uniserver

D5.2 Error-Resilient Hypervisor

Contract number	688540
Project website	http://www.Uniserver2020.eu
Contractual deadline	Project Month 18 (M18): 31st July 2017
Actual Delivery Date	15th September 2017
Dissemination level	PUBLIC
Report Version	1.0
Main Authors	Bin Wang (QUB), Lev Muchanov (QUB) Christos Antonopoulos (UTH), Charles J Gillan (QUB), Georgios Karakonstantis (QUB), Christos Kalogirou (UTH), Panos Koutsovasilis (UTH), Emmanouil Maroudas (UTH), Dimitrios Nikolopoulos (QUB)
Reviewers	Rafique Mustafa (IBM) , Hans Vandierendonck (QUB)
Keywords	Hypervisor, Operating System, Error-Resilience

Notice: The research leading to these results has received funding from the European Community's Horizon 2020 Programme for Research and Technical development under grant agreement no. 688540.

© 2017. UniServer Consortium Partners. All rights reserved

Disclaimer

This deliverable has been prepared by the responsible Work Package of the Project in accordance with the Consortium Agreement and the Grant Agreement Nr 688540. It solely reflects the opinion of the parties to such agreements on a collective basis in the context of the project and to the extent foreseen in such agreements.

Acknowledgements

The work presented in this document has been conducted in the context of the EU Horizon 2020. UniServer is a 36-month project that started on February 1st, 2016 and is funded by the European Commission. The partners in the project are:

The Queen's University of Belfast (QUB)

The University of Cyprus (UCY)

The University of Athens (UoA)

Applied Micro Circuits Corporation Deutschland Gmbh (APM)

ARM Holdings UK (ARM)

IBM Ireland Limited (IBM)

University of Thessaly (UTH)

WorldSensing (WSE)

Meritorious Audit Limited (MER)

Sparsity (SPA)

More information

Public UniServer reports and other information pertaining to the project are available through the UniServer public Web site under http://www.Uniserver2020.eu.

Change Log

Version	Description of change
1.0	Final version prepared with all inputs and review comments addressed for submission to EU.

Table of Contents

Index of Figures

Glossary of Terms

Glossary	Explanation
KVM	KVM (Kernel-based Virtual Machine) is a full virtualization technique in kernel space that enable the host directly manage guest kernel [1].
Userspace	Userspace is the memory area where application and drivers execute [2].
Kernelspace	Kernel space runs a privileged system kernel, kernel extensions, and most device drivers [2].
Ptrace	ptrace is a system where a process can control another process [3].
Hypervisor	A hypervisor or virtual machine monitor (VMM) is computer software, firmware or hardware that creates and runs virtual machines [1]. In UniServer project, we focus on both userspace hypervisor (QEMU) and kernel space hypervisor (KVM).
Host & Guest	A computer on which a hypervisor runs one or more virtual machines is called a host machine, and each virtual machine is called a guest machine.
Syscall	A system call is the kernel level function where system respond the request of different kernel.
Checkpoint	In UniServer project, a checkpoint indicates snapshot which is the state of a system at a particular point in time. Also during each checkpoint, the system will compare and record the data to mirror memory.
Reliable System	A reliable system is defined as the system with error rates lowering the provided standard.
Memory Backend	The memory-backend device contains the actual host memory that backs guest RAM. This can either be anonymous mmapped memory or file-backed mmapped memory.
HugeTLB	A Translation Lookaside Buffer (TLB) that tolerates large page. Users can use the huge page support in Linux kernel by either using the mmap syscall.
Hook Function	Code that handles intercepted function calls, events or messages is called a hook.
Page Fault Handler	A page fault is a type of exception raised by computer hardware when a running program accesses a memory page that is not currently mapped by the memory management unit (MMU) into the virtual address space of a process [2]. A handler that raises page fault is called page fault handler.
Page Allocator	The page allocator includes various syscalls from Linux memory modules, such as, mmap, alloc_page, etc.
Hot-Plug	It describes the addition of components that would expand the system without significant interruption to the operation of the system [4].

Executive Summary

This deliverable describes the developed schemes that aim at enhancing the error-resiliency of the hypervisor layer that essentially enables the use of virtual machines in the UniServer platform as we discussed in the deliverable D5.1. Our work, initially aims at characterizing the sensitivity of the hypervisor structures against errors to better understand which of them critically affect the system operation. In particular, based on our initial characterization campaign we identify data structures that, in case of a hardware error, cause system crashes with an increased probability which we may choose to protect by executing on reliable cores and storing them on reliable memory zones. Essentially, the initial error-resilient scheme implemented at the hypervisor layer lies on the adoption of a heterogeneous reliability architecture, which is composed by a set of cores and memory zones that are operating at conservative but reliable operating points and a set of cores and memory zones that operate at the extended operating points. Hypervisor data structures that are found to be critical for the system operation are forced to be executed on the reliable cores and memory zones, whereas less critical structures and application tasks/data are allowed to be executed on the unreliable cores/memory. In the implemented scheme we try to choose what to execute on the reliable cores/memory to minimize the system crashes, while also getting advantage of the allowed operation at extended margins in the unreliable domain and the resultant energy savings.

In the final section, we discuss possible future enhancements including the implementation of a mirrored memory and the implementation of a checkpoint and restart mechanism for sensitive data structures that could further shield system software against the effects of potential hardware errors.

1. Introduction

The UniServer project focuses on the design and development of a system capable of operating at extended hardware margins. This will be useful for a wide range of use cases, ranging from standalone deployments and fog computing to big cloud deployments in data centers. To support such diverse scenarios, the platform must be equipped with a complete software stack able to efficiently manage the available compute / storage resources and facilitate installation, migration and replication of applications, both at the node and cloud-level. Each component of the software stack is built on top of state-of-the-art software packages ported onto the 64-bit ARM (Aarch64 or arm64) architecture of the targeted micro-server hardware.

In particular, we adopt the KVM [1] (kernelspace) and QEMU [5] (userspace) hypervisors to provide the virtualization layer to the ecosystem with a handful of benefits such as easy installation, replication and migration of applications.

In this deliverable, we describe our initial approach on enhancing the error-resiliency of the hypervisor which can be formulated into the following main steps that reflect also the organization of the document:

- Characterize the sensitivity of the hypervisor data structures under hardware errors
- Implement a heterogeneous reliability architecture consisting of reliable and unreliable cores and memory zones
- Make the necessary changes to system software to guide the execution of the data structures base on their sensitivity on cores/memory zones with the appropriate reliability which is essentially being controlled by the adopted operating points

1.1. Hypervisor Data Structure Characterization

Introducing more reliability into a system, at any level, invariably reduces performance. This means that a practical design has to strike a balance between these two attributes in the design. We conducted a set of experiments in which we performed static data characterization of both the KVM and QEMU hypervisors, in order to find an appropriate balance for the UniServer system. Our experiments considered the reliability of both memory and processors, determined which software components can safely remain in the relaxed side of the system and how dynamic adaptation of the system configuration can help us limit performance degradation to a minimum. We also need to understand per resource limiting factors, such as the delay time between a transition from nominal configuration to extended margins and vice-versa and the verification process of a completed transition. This kind of information is very important to the design decisions and the overall approach of the UniServer project. For example, it can be used as an input to the predictor module and the checkpoint mechanism.

1.2. Hypervisor Design and Implementation

We address and discuss the fault-tolerance and efficiency of the hypervisor in two separated but closely related threads, one from the memory / storage and one from the execution point of view. This separation facilitates the design, development and exploration space of the ongoing work, as well as the organization of this document. All methods and features we present in these two threads are orthogonal yet complementary and compatible.

Linux kernel divides physical memory into a number of different memory regions, which are called zones. Some zones (high mem zones) are reserved for large memory usage. We exploit this and introduce the concept of heterogeneous reliability memory which is being controlled by the selected operating points.

A heterogeneous reliability memory contains fully reliable and less reliable domains. A fully reliable domain is the domain mapped to reliable DIMMs operated at the nominal refresh rate and voltage level. On the other hand, a less reliable domain is created where unreliable DIMMs operate marginal refresh rate and voltage level. To support it, we firstly create the unreliable zone in the Linux memory layout and add flags *ZONE_UNREL*, *GFP_UNREL* to direct this zone. We then revise the corresponding syscall interfaces such as *mmap, kmalloc, alloc_page* to support unreliable flags. To implement the error resilient hypervisor for heterogeneous reliability memory, we revise the memory allocation hypercalls inside of QEMU hypervisor. For example, we can use the QEMU to boot a backend memory (guest physical RAM) for KVM.

Regarding the error resiliency of the execution units (CPUs/cores), we have developed functionality migration techniques that redirect a subset of the system functionality to reliable CPUs only. Namely we migrate interrupts, system calls/hypercalls and the page fault handler. In these cases, the SMP assumptions of the system do not hold anymore. As an abstract picture, one can assume that relaxed CPUs/cores request from the reliable ones to perform these specific tasks on behalf of them. We also provide the baseline changes in order to migrate the scheduler decisions away from the relaxed CPUs. There are more places in the hypervisor where functionality migration may prove beneficial for the overall error resilience of the system. As a technique, functionality migration is orthogonal to the mirroring and checkpoint solutions for memory and both can be combined in order to fully protect the hypervisor.

1.3. Organization

The rest of the document is organized as follows. Section 2 discusses the data characterization and analysis performed on both KVM and QEMU components of the hypervisor. This analysis reveals the most sensitive data and code of the hypervisor. In Section 3, we present the design and implementation of the error resilient memory of the hypervisor. Section 4 presents the design of the error resilient execution of the hypervisor and the mechanisms to enable / disable the introduced features. Section 5 discusses related work. Section 6 concludes the document presenting conclusions for our work and looking at future directions in which we can develop our research.

2. Hypervisor Structure Characterization

In this section, we discuss the characterization framework, and experimental results in terms of the structures / code sensitivity as well as in terms of the time to failure.

2.1. Kernelspace

The faults that hypervisor must tolerate, may originate from the two main resources of the system operating outside nominal margins: memory and processor. We consider two potential protection approaches: The first assumes full hypervisor protection and is less intrusive to the existing monolithic hypervisor architecture of Linux-KVM. The second, more challenging, approach opts for selective hypervisor protection. Selective protection requires deep insight on the fault sensitivity of the hypervisor's data structures and operations. At the same time though, selective protection has the potential to introduce less overhead.

In order to choose the appropriate and most realistic type of memory protection for the hypervisor, we performed a set of experiments to quantify the memory footprint of the hypervisor under different kinds of workloads. For these experiments, we used the applications contributed by the application partners in the UniServer project (Sparsity, WorldSensing, Meritorius).

The emulation environment as depicted in Figure 1, includes a nested (two-level) virtualization setup. The first (closer to the real hardware) level enables us to control and monitor the system from the hardware perspective, where the second (higher) level is the hypervisor under characterization and runs the workload that replicates a real-project scenario. Our emulation setup can run on the x86_64 architecture as well, leveraging our existing hardware infrastructure thus enabling development / testing before the actual ARM-based machine and arm64 versions of the applications were available.

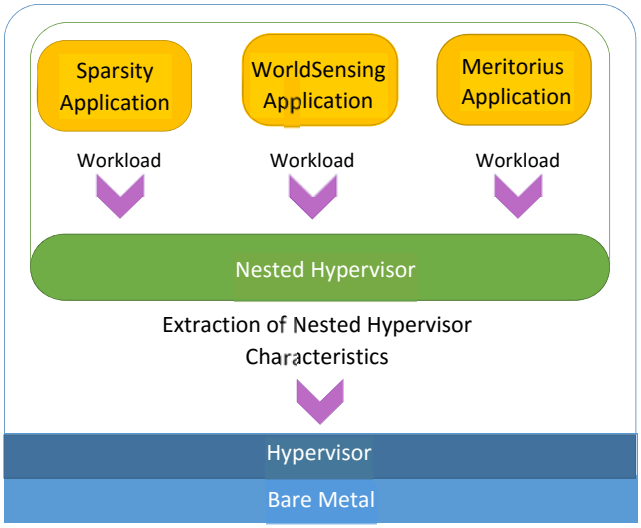

Figure 1: Emulation Environment Setup

We quantitatively evaluated the memory overhead of hypervisor data structures with respect to the memory occupied by the virtual machines (VMs) and applications running on top of them. We have measured the KVM hypervisor memory footprint by repeatedly executing four instances of VMs, each of which accommodates a benchmark (Sparsity LDBC benchmark -Figure 2, Meritorius - Figure 3, WorldSensing - Figure 4 and a mixture of the three - Figure 5). These figures show the hypervisor size in MB over time. The benchmarks stress the

CPU, disk I/O and network. As shown in the figures below, the KVM footprint (red line) is always less than 7% compared to the total utilized memory of the system. This indicates that placing the whole kernel part of the hypervisor in a reliable memory domain is a realistic and probably optimal approach. The lower part of the two figures depicts a breakdown of the memory footprint of the internal data structures of the hypervisor. Disk and disk cache-related data structures (such as buffer-head) typically dominate the KVM footprint.

