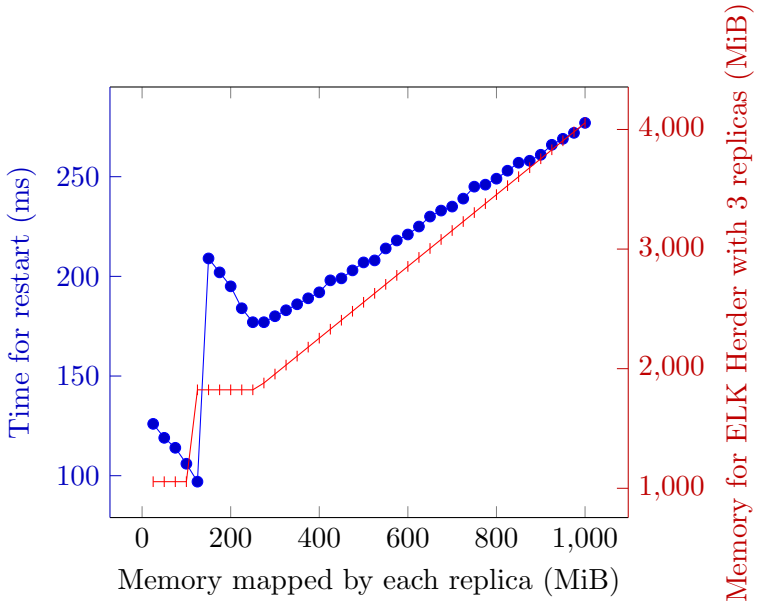


Figure 5.6: Time required to reinitialize a replica

Contrary to expectations, between 25 MiB and 125 MiB the reinitialization time decreases. However this behavior is explained by the way the benchmark binary uses the allocated memory. Because this binary touches each mapped page once, each of these pages will be mapped in the host. All unused pages will be used the first time on the `memcpy()`, when the replica is reinitialized. This causes a page fault in the host Linux kernel for each of these pages. A binary that uses a higher percentage of the memory mapped by ELK Herder will cause less pages faults on replica reinitialization.

I verified this theory with a version of ELKVM, that touches each page of a new mapping, thus mapping the page eagerly. With this version the reinitialization times are constant with regard to the memory needed by ELK Herder. Reinitialization is also about two times faster than with lazily mapped memory.

In 1981 Bartlett found, that faults do not happen often, therefore "the correctness of the error recovery scheme is far more important than its efficiency" [Bar81]. Although this may have changed during the last 33 years, I prioritized on the correctness of ELK Herder's recovery mechanism during the course of this work. For future versions of ELK Herder, I plan on improving the efficiency of the error recovery.

Currently ELK Herder copies all memory, that has been mapped to a virtual machine, for error recovery. This can be improved by copying only the memory that is actually in use at the time of the reinitialization. The `ReplicaManager` can find the memory, which is in use by inspecting the guest page tables.

6 Conclusion and Further Work

With ELKVM I created a library, that can run an ELF binary in a virtual machine without a full-scale operating system. In order to achieve this, ELKVM sets up a minimal proxy OS inside the virtual machine and redirects all system calls to a monitor application. I created ELKVM as a library, because ELKVM can be used for any use-case that requires interception of system calls, including but not limited to replication. Examples include memory leak detection, system call security monitoring and application profiling. KVM's single-stepping and breakpoint features can be used to create a debugger with ELKVM.

ELKVM only requires KVM to run, therefore ELKVM is portable between different hardware architectures. ELKVM benefits, whenever KVM is ported to a new architecture, if ELKVM is adjusted to run on this architecture. In contrast to porting binary recompilation to different architectures, the necessary adjustments are small.

Full-scale virtual machines can be stopped and migrated from one host system to another in less than one second [NLH+05]. ELKVM can be used for a user-level implementation of process migration such as in MOSIX [BL98, MDP+00].

My benchmarks show, that for system call interception ELKVM introduces considerably less overhead than the `ptrace()` system call. Most system calls are intercepted with an order of magnitude less overhead, while the `brk()` and `mmap()` system calls used to obain memory, have the same overhead in ELKVM and `ptrace()`.

With ELK Herder I created a replica manager, which uses virtual machines to achieve isolation between the replicas. ELK Herder uses ELKVM to intercept the replica's system calls and make sure the replicas receive the same inputs at all times.

ELKVM and ELK Herder can successfully run most of the SPEC CPU 2006 benchmarks. The 403.gcc, 429.mcf, 464.h264ref, 483.xalancbmk, 447.dealII and 465.tonto benchmarks do not run due to a memory corruption bug in ELKVM. I could not find the exact cause of this bug on time for this work. However, a fix for this bug is a priority for the future work on ELKVM.

The implementation of the system calls `fork()`, `exec()` and `ioctl()` is left for the future. I explored a general idea to implement support for these system calls in Chapter 3.

Currently signals are only injected into the replicas after system calls, for the reasons described in Section 4.1.3. This behavior can introduce long periods without signal handling in compute-bound applications. Injecting signals with instruction-based watchdogs, described by Kriegel [Kri13], may provide more frequent opportunities for ELK Herder to inject signals into these applications.

Inter-process-communication with shared memory does not use signals or system calls and is an additional method of input into an application. Mushtaq et al. [MAAB12] described a method to support shared memory in replicated applications. I outlined an idea to adapt their method for ELK Herder in Section 3.6.

Dynamic replication, described by Muschner [Mus13] is useful to distinguish between sections, which are more susceptible to transient faults and sections which are less susceptible to those faults. With dynamic replication, the amount of replicas can be lowered or replication can be completely turned off in program sections, where performance is more important. In combination with checkpoints and roll-back, ELK Herder can run with less overhead in the fault-free case, and still provide recovery of replicas.

At the moment ELK Herder only supports single-threaded applications and ELKVM only supports virtual machines, which use only one virtual CPU. However KVM in Linux 3.11 supports 128 virtual CPUs and ELKVM can be adapted to support more CPUs as well. Multi-threading requires support for multiple VCPUs in ELKVM.

The 23 running SPEC CPU 2006 benchmarks showed a geometric-mean overhead of 2.2% in ELKVM and 16.3% in ELK Herder with triple-modular redundancy.

Error injection experiments showed that ELK Herder, running with three replicas, can handle transient hardware faults in one replica. I expect ELK Herder to be able to tolerate transient hardware faults in multiple replicas, as long as ELK Herder is run with more than four replicas and less than half of the replicas experience a transient hardware fault.

Appendix A

Glossary

API Application Programming Interface

DMR Double modular redundancy

ECC Error Checking Code

KVM The Linux Kernel Virtual Machine

L4Re L4 Runtime Environment

OS Operating System

SDC Silent data corruption

SoR Sphere of Replication

TMR Triple modular redundancy

VM Virtual Machine

VMM Virtual Machine Monitor

Bibliography

[ALRL04] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, 2004.

[AN97] K Arun and H Nitin. Understanding fault tolerance and reliability. 1997.

[Bar81] Joel F Bartlett. A nonstop kernel. *ACM SIGOPS Operating Systems Review*, 15(5):22–29, 1981.

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[BL98] Amnon Barak and Oren La'adan. The mosix multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4):361–372, 1998.

[BS96] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. Comput. Syst.*, 14(1):80–107, February 1996.

[dKNS10] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 497–508, New York, NY, USA, 2010. ACM.

[EEL+97] Susan J Eggers, Joel S Emer, Henry M Leby, Jack L Lo, Rebecca L Stamm, and Dean M Tullsen. Simultaneous multithreading: A platform for next-generation processors. *Micro, IEEE*, 17(5):12–19, 1997.

[Gra86] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on reliability in distributed software and database systems*, pages 3–12. Los Angeles, CA, USA, 1986.

[HBD+13] Jörg Henkel, Lars Bauer, Nikil Dutt, Puneet Gupta, Sani Nassif, Muhammad Shafique, Mehdi Tahoori, and Norbert Wehn. Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 99:1–99:10, New York, NY, USA, 2013. ACM.

[JF07] Casey M Jeffery and Renato JO Figueiredo. Towards byzantine fault tolerance in many-core computing platforms. In *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*, pages 256–259. IEEE, 2007.