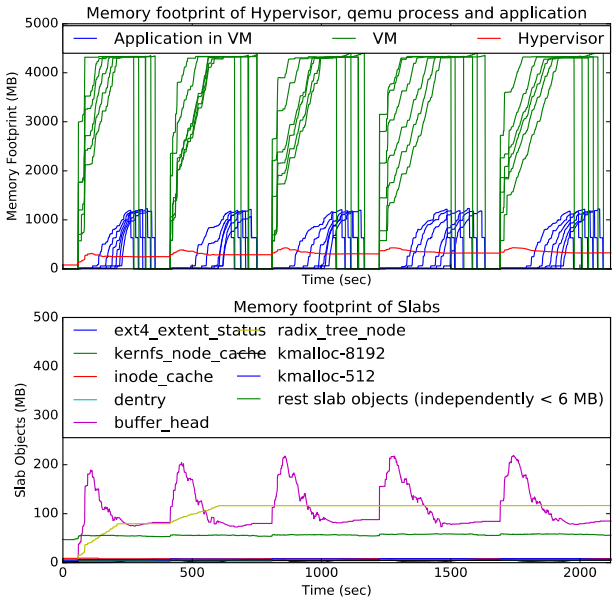

Figure 2: Footprint of Hypervisor and Slab Objects for LDBC Graph Analytics Benchmark

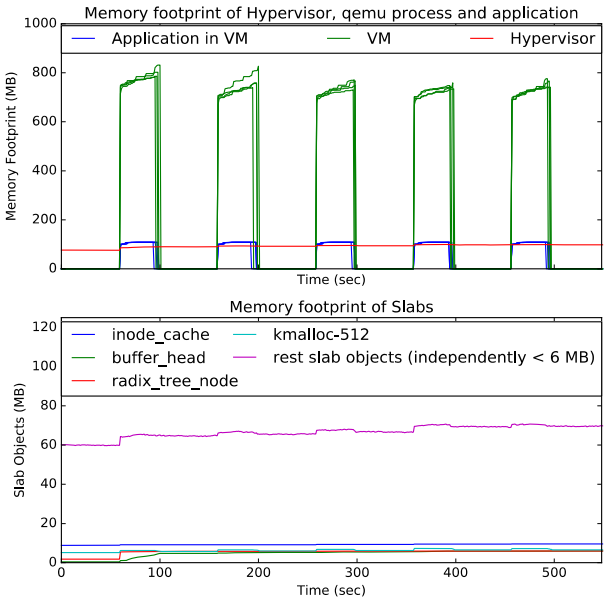

Figure 3: Footprint of Hypervisor and Slab Objects for Meritorius Benchmark

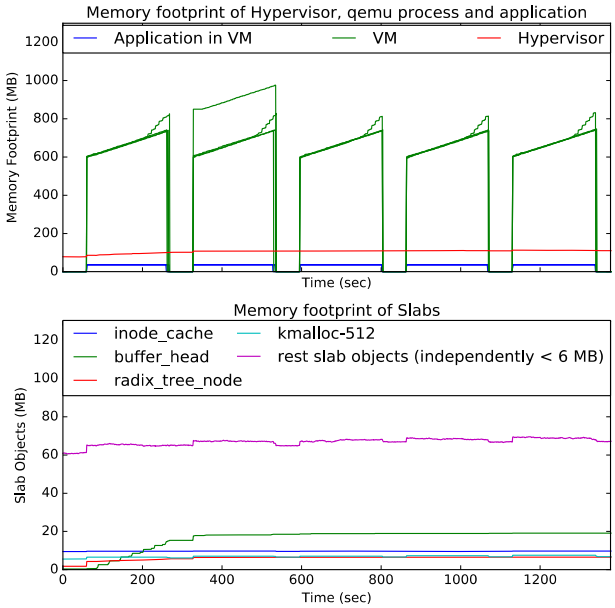

Figure 4: Footprint of Hypervisor and Slab Objects for World Sensing Benchmark

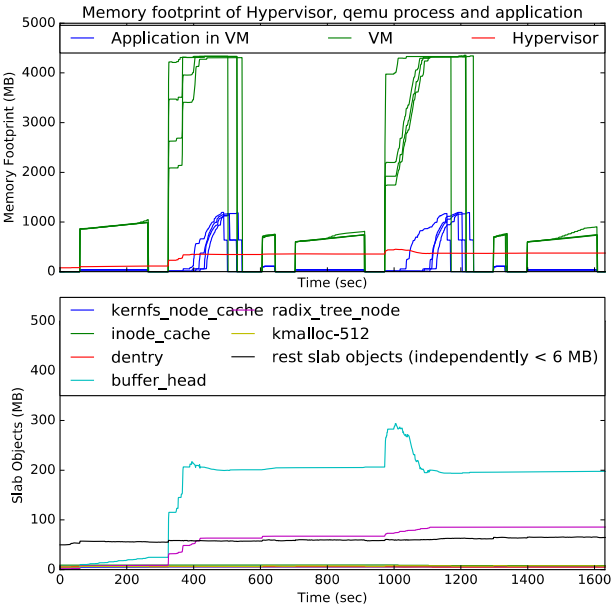

Figure 5: Footprint of Hypervisor and Slab Objects for Mixture of Benchmarks

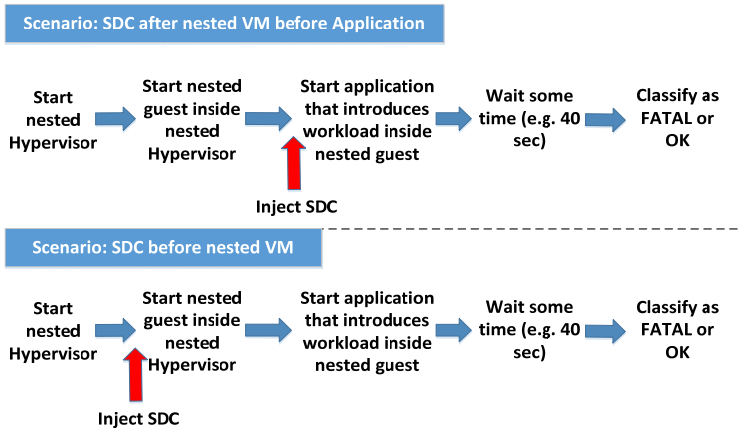

Figure 6: Workflow of Fault Injection Experiments

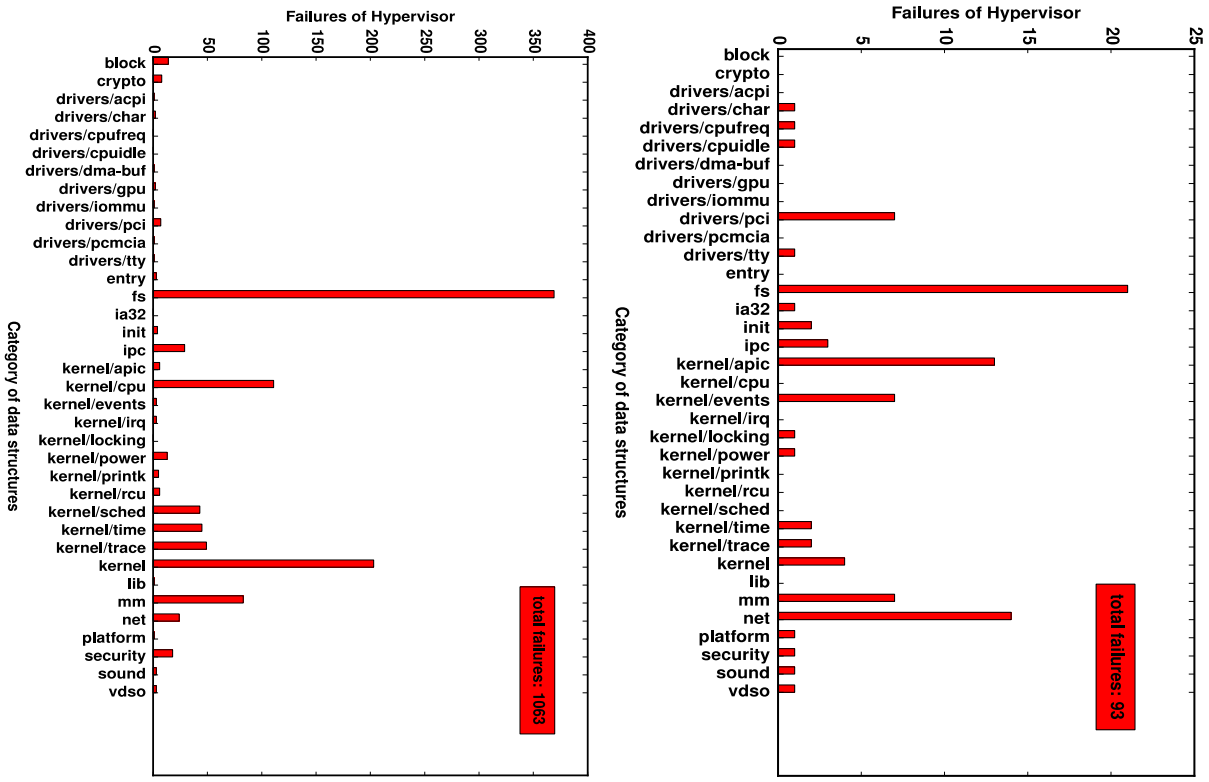

Figure 7: Left - WorldSensing SDCs before the VM, Right - WorldSensing SDCs before the Application

The KVM can be affected by CPU faults as well, which are not avoided by allocating reliable memory for KVM data structures. To characterize the sensitivity and significance of hypervisor kernel data structures and code, we have applied fault injection using our emulation environment. Specifically, for each statically allocated object of the hypervisor (total 13621 objects - Linux Kernel v4.3.0), we introduced, silent data corruptions (SDCs) in 5 independent executions (in Figure 6). For each execution, we checked whether the data corruption resulted to a non-responsive hypervisor, and marked this object accordingly as crucial or not, for the hypervisor state. In addition, we experimented with the SDC injection-time implementing two scenarios: either injecting SDCs just before initiating the application workload, or right before starting the VM.

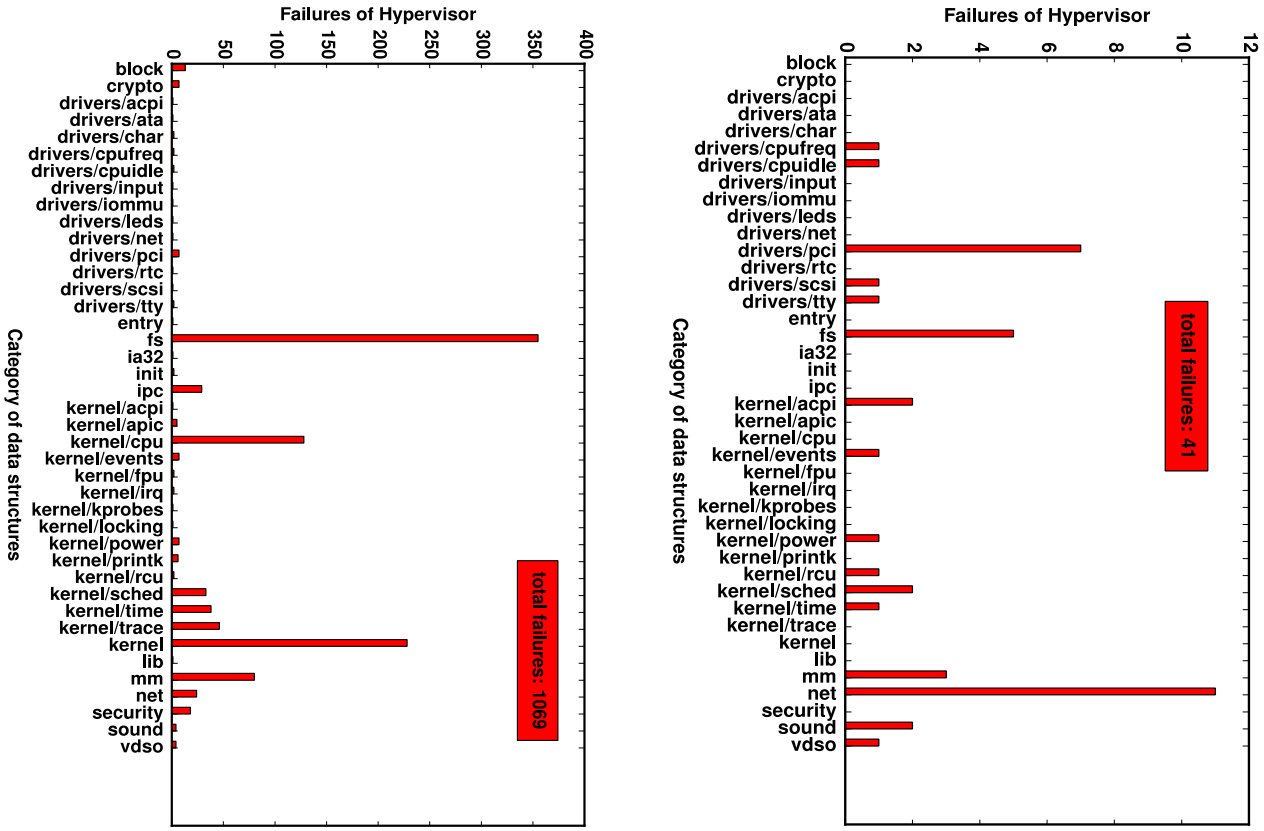

Figure 8: Left - Meritorius SDCs before the VM, Right - Meritorius SDCs before the Application

As an example, Figure 7 and Figure 8 depict the number of failures sorted according to the functionality of the static data structure affected by the SDC for the Meritorius and WorldSensing application respectively. We observe that the same fault injection rate leads to an order of magnitude more hypervisor crashes when injecting SDCs just before the initiation of the VM, compared with injecting SDCs after VM startup but right before application execution. At the same time, there is a clustering in the criticality and sensitivity of data structures and kernel code, according to their functionality. For example, data structures responsible for the fs, kernel and net subsystems are sensitive and should be protected. Interestingly, the sensitive data structures appear to be the same in both scenarios. On the other hand, they may differ in an intuitive manner under different types of workloads (for example, the WorldSensing application indeed stresses the network and filesystem, therefore increased sensitivity in those hypervisor modules should be expected).

Another metric we quantified through our experiments is the time between a fault injection and the moment an error potentially manifests and eventually leading to an unresponsive hypervisor. This information is necessary if a checkpoint-restore protection mechanism is chosen, as it provides an insight on the required frequency of checkpointing. In our experiments, we were particularly interested in latent error manifestation.

Figure 9 and Figure 10 show that the majority of errors manifested practically instantly (~97% of total manifested errors) but there are also some cases that errors manifest as late as 510 seconds after fault injection. This introduces a challenge for traditional checkpoint-restore protection mechanisms: in this time frame every computation may be untrustworthy and the safe state that the hypervisor may have to restore could be 510 or more seconds in the past, resulting to a significant amount of energy / performance loss.

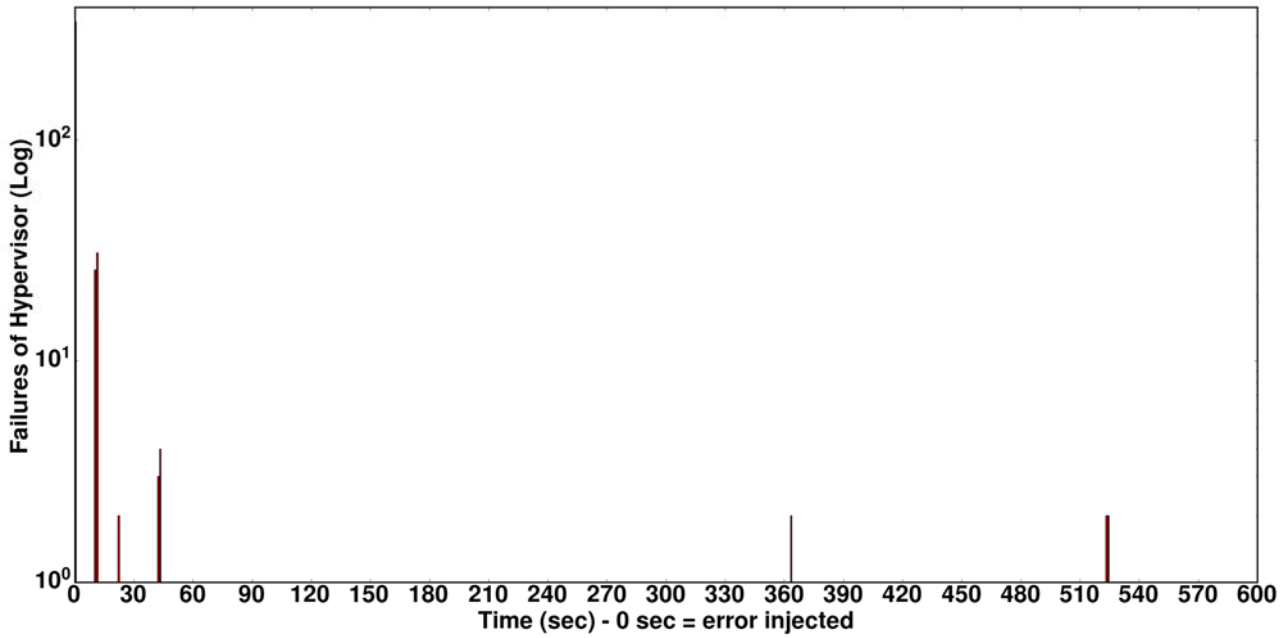

Figure 9: Latency of Error Manifestation for Meritorius Application Workload

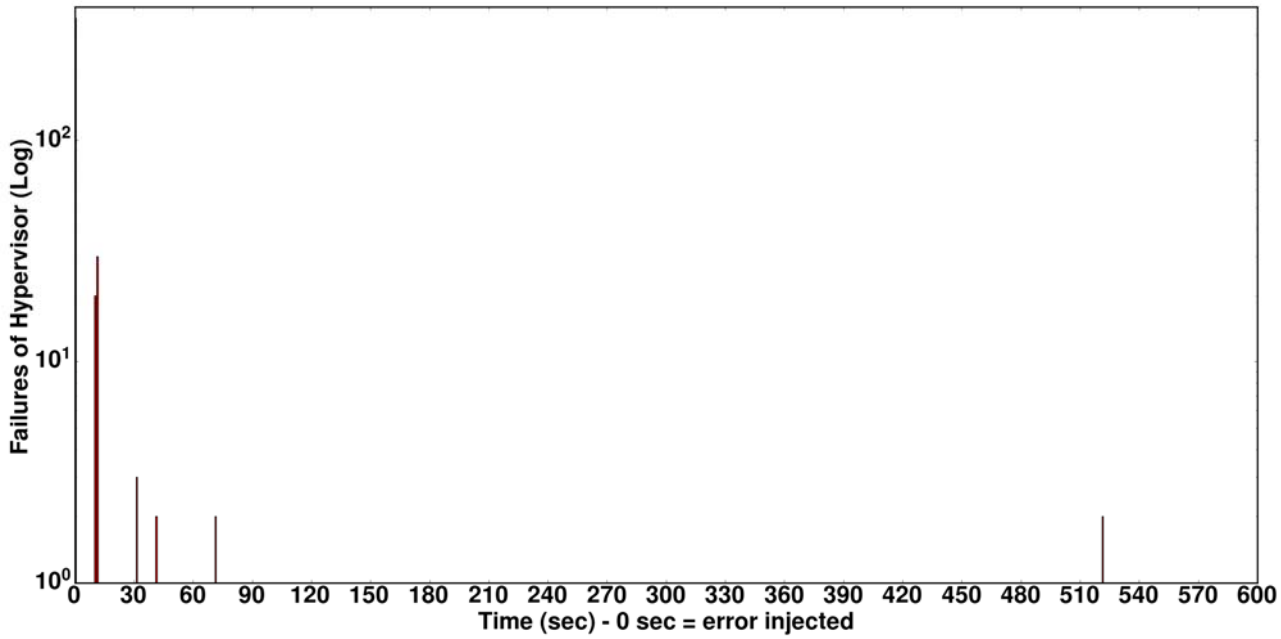

Figure 10: Latency of Error Manifestation for WorldSensing Application Workload

2.2. Userspace

In userspace data characterization, we focus on errors injected in the QEMU and trace its performance. The error is generated by *ptrace* to the QEMU process.

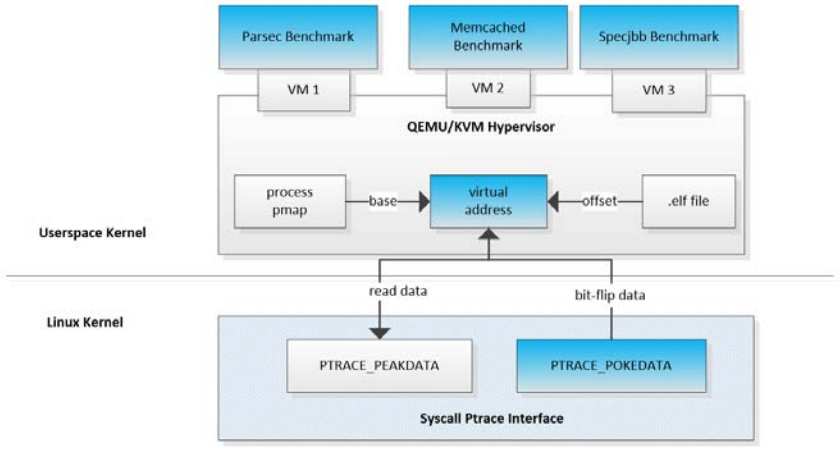

Figure 11: Userspace Error Injection Setup

Figure 11 depicts the architecture of the userspace error injection scheme. We use *objdump* to print all QEMU code value (offset) to a disassemble file. We use the *ptrace* interface which can observe and control the execution of any process at the user level, and examine and change its memory and registers. We setup two flags of *ptrace* to inject errors to QEMU modules: *PTRACE_PEEKDATA* to read data from the target address and *PTRACE_POKEDATA* to inject an error by changing data stored at a certain address.

We mainly targeted the ARM architecture and XGene2 but we used also the Intel architecture. The Libvirt [6] is used as the monitor APIs. We executed three typical Cloud benchmarks on the VMs: Parsec [7], Memcached [8], Specjbb [9]. We inject all errors when running the benchmarks and we trace only QEMU crashes. In our experiments, we discovered that some kinds of errors trigger QEMU crash and thus we record all such errors. We choose 80 typical modules which affect 3,000 QEMU functions. In this report, we list most sensitive modules and functions. For each trace, we run our simulation 50 times (1 minute per time).

Figure 12 shows the failures of modules in case the Parsec benchmark is used as a workload on top of QEMU. In this experiment, we found that the memory section has many sensitive modules. For example, the page module incurs 60 system failures, while the memory module incurs 16 failures.

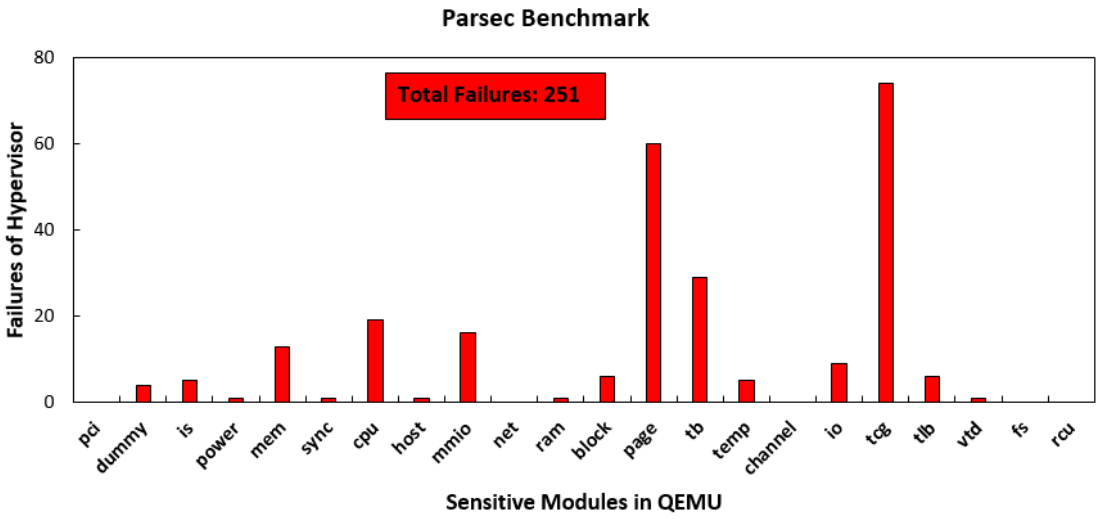

Figure 12: Failures of QEMU Modules in Parsec Benchmark

We discovered that the QEMU [5] instruction translate module, which includes *tcg* and *tb,* is the most sensitive module. We explain this by that fact that most userspace instructions of a guest VM is translated in this module and thus it is likely that any data corruption in the module will result in a hypervisor crash.