[Kno65] Kenneth C Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–624, 1965.

[Knu73] Donald Ervin Knuth. *The art of computer programming. 1, (1973). Fundamental algorithms*. Addison-Wesley, 1973.

[Kol06] Kirill Kolyshkin. Virtualization in linux. *White paper, OpenVZ*, 2006.

[Kri13] Martin Kriegel. Bounding error detection latencies for replicated execution. Bachelor thesis, TU Dresden, 2013.

[kvma] Kvm api documentation.

[kvmb] Kvm: Kernel-based virtualization driver.

[KW00] Poul-Henning Kamp and Robert NM Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, volume 43, page 116, 2000.

[M+65] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[MAAB11] Hamid Mushtaq, Zaid Al-Ars, and Koen Bertels. Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems. In *Design and Test Workshop (IDT), 2011 IEEE 6th International*, pages 12–17. IEEE, 2011.

[MAAB12] Hamid Mushtaq, Zaid Al-Ars, and Koen Bertels. A user-level library for fault tolerance on shared memory multicore systems. In *Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2012 IEEE 15th International Symposium on*, pages 266–269. IEEE, 2012.

[MAAB13] Hamid Mushtaq, Zaid Al-Ars, and Koen Bertels. Efficient software-based fault tolerance approach on multicore platforms. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 921–926. EDA Consortium, 2013.

[mana] *clone(2) - Linux man page.*

[manb] *ld.so(8) - Linux man page.*

[manc] *ptrace(2) - Linux man page.*

[man11] *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, Dec 2011.

[man13] *Intel® 64 and IA-32 Architectures Software Developer's Manual*, June 2013.

[MDP+00] Dejan S Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Computing Surveys (CSUR)*, 32(3):241–299, 2000.

[MP86] Sanjay Mehrotra and Gust Perlegos. Error checking and correction circuitry for use with an electrically-programmable and electrically-erasable memory array, September 16 1986. US Patent 4,612,640.

[Mus13] Robert Muschner. Reducing resource consumption of replication using dynamic replicas. Diploma thesis, TU Dresden, 2013.

[NLH+05] Michael Nelson, Beng-Hong Lim, Greg Hutchins, et al. Fast transparent migration for virtual machines. In *USENIX Annual Technical Conference, General Track*, pages 391–394, 2005.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *ACM Sigplan Notices*, 42(6):89–100, 2007.

[per] perf(1) - linux man page.

[RCV+05] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. Swift: Software implemented fault tolerance. In *In Proceedings of the 3rd International Symposium on Code Generation and Optimization*, pages 243–254, 2005.

[rep97] System V Application Binary Interface. Technical report, Mar 1997.

[RM00] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 25–36, New York, NY, USA, 2000. ACM.

[Rot99] Eric Rotenberg. Ar-smt: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, FTCS '99, pages 84–, Washington, DC, USA, 1999. IEEE Computer Society.

[SBM+09] Alex Shye, Joseph Blomstedt, Tipp Moseley, Vijay Janapa Reddi, and Daniel A. Connors. Plr: A software approach to transient fault tolerance for multicore architectures. *IEEE Transactions on Dependable and Secure Computing*, 6(2):135–148, 2009.

[Sch90] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[Sch93] Fred B Schneider. Replication management using the state-machine approach, distributed systems. 1993.

[Sea00] David Seal. *ARM architecture reference manual*. Pearson Education, 2000.

[Sie95] Daniel P Siewiorek. Niche successes to ubiquitous invisibility: Fault-tolerant computing past, present, and future. In *25th Fault-Tolerant Computing Symp*, pages 26–33, 1995.

[SKK+02] Premkishore Shivakumar, Michael Kistler, Stephen W Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 389–398. IEEE, 2002.

[str] strace(1) - linux man page.

[WEMR04] Christopher Weaver, Joel Emer, Shubhendu S. Mukherjee, and Steven K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 264–, Washington, DC, USA, 2004. IEEE Computer Society.

[Yeh96] YC Yeh. Triple-triple redundant 777 primary flight computer. In *Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE*, volume 1, pages 293–307. IEEE, 1996.