We also notice that the memory related modules (*mem, page*) induce a higher number of failures in comparison with other modules, such as *pci* and *io* modules (Figure 13).

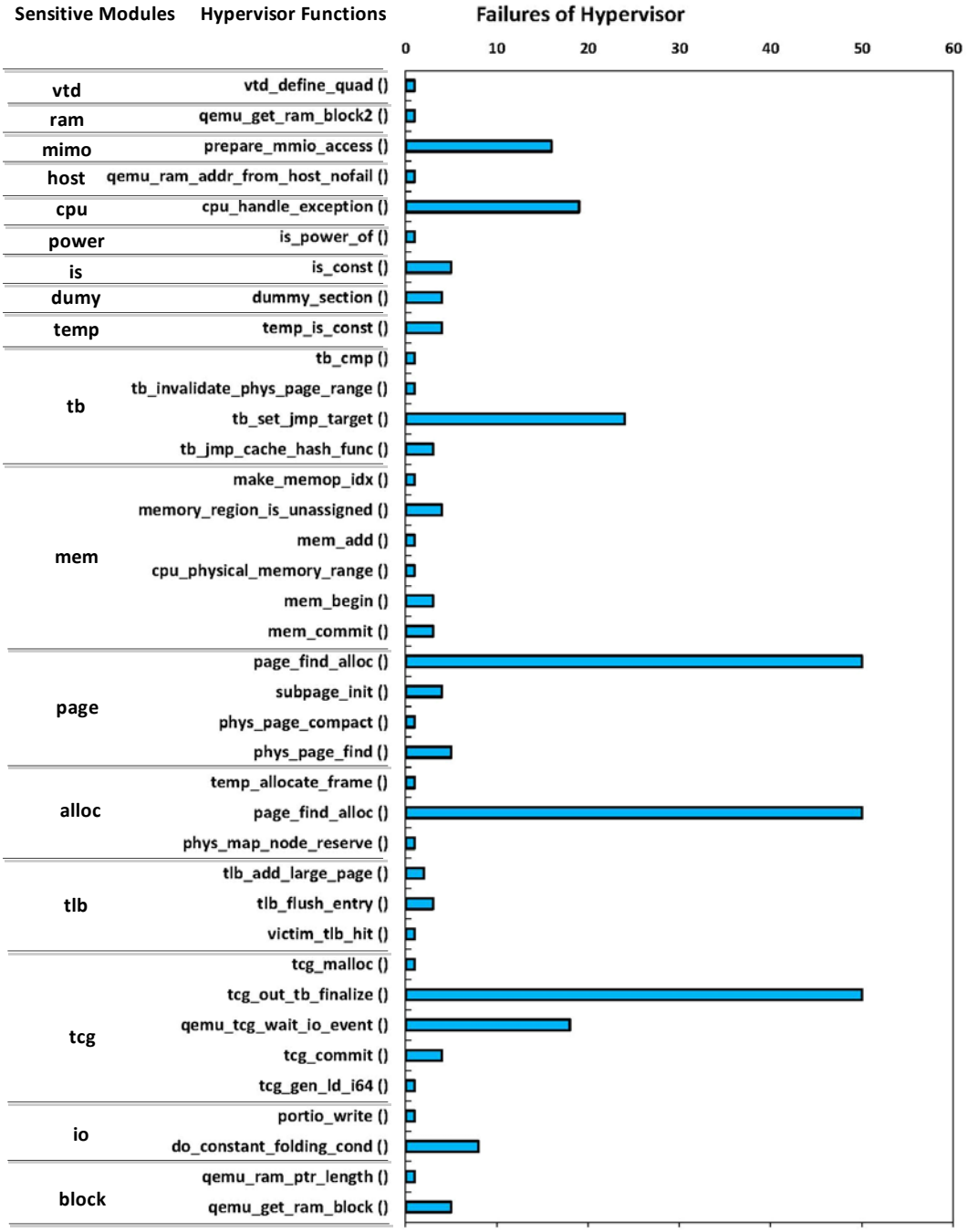

Figure 13: Hypervisor Function Failures in Parsec Benchmark

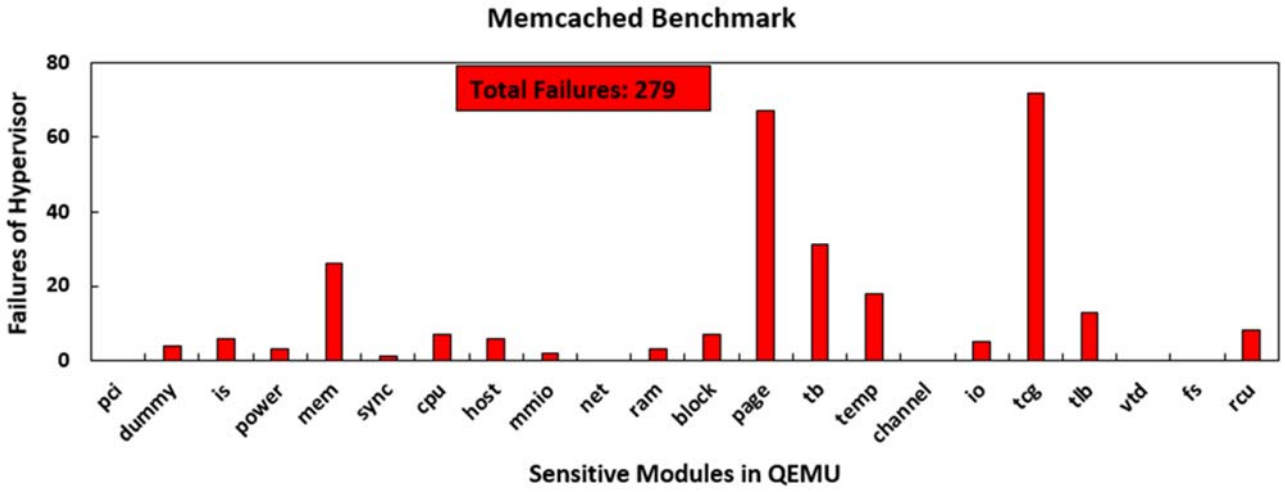

Figure 14: Failures of QEMU Modules in Memcached Benchmark

Figure 14 shows distribution of failures between QEMU modules for the Memcached benchmark run. In this experiment, we use CloudSuite [8] to emulate Memcached client and server. Memory modules such as *mem*, *page*, they incur higher failures than that in Parsec benchmark. On the other hand, *tcg* and *tb* modules are the sensitive in Figure 16.

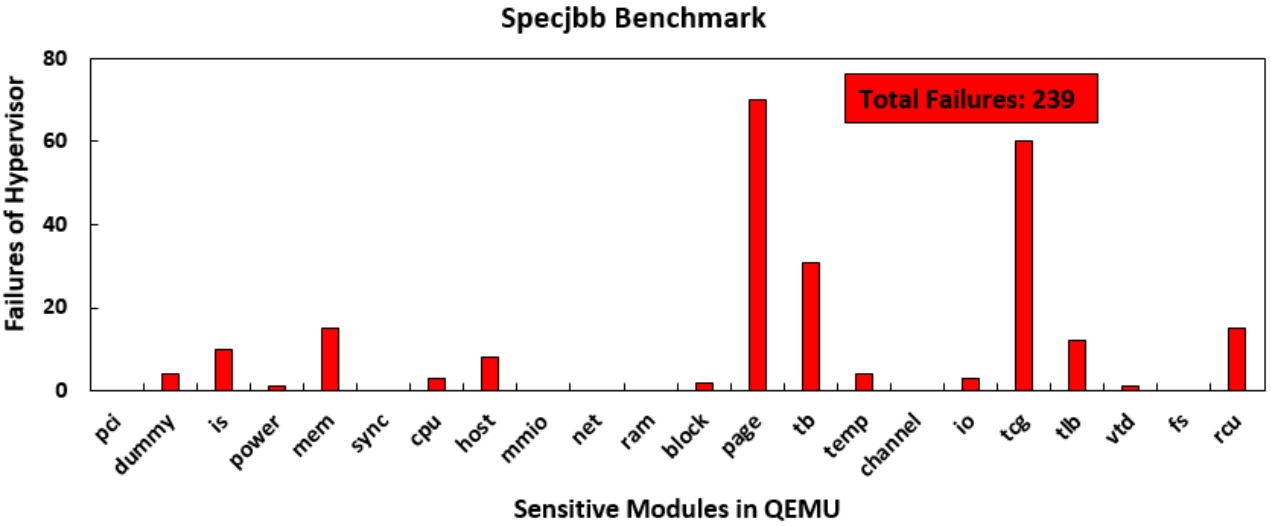

Figure 15: Failures of QEMU Modules in Specjbb Benchmark

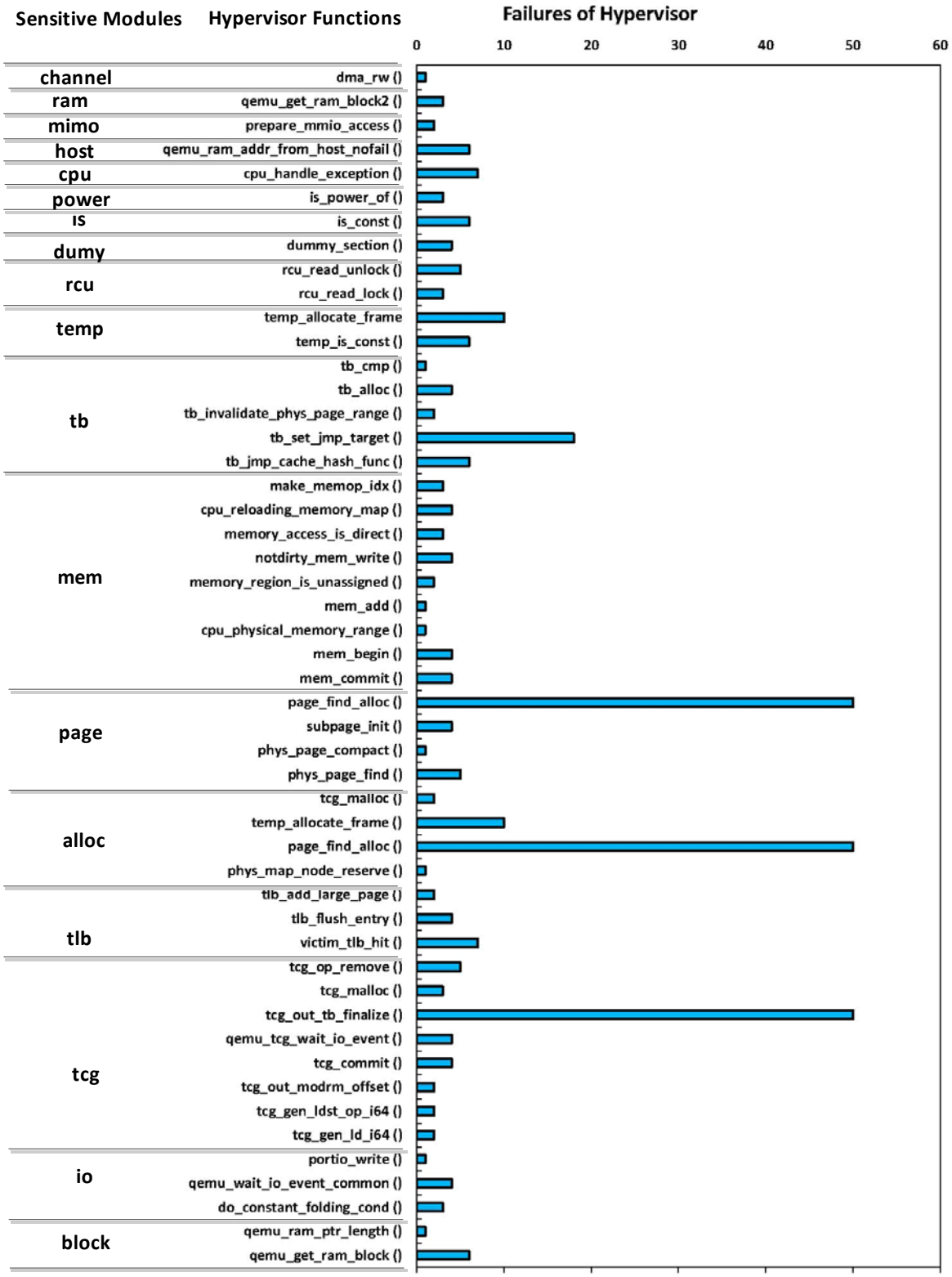

Figure 16: Hyervisor Function Failures in Memcached Benchmark

Figure 15 and Figure 17 show the distribution of failures between different functions when running the Specjbb benchmark. For experiments with this benchmark, we create 2 JVMs inside the guest virtual machine and setup the warehouse threshold as 1 minute.

We found that *rcu* failures are higher than the previous two benchmarks. That means Specjbb JVM needs more read-copy update [2] for communication.

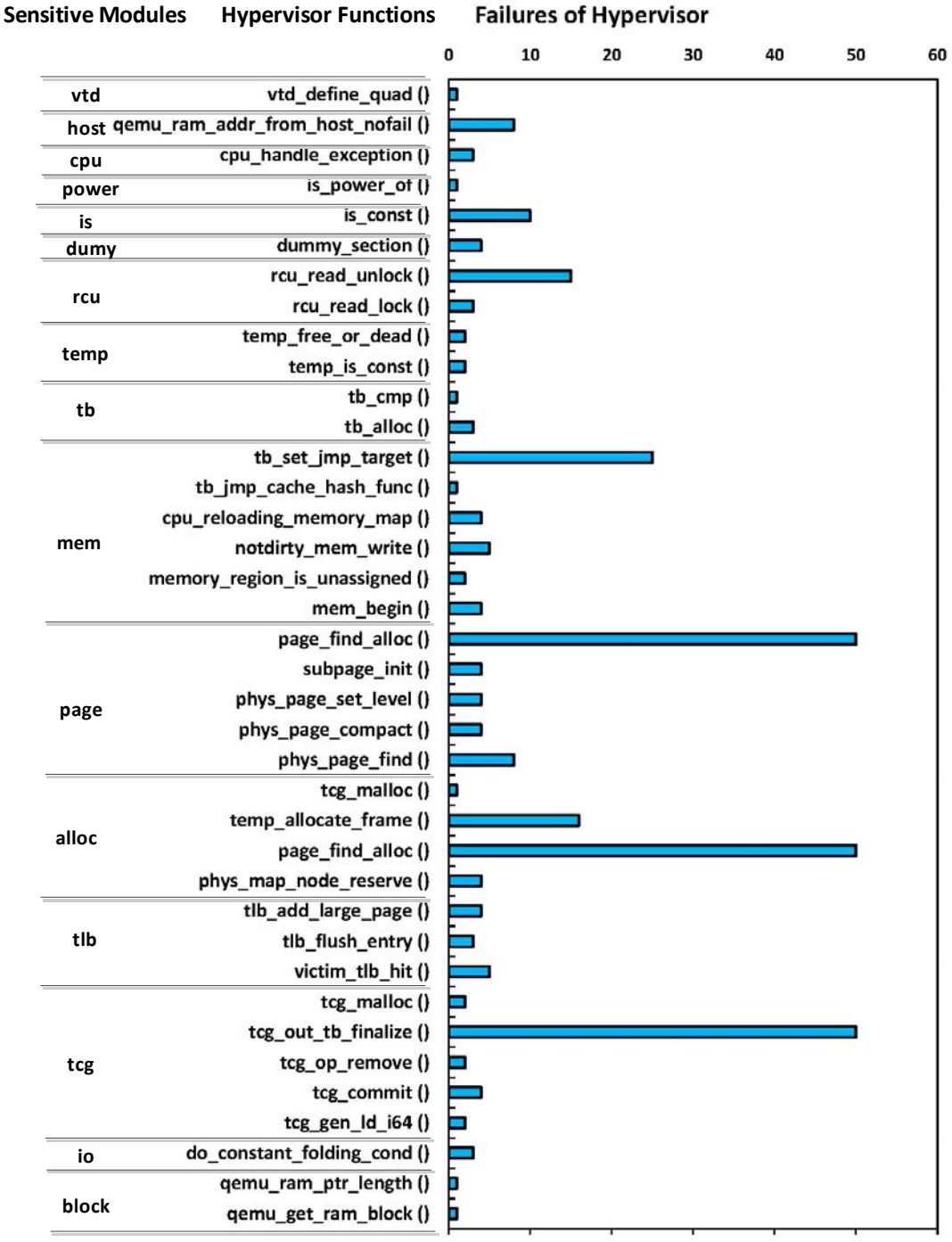

Figure 17: Hypervisor Function Failures in Specjbb Benchmark

Through the testing of the three benchmarks, we can conclude some analysis as follows:

- The hypervisor crashes when errors are injected in *tcg* and *tb* modules regardless of the running benchmark.
- The behavior of hypervisor varies between different benchmarks if an error is injected in other modules, such as mem and page: for some benchmarks, we can observe hypervisor crashes, while other benchmarks do not trigger any crashes.
- Modules of the QEMU hypervisor could be selectively protected.

3. Error Resilient Hypervisor Implementation – Storage

As we discussed in the previous section, some data structures may be more sensitive to errors and may cause system crashes more easily than others. A failure in a critical data structure can cause a complete system crash. For example, in Section 2, we already provide a sensitivity characterization of hypervisor structures. When we inject errors to sensitive structures, QEMU and KVM are significantly more prone to crashes. This section focuses on the implementation of a heterogeneous reliability memory scheme that is composed of a fully reliable and a less-reliable memory domain. In addition, we discuss the modifications at the system software that enable memory domains with different reliability and storage of critical structures on the reliable domain and the less critical data on the less reliable domain

Our current implementation, based on the APM Linux kernel, is a software-based approach that implements a heterogeneous memory architecture by adopting independent memory zones with different reliability. When the QEMU or the KVM requests memory with high reliability, the syscall *mmap* or *alloc_page* will allocate the space and map it to a reliable domain. Through this, we can selectively protect the sensitive data structures of the hypervisor. In this section, firstly, we introduce the concept of the heterogeneous reliability memory domains, and then we provide implementation details about the memory allocation syscalls. Finally, we enable QEMU to support the heterogeneous memory zones.

3.1. Heterogeneous Reliability Memory

We introduced a new zone, *ZONE_UNREL*, and modified the zone initialization procedure to split all PFNs (Page Frame Number) into two groups [10]. The first group, called reliable domain is used for *ZONE_NORMAL* and *ZONE_DMA* and the second group called less reliable domain is used for *ZONE_UNREL*. We note that the first group contains PFNs which correspond to reliable DIMMs operated at the nominal voltage and refresh rate levels, while PFNs belonging to the second group address unreliable DIMMs operated at the marginal refresh rate and voltage level. Memory from the unreliable zone is allocated through *mmap* system call which uses memory from *ZONE_UNREL* if the user has set a specific flag. Thus, we can operate memory segments at the marginal refresh rate and voltage levels to reduce power consumption without significant modifications of Linux and other system software, since these segments are not available for allocation until it is explicitly requested.

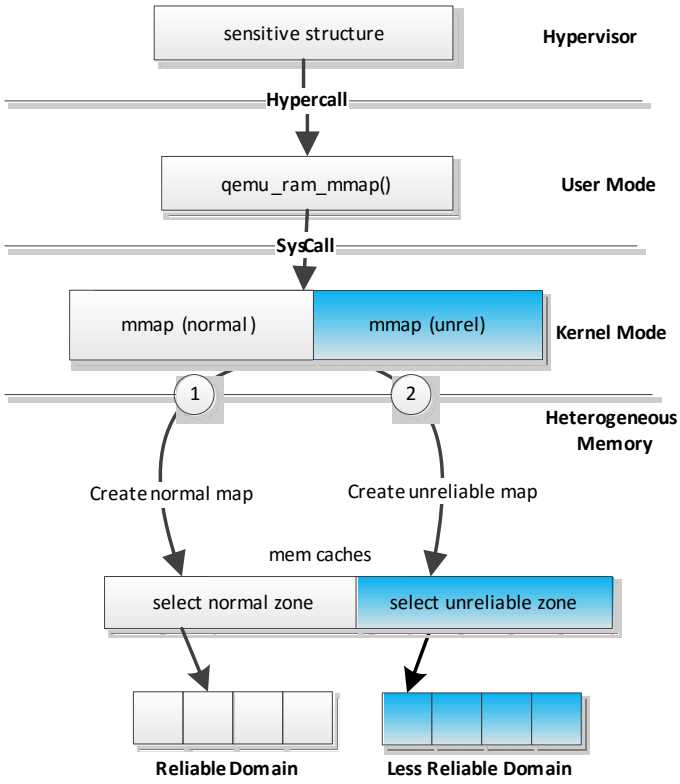

Figure 18: Heterogeneous Reliability Memory

Figure 18 shows the workflow of memory allocation for both reliable and less reliable domains. In this figure, the reliable domain includes normal zone and DMA zone, while the less reliable domain is created for unreliable DIMMs. We use the following steps to revise the UniServer kernel (APM kernel) for heterogeneous reliability memory:

- *Step 1 Layout and flags:* We create the less reliable domain in the kernel and support to enable it with a flag.
- *Step 2 Backend map extension*: We initialize the value of these flags and also enable syscall *mmap* to create map from userspace to unreliable zone.
- *Step 3 Page allocator extension*: We revise the syscall *alloc_page* and define new flags for unreliable zone.
- *Step 4 Hypervisor for heterogeneous reliability memory*: We revise QEMU related hypervisor using memory-backend approach, so that when QEMU creates memory area it can access unreliable domain.

In Section 2, we provided the results of characterization showing how hypervisor structures are sensitive to errors. As we can identify virtual addresses of all sensitive structures, we are able to map such structures to the reliable memory domain. Such mapping is one way to enable fault-resiliency in the hypervisor which is the main target of our research.

3.2. Layout and Flags

Figure 19 shows the definition of heterogeneous reliability memory. First, the function *zone_sizes_init* creates the memory layout supporting extended memory domain (see its source code in Figure 20). Typically, the lowest region is the DMA zone, while the remaining is the normal zone and we separate a part of normal zone to the less reliable zone. In this figure, all the zones directly communicate with MCBs. For example, DMA zone and normal zone direct the hardware memory layout to the first Memory Controller Block (*MCB0)*, while unreliable zone direct the layout to *MCB1* which is regarded as less reliable domain.

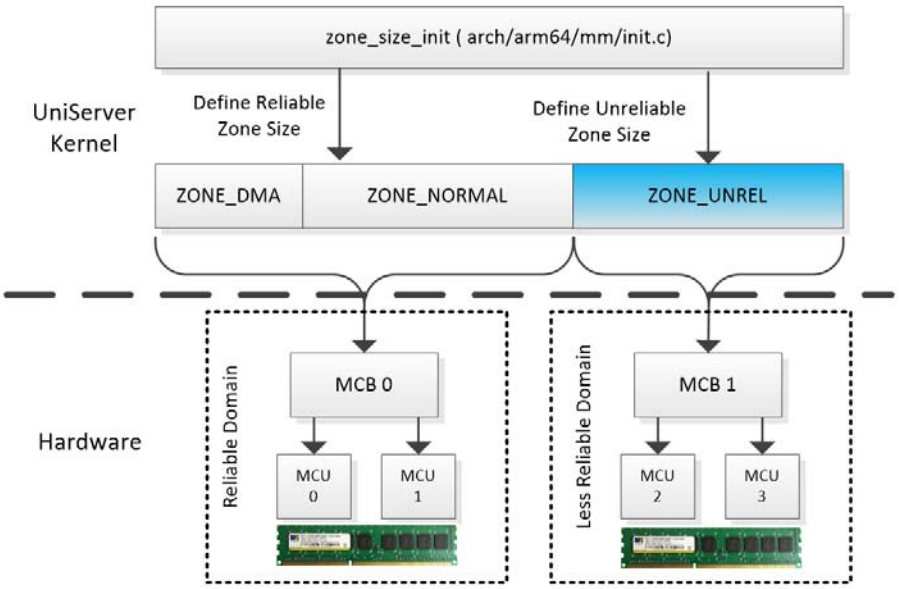

Figure 19: Definition of Heterogeneous Reliability Memory

Then during the boot time, physical memory is initialized following the configuration of zone size. Finally, *start_zone_unrel ()* (in Figure 20) will return the address of unreliable zone.

```
/** In the UniServer kernel arch/arm64/mm/init.c function is zone_sizes_init  **/
       min_unrel = PFN_DOWN(start_zone_unrel());
        if (min_unrel < max_dma)
                min_unrel = max_dma;
        zone_size[ZONE_NORMAL] = min_unrel - max_dma;
        pr_warn("Normal: %lu - %lu (size: %luGB)\n", max_dma, min_unrel,
                zone_size[ZONE_NORMAL] >> (30 - PAGE_SHIFT));
        zone_size[ZONE_UNREL] = max - min_unrel;
        zone_size[ZONE_NORMAL] = max - max_dma;
        pr_warn("Normal: %lu - %lu (size: %luGB)\n", max_dma, max,
                zone_size[ZONE_NORMAL] >> (30 - PAGE_SHIFT));
```

Figure 20: Memory Region of Heterogeneous Reliability Memory

To expose the different domains of our scheme to the user, we have implemented an API for requesting memory from the less reliable domain. Figure 21, shows the additions in *include/Linux/gfp.h*, defining the necessary flags.

```
/** In UniServer Kernel include/Linux/gfp.h, we define kernel flags to support unreliable
zone **/
#define ___GFP_UNREL          0x10u
#define __GFP_UNREL   ((__force gfp_t)___GFP_UNREL) /* Page comes from less reliable
domain */
#define GFP_ZONEMASK  (__GFP_DMA|__GFP_HIGHMEM|__GFP_DMA32|__GFP_MOVABLE|__GFP_UNREL)
#define GFP_UNREL     __GFP_UNREL
+#ifdef CONFIG_ZONE_UNREL
#define OPT_ZONE_UNREL ZONE_UNREL
#else
#define OPT_ZONE_UNREL ZONE_NORMAL
#endif
static inline enum zone_type gfp_zone(gfp_t flags)
{
     if (bit & __GFP_UNREL)          return ZONE_UNREL;
}
```

Figure 21: Basic Kernel Flags for Heterogeneous Reliability Memory

We modify *zone_type gfp_zone()* to force the bit check, because adding *ZONE_UNREL* overflows *GFP_ZONE_TABLE*. The detailed workflow of the implemented scheme can be seen in Figure 22. *GFP_UNREL* is set to support kernel level allocator such as *kmalloc*.

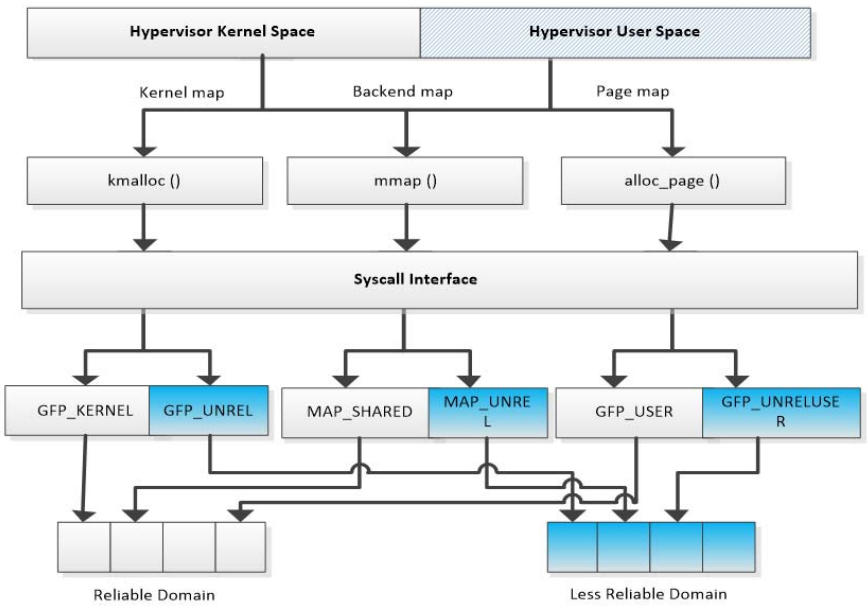

Figure 22: Syscall Implementation for Heterogeneous Reliability Memory

3.3. Backend Map Extension

Figure 23 shows implementation of *mmap* being used for QEMU to map the guest RAM to the host file both in private map and anonymous map. We also try to use the less reliable domain in *mmap* function thus we define the threshold in our architecture (Aarch64). However, this value could be changed for other machines. In UniServer, the mmap syscall can be used to directly map virtual space to the heterogeneous memory.

```
/** In the function include/Linux/mm.h, we define the access threshold address **/
#if defined(CONFIG_ZONE_UNREL)
#define VM_UNREL      value   /* Define the threshold of unreliable domain */
#endif

/**** In the function mm/memory.c, we add the support for mmap syscall ****/
#ifdef CONFIG_ZONE_UNREL
        if (flags & MAP_UNREL) {
                pr_info("%s: Setting up VM_UNREL flag\n", __func__);
                vm_flags |= VM_UNREL;
        }

#endif

/** In include/uapi/asm-generic/mman-common.h, we add flag for mmap so that it can map to
unreliable domain **/

#ifdef CONFIG_ZONE_UNREL
#define MAP_UNREL     0x40           /* Mem from ZONE_UNREL can be used */
endif
```

Figure 23: Implementation for *mmap()*

3.4. Page Allocator Extension

Syscall *alloc_page()* facilitates frequent allocations and deallocations of data, programmers often introduce free lists. It can create the page level map when the new allocation is ready. Since it can be frequently called when kernel allocate memory to the application, we minor revise *alloc_page* function when QEMU workload requests the page level allocation.

We have created the flag *GFP_UNRELUSER* as *GFP_USER* | *GFP_UNREL*, thus we can force *alloc_page()* to map pages to the less reliable domain, as shown by the changes in the code of the implementation of *alloc_page()* in Figure 24.

```
/** mm/page_alloc.c **/
struct page *__alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
                        struct zonelist *zonelist, nodemask_t *nodemask)
{
#ifdef CONFIG_ZONE_UNREL
        if (ac.high_zoneidx == ZONE_UNREL) {
                alloc_flags |= ALLOC_NO_WATERMARKS;
                alloc_flags &= !ALLOC_FAIR;
        }
#endif
}

static char * const zone_names[MAX_NR_ZONES] = {
#ifdef CONFIG_ZONE_UNREL
         "Rel",
#endif

/** include/Linux/gfp.h **/

#define GFP_UNRELUSER (GFP_USER | __GFP_UNREL)

/** mm/memory.c **/
+#ifdef CONFIG_ZONE_UNREL
       if (vma->vm_flags & VM_UNREL) {
               pr_debug("%s: Trying to allocate page from ZONE_UNREL\n",
__func__);
               page = alloc_page(GFP_UNRELUSER);
       } else
               page = alloc_zeroed_user_highpage_movable(vma, address);
#else
```

Figure 24: Implementation for *alloc_page()*

3.5. Hypervisor Using Heterogeneous Reliability Memory

To support heterogeneous reliability memory, we revise the QEMU hypervisor. Our revision rationale is: when QEMU tries to allocate memory for a KVM, we can choose on-the-fly the mapping to which reliability zone it will end up using. If we identify all the hypercalls from QEMU hypervisor to Kernel syscall *mmap* and trap them, we can modify those hypercalls and further control the map to heterogeneous reliability memory. In the following section, we will detail the QEMU allocation mechanism and how to revise the allocation hypercall for our purpose.

For the QEMU and KVM, each memory region is a range of host memory that is available to the guest. Accesses into the region do map directly to host memory. In general, KVM memory could be created through two approaches: 1) QEMU Malloc, or 2) Memory backend. This subsection will describe the approaches.

- *QEMU Malloc*: QEMU process runs mostly like a normal Linux program. It allocates its memory with normal *malloc()* or *mmap()* calls. It will not be actually allocated until the first time it is touched. Once the guest is running, the KVM provides a part of the *malloc()*'d memory area as being the physical memory of the guest.
- *Memory Backend*: Memory-backend device (*backends/hostmem.c*) contains the actual host memory that backs guest RAM [11]. It includes anonymous mmap and file-backed mmap. An anonymous mmap is used if the memory space could be shared with many applications, while file-backed mmap is used when a file area is regarded as guest physical memory.

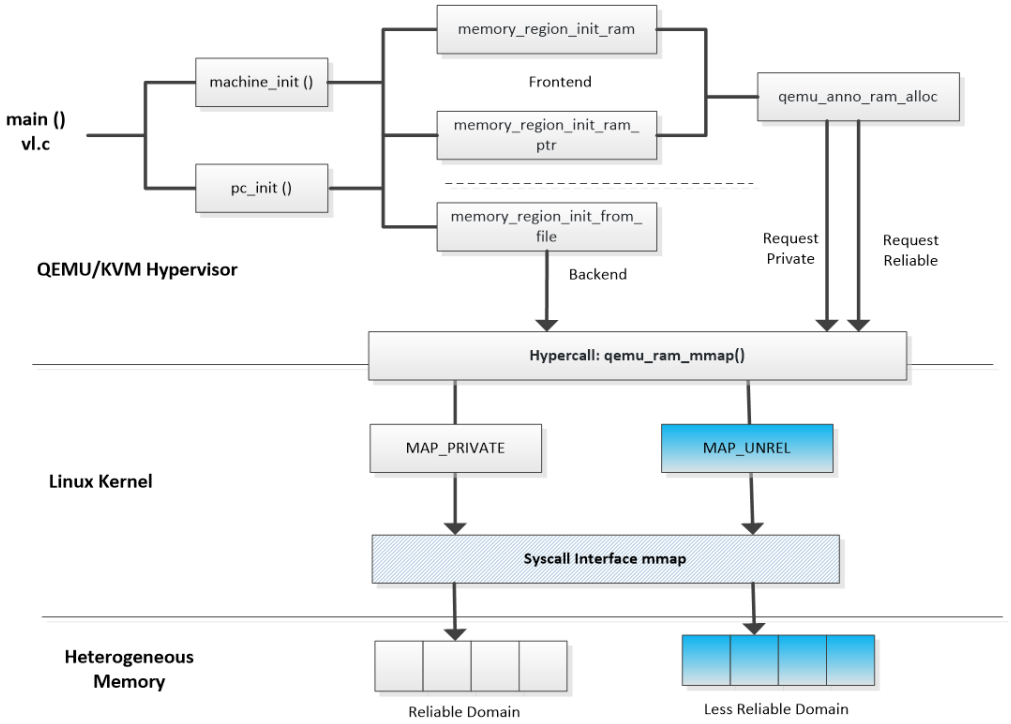

Figure 25: Architecture of Error-Resilient Hypervisor

We revise the QEMU APIs to support the reliability domains. As an overview, Figure 25 details the architecture of error resilient hypervisor of our current implementation. If QEMU creates memory area using memory backend, guest RAM is initialized by *memory_region_init_ram()* [12]. We can also use hypercalls, such as *memory_region_init_resizeable_ram()*, *memory_region_init_ram_ptr()*, or *memory_region_init_from_file()* to create memory for special use-cases. Memory from both frontend and backend are finally mapped through *qemu_ram_alloc()* (*exec.c*).

```
/** Define the unreliable map in QEMU hypervisor to support mmap syscall **/

#ifndef MAP_UNRELIABLE
#define MAP_UNRELIABLE MAP_UNREL // define MAP_UNRELIABLE in the QEMU
```

Figure 26: Flag Setup of QEMU Hypervisor

The two APIs will call *qemu_ram_mmap()* (*util/mmap-alloc.c*) to create mapping to host memory region. We revise this hypercall to support *MAP_UNREL* flag in the host server. To revise the QEMU for less reliable domain, we define the flags in Figure 26 (*include/qemu/osdep.h*).

```
/** Add the support of map unreliable in QEMU hypervisor (util/mmap-alloc.c) **/
#if defined(__powerpc64__) && defined(__Linux__)
       int anonfd = fd == -1 || qemu_fd_getpagesize(fd) == getpagesize() ? -1 :  fd;
       int flags = anonfd == -1 ? MAP_ANONYMOUS : MAP_NORESERVE;
       void *ptr = mmap(0, total, PROT_NONE, flags | MAP_PRIVATE | MAP_UNRELIABLE, anonfd,
0);
#else
       void *ptr = mmap(0, total, PROT_NONE, MAP_ANONYMOUS | MAP_PRIVATE |
       MAP_UNRELIABLE, -1, 0);
```

Figure 27: Implementation of mmap Entry for Heterogeneous Reliability Memory (*MAP_UNRELIABLE*)

Figure 27 shows the *mmap* entry implemented in QEMU. In the flag we add the *MAP_UNREL* when *mmap()* function checks RAM zone using either anonymous flag or no reserve flag. Thus, when QEMU initializes guest RAM, the entry (start address) of RAM region will be mapped to the less reliable domain.

```
  /** Add the support of map unreliable in QEMU hypervisor (util/mmap-alloc.c) **/
    size_t offset = QEMU_ALIGN_UP((uintptr_t) ptr, align) - (uintptr_t)ptr;
    void *ptr1;
    ptr1 = mmap(ptr + offset, size, PROT_READ | PROT_WRITE,
                MAP_FIXED | MAP_UNRELIABLE |
                (fd == -1 ? MAP_ANONYMOUS : 0) |
                (shared ? MAP_SHARED : MAP_PRIVATE),
                fd, 0);
    if (ptr1 == MAP_FAILED) {
            munmap(ptr, total);
           return MAP_FAILED;
    }
    ptr += offset;
    total -= offset;
```

Figure 28: Implementation of *mmap* to Normal Position for Heterogeneous Reliability Memory (*MAP_UNRELIABLE*)

Figure 28 shows the normal position *mmap* in our implementation. The normal position does not begin with ptr entrance. At the beginning, we use *QEMU_ALIGN_UP()* to do the page align for the offset. *ptr* is the entry mapped to the host less reliable domain. In this function, we map the target position of the guest RAM to the host less reliable domain when QEMU decides whether the flag is *MAP_SHARED* or *MAP_PRIVATE*. After that, we revise the callers of *qemu_ram_mmap()*: *flie_ram_alloc()* and *qemu_anno_ram_alloc()* (Figure 29).

```
#ifdef RELIABLE_MAP
    area = qemu_ram_mmap(fd, memory, block->mr->align,
                         block->flags & RAM_SHARED);
    if (area == MAP_FAILED) {
        error_setg_errno(errp, errno,
                         "unable to map backing store for guest RAM");
        goto error;
   }
#endif


void *qemu_anon_ram_alloc(size_t size, uint64_t *alignment)
{
  size_t align = QEMU_VMALLOC_ALIGN;
#ifdef RELIABLE_MAP
  void *ptr = qemu_ram_mmap(-1, size, align, false);
  if (ptr == MAP_FAILED) {
        return NULL;
  }
  if (alignment) {
        *alignment = align;
  }
  trace_qemu_anon_ram_alloc(size, ptr);
  return ptr;
#endif
```

Figure 29: Support for Less Reliable Domain in Caller of *qemu_ram_mmap()*

Through those modifications, we create memory allocations in QEMU following the example in Figure 30. The flag "–m 1G" enables 1 GB of main memory for the QEMU based on the malloc approach in the reliable zone. Furthermore, the two following flags: "-object memory-backend-file", enables the QEMU to use memory from the less reliable domain and "-device pc-dimm", maps the allocated memory to correspond as a DIMM in the QEMU virtual machine. Through this simple example, users can utilize the heterogeneous reliability memory for different use-cases.

```
/** Memory backend creation for less heterogeneous reliability memory, we can create the
normal dimm for reliable memory, and we choose the command memory-backend-file for the less
reliable memory **/

qemu [...] \
   -m 1G,slots=2,maxmem=2G \
   -object memory-backend-file,id=mem1,size=1G,mem-path=/dev/mem_unrel_1 \
   -device pc-dimm,id=dimm1,memdev=mem1
```

Figure 30: QEMU Commands of Backend Mode to Create KVM

In my previous idea, we hope to put the whole QEMU (userspace hypervisor) and VM memory space into the reliable domain. However, if we put all the VM and hypervisor to the reliable domain on the fly, it is not a space-efficient approach because we should backup the whole data. We will discuss the solution in the future work (Section 6).

4. Error Resilient Hypervisor – Execution

The primary objective of system software is to self-protect against failures and on top of that to provide transparent protection to the applications whenever possible. The hypervisor should minimize the critical system code that executes on relaxed cores whenever possible. As a rule of thumb, we initially consider all code that runs in kernel-mode and manages the KVM state as critical and user-mode code i.e. applications to be non-critical.

Virtualization support is closely integrated in the Linux kernel. The KVM subsystem of Linux is responsible for the management of virtualized hardware, especially virtual CPUs. A virtual CPU is a data structure that holds the CPU state of the hardware CPU when running a virtual machine on it. This resembles a lot the purpose of the *task_struct* data structure of Linux that maintains the state of each thread on the system. Linux takes advantage of this similarity and manages each virtual CPU as a separate process with all the benefits of the latter, such as scheduling, migration, pinning, grouping etc. Under this abstraction, virtual CPUs and by extension virtual machines are just user applications, therefore consist of non-critical code according to our definition above.

A failure in a critical code path can cause a complete system crash. On the other hand, a failure in a non-critical code path is usually constrained to just one or a few related applications with rather limited propagation probability to other parts of the system. This section focuses on mechanisms that can relieve the relaxed cores of the system from executing critical code and hand it over to one of the reliable cores in order to minimize the probability of catastrophic errors at the system level. The actual use and optimization of the mechanisms is left to the policy manager for better flexibility.

This reliable / relaxed scheme breaks the Symmetric multiprocessing (SMP) assumptions of the traditional Linux kernel, where all processors are treated as equal. Our approach essentially downgrades the relaxed CPUs to thin clients of the reliable ones, resulting in higher resilience. In this deliverable, we focus mostly on the functionality of the mechanisms and their contribution to an error resilient hypervisor, keeping any optimizations as an improvement for Deliverable D5.4.

Whenever possible our design and implementation opts to use / extend existing features of Linux / KVM kernel, rather introducing new ones. The main points of interest are the entry points from non-critical to critical code, namely the system calls, interrupts and page faults. Considering virtualization support, hypercalls are another common entry point from guest-mode (non-critical) to host-mode (critical). The hypercall mechanism is used by the guest operating system to request attention from the hypervisor. According to *Documentation/virtual/kvm/hypercalls.txt* of the Linux documentation, KVM does not provide any hypercalls for the Aarch64 architecture. Nevertheless, critical parts of hypercalls can be protected – on any architecture – in the same way as system calls. Another subsystem of the hypervisor that is frequently executed and prone to crash the system in case of failure is the scheduler. The next chapters discuss the existing infrastructure, the architecture of our approach and provides technical insight for each one of the aforementioned points of interest.

4.1. Interrupt Handlers

Hardware interrupt handlers reside in the lowest-level of the system software stack and play a significant role for the stability and performance of the system. They are responsible for CPU inter-communication and for the communication of a CPU with the rest of the peripheral devices. They can be triggered to execute at almost any point and disrupt the normal execution flow of applications, or even parts of kernel code, to serve a pending request that needs immediate attention.

The hardware device responsible for handling and delivering interrupts to CPUs is called interrupt controller. When a CPU is operating at extended margins, thus more likely to suffer errors, it is better for the operating

system to exclude it from running any interrupt handlers. The interrupt controller in most systems is a programmable component that can selectively deliver incoming interrupts to specific CPUs. Under our error resilient hypervisor, these would be the reliable CPUs whereas the relaxed ones should never receive any interrupts. However, this interrupt rerouting is not possible for all types of interrupts, either due to architectural reasons or to ensure correct system behavior. Non-routable interrupts are timer interrupts that drive the scheduler tick, perf events and time-related per CPU statistics such as system load.

We reroute all routable interrupts using the *irq_set_affinity(irq, mask)* function, which changes the CPU mask for the specified interrupt number to a CPU mask that contains only reliable CPUs. Figure 31 outlines the code that performs this change.

```
        for_each_irq(irq)
                if (irq_can_set_affinity(irq))
                        irq_set_affinity(irq, reliable_cpu_mask);
```

Figure 31: Pin Interrupts to Reliable CPUs

On a very active system with high load that generates many interrupts, this may create a bottleneck on the reliable CPUs. In such a case, a policy manager may decide to increase the number of reliable CPUs by restoring a relaxed one to its nominal configuration. Alternatively, it may select to exclude some interrupts / cores from re-routing, according to the criticality of individual types of interrupts and the relative error resilience of cores.

4.2. System Calls

The system call mechanism is the most common path used by applications to request services from the operating system. It is very similar to the user-level function call mechanism, but with an important difference: it crosses the user-kernel boundary and executes privileged code. Most of the system calls take some user input and operate on critical system data structures in a controlled fashion. By nature, system calls are prone to propagating hardware errors isolated to one application to the whole system, as many system data structures invisible to user are shared among processes and could be touched by system calls.

For an application that executes on a relaxed core, migrating each system call to reliable cores is a technique that minimizes the chances of introduction and propagation of such errors, although it does not completely eliminate it: there is a short "gray" time window when migration takes place, during which code is executed on the relaxed core. To the best of our knowledge, system call migration is a novel approach to system reliability and according to our experiments so far it is effective and worth of further evaluation and extension.

Linux creates a separate wrapper for each system call through a series of macros in the system call's definition. These wrappers just encapsulate the actual call and provide a consistent naming pattern with the prefix *"sys_"* for all system calls. Without delving into distracting implementation details, each time an application requests a service through a system call, the kernel performs the following pattern:

1. Application - Request a service through a system call
2. Kernel - Switch from user- to kernel-mode, save user state
3. Kernel - Call the appropriate system call wrapper
4. Kernel - Switch from kernel- to user-mode, restore user state
5. Application - Obtain result, continue execution

The switch part and call of the wrapper (steps 2-4) are architecture-dependent and implemented directly in assembly. Working at this level requires concrete understanding of the target architecture and how the

hardware works. Nevertheless, it gives the programmer the flexibility to fully control the migration and shrink the time window between the transitions from relaxed to reliable core and vice-versa.

On the other hand, the system call wrapper is architecture independent and written in C. This seems the ideal place to decide which core is going to execute the requested system call. To achieve the migration, inside the wrapper we manipulate the core affinity of the process right before and after the execution of the system call handler. Linux already provides a way to specify which cores a task is allowed to execute on through the *sched_setaffinity()* system call. Each core has its own migration thread pinned to it and stopped. The migration threads are special processes that migrate tasks between cores. When a core decides to move a task to another core, wakes up its migration thread to perform the actual work with high priority. After the migration is complete, the migration thread stops.

The *sched_setaffinity()* function takes as parameter a CPU mask with the new allowed cores and ensures that the mask is valid for the current task. If the current task is currently attached on a core not included in the new CPU mask, it gets stopped if running. Then the *cpus_allowed* attribute of the task changes accordingly and the migration thread of this core takes over and moves the task to a suitable destination core. It is up to the scheduler to decide whether to rerun the task immediately or have it wait for its turn. If the current core of the task belongs to the new CPU mask, the aforementioned procedure is simplified. The running state of the task remains unchanged and the code merely adjusts the *cpus_allowed* attribute of the task.

Having this functionality available, we have created two hook functions with a modified version of *sched_setaffinity()* that skip some sanity checks for the new CPU mask, since the input is not provided by the user but from the kernel itself and can be trusted. Next, we modify the system call wrapper to call these hooks. Our modified version is described in Figure 32. The listing below highlights just the system call wrapper, with our hook functions depicted in bold:

1. System call enter hook
2. **Call the actual implementation**
3. Perform some sanity checks
4. **System call exit hook**
5. Return result

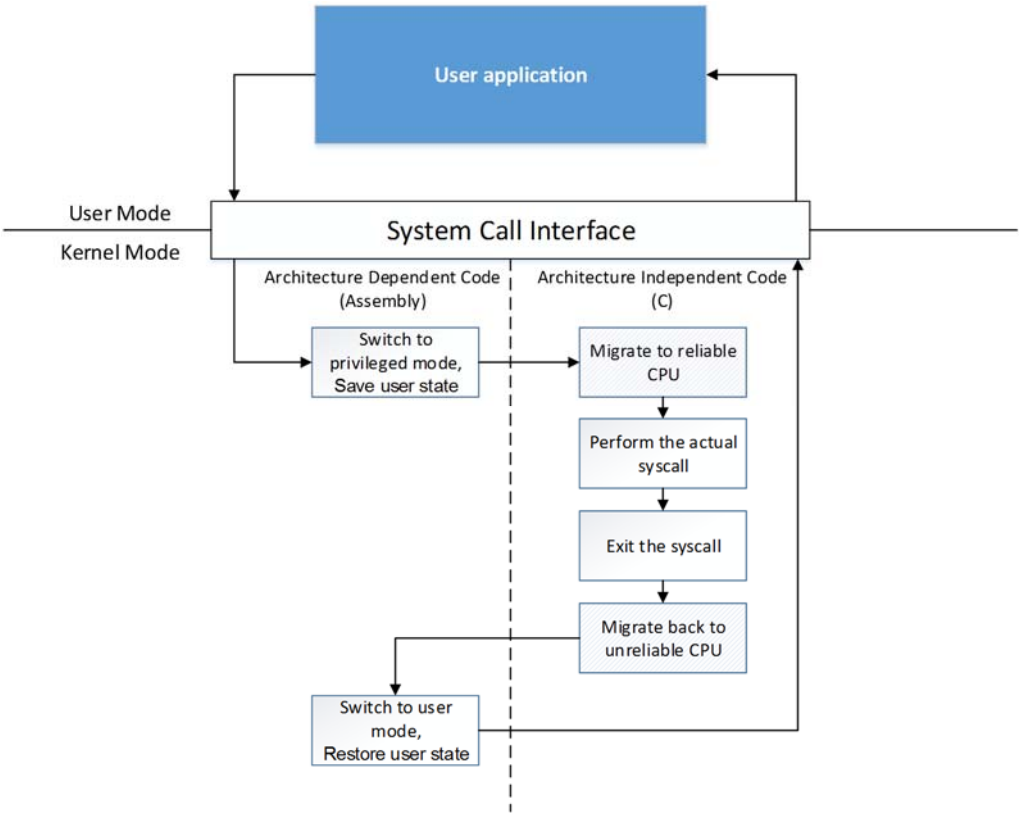

Figure 32: Modified System Call Diagram

The first hook restricts the CPU affinity of the task to reliable CPUs only, where its counterpart at the end restores the previous CPU affinity, which may contain relaxed cores as well. We had to explicitly exclude the *sched_setaffinity()* system call from the migration process, as it is part of the implementation and trying to migrate it as well would result in an infinite loop.

4.3. Page Fault Handler

The page fault handler is the most common asynchronous entry point to critical code. The page fault handler is responsible for detecting the cause of a missing translation of a virtual address, fetch any missing data to memory and update the necessary page tables of the interrupted process. After that, the process can continue its execution as expected. The page table structures of a process maintain the mapping from virtual to physical pages. Therefore, the page fault handler that modifies those structures can introduce security- apart from error resilience-issues.

The function *do_page_fault()* is the main entry point of the page fault handler. After some initial checks, the most common code path calls the core function *handle_mm_fault()* which in turn calls *__handle_mm_fault()* to resolve the missing mapping and allow the application to continue. We restrict the execution of the latter function to reliable CPUs to avoid the corruption of page table entries. Figure 33 shows a simplified, pruned (mainly for visibility reasons) function graph of the page fault handler, indicating which part of the graph is being migrated. Apart from a few technical issues regarding the interrupt handling mechanism in general, we have chosen to protect this core function instead of the *do_page_fault()* entry point mainly because *handle_mm_fault()* is called from other points inside the kernel as well, and we want to cover them too. Additionally, focusing on *handle_mm_fault()* keeps the implementation architecture independent.

The migration mechanism for page fault handlers is different from the system call migration approach. We reuse the workqueue mechanism to offload the execution of *__handle_mm_fault()* from relaxed cores to a reliable one. Workqueues perform deferred execution of every committed work in their list and can be bound to one or more CPUs. The workqueue mechanism is supported by worker threads that undertake the actual execution of the queued work items.

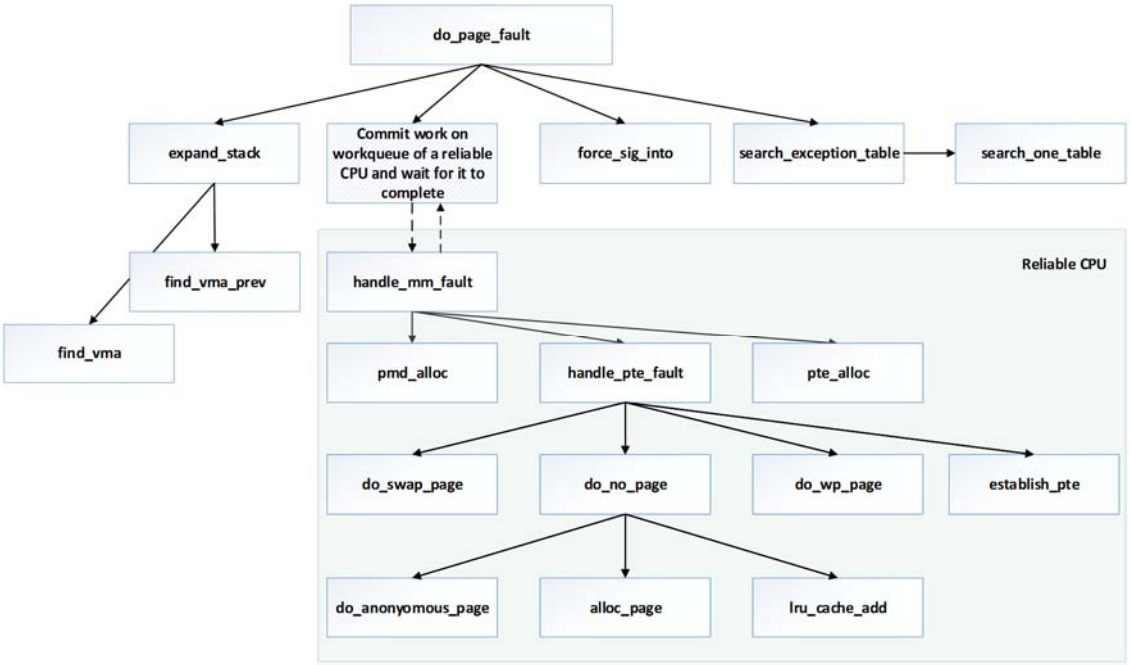

Figure 33: The Core Code of the Page Fault Handler

The execution of the page fault handler on a relaxed CPU looks like this:

1. Application - Page fault exception, pause execution
2. Kernel - Jump to entry point *do_page_fault()*
3. Kernel - Commit *__handle_mm_fault()* on a reliable CPU
4. Kernel - Wait for the committed work to complete.
5. Kernel - Exit the page fault handler
6. Application - Resume execution

4.4. Scheduler

The task scheduler of the OS is the component responsible for the fair distribution, execution and management of all tasks / processes present in the system. It is also responsible for load balancing between CPUs. When the scheduler detects an idle CPU, it tries to offload some tasks from a busy one. Each task is associated with one CPU at any time, regardless of the task's CPU affinity stored in the *cpus_allowed* attribute controlled by the *sched_setaffinity()* system call.

Every CPU has its own runqueue of runnable tasks. The scheduler at specific intervals, or whenever triggered asynchronously, picks the next task to execute. Although the selection of the next task depends on the priority and policy class of each task (fair, deadline, realtime), the core scheduling algorithm is common for all policies. In order to provide an error resilient scheduler, we focus on this backbone code.

The scheduler code is triggered either on each clock tick or programmatically. On each CPU, the timer interrupt handler calls the *scheduler_tick()* function. This function is responsible for updating the time statistics of the

runqueue and the currently running task. It also sets the task reschedule flag if necessary, flagging that the scheduler must be executed to pick a next task to run as soon as possible. The *schedule()* function on the other hand is called across the kernel whenever the current task of a CPU either has exceeded its time share or is waiting for a resource or is about to block due to I/O operations or to be terminated.

Each CPU executes the scheduling algorithm for its own runqueue. The core implementation is in the *__schedule()* function. As the code of the scheduler manipulates critical data for the proper operation and behaviour of the system, the goal is to make the reliable CPUs execute the scheduler on behalf of the relaxed CPUs as well.

The first step is to isolate the relaxed CPUs from the load balancing mechanism. This means that the only way to execute tasks on isolated CPUs is to explicitly set the task affinity to include them. We can achieve the aforementioned isolation with the provided *isolcpus* command line option available in Linux. The user provides, at boot time, the ranges or number of CPUs she wishes to exclude from the load balancing functionality. For example, on a system with eight CPUs the "isolcpus=2-7" parameter restricts the scheduler to automatically load-balance only among CPUs 0 and 1. The only downside at this point is that this information is fixed at boot time and cannot dynamically change at execution time. We aim to provide such a functionality in the future, if required by the project.

In the current SMP design of Linux there are parts of the code that manipulate per-CPU data only, assuming none of the other CPUs changes them. Moreover, each CPU assumes that it takes its own decisions. Therefore, the second step is to make the scheduler code capable of running on one CPU and process data that belongs to another CPU, as well as to make the code aware of the distinction between reliable and relaxed CPUs. We changed the prototypes of critical functions to take the CPU number as an extra parameter and their implementations so that they can work with the correct per-CPU data. The extent of the changes is relatively limited. Figure 34 shows the functions and their call sites that have been refactored for the *scheduler_tick()* migration, whereas Figure 35 shows the high level flow chart of the reliable scheduler.

```
void sched_clock_tick(void) {…}
void scheduler_tick(void) {…}
void set_cpu_sd_state_busy(void) {…}
```

Figure 34: Functions That Need Refactoring for the *scheduler_tick()* Migration

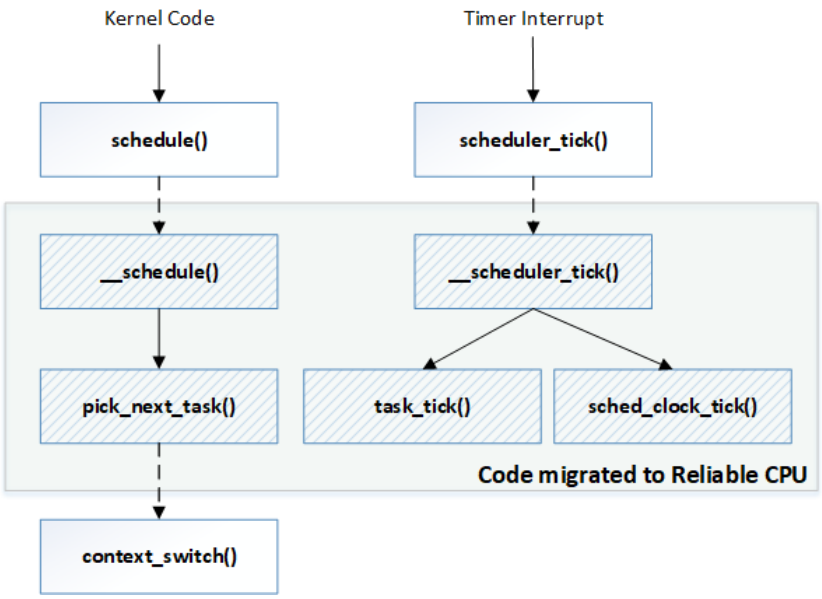

Figure 35: Flow Chart of the Reliable Scheduler

The *scheduler_tick()* on the relaxed CPUs skips the time statistics update of the task and runqueue. On the contrary, the reliable CPUs update their own time statistics plus the necessary time statistics for the relaxed CPUs. Figure 36 shows the code segment that runs the *scheduler_tick()* on the reliable CPUs on behalf of the relaxed CPUs.

```
if (is_reliable_cpu(cpu))
        __schedule_tick(cpu);          /* update this CPU */

if (cpu == first_reliable_cpu())
        for_each_reliable_cpu(i)
                __schedule_tick(i);     /* update each relaxed CPU */
```

Figure 36: The *scheduler_tick()* Code Refactoring

For the *__schedule()* function, the rationale and current approach is very similar to *scheduler_tick()*. The first reliable CPU selects the next task for each relaxed CPU and stores it in a new per-CPU variable *next_task*. The next time the relaxed CPU calls *schedule()*, it just reads its *next_task* and performs the context switch. Note that the actual context switch must stay on the individual CPU regardless of it being reliable or relaxed as each register file is visible only from the CPU it belongs. Figure 37 shows a simplified version of the code segment of *__schedule()* that implements the above logic.

```
if (cpu == first_reliable_cpu())
        for_each_reliable_cpu(i)
                __schedule_cpu(i);  /* sets the next_task for each relaxed CPU*/
if (is_reliable_cpu(cpu))
        __schedule_cpu(cpu);
                /* sets the next task of this CPU */

/* common actions for both reliable and relaxed CPUs */
next = percpu(next_task, cpu)     /* get the next_task for this CPU */
```

Figure 37: The *__schedule()* Code Refactoring

At this point we assume that at least CPU 0 is always reliable and can perform all the extra work for every relaxed CPU on the system. In the future, we will generalize this assumption to distribute the extra work on all available reliable CPUs.

4.5. System Setup

The above mechanisms are independent from each other and are disabled by default. Each of them can be activated and studied separately in terms of performance and reliability trade-offs. Keeping them independent also facilitates debugging. The system call and page fault migration are implemented as per process features, whereas the scheduler migration is a system wide feature.

We control the per task features with a new system call *uniserver_ctl(int action*). The user can explicitly enable / disable the system call and page fault migration discussed earlier independently. In future versions, the default behavior may change. The actions that are currently available are shown in Figure 38.

```
A_SYS_GETFLAGS              - return the status of features for the task
A_SYS_AFFINITY_ENABLE       - enable system call migration
A_SYS_AFFINITY_DISABLE      - disable system call migration
A_SYS_MMFAULT_ENABLE        - enable page fault migration
A_SYS_MMFAULT_DISABLE       - disable page fault migration
```

Figure 38: The *uniserver_ctl()* Actions

The *uniserver_ctl()* system call sets the new features for the calling task and returns the old value.

The system wide features such as changing the set of reliable CPUs and enabling the error resilient scheduler are controlled through the */sys* interface of Linux. In particular, the control file is */sys/kernel/reliable_cpus*. Reading this file returns the current mask of reliable CPUs. Writing a CPU mask notifies the system about the new reliable and relaxed CPU domains. The mask follows a common syntax across the kernel which is the same with the *isolcpus* command line parameter discussed earlier. Some examples of valid actions on this control file in a system with eight available CPUs are shown in Figure 39.

```
#!/bin/bash
# Get the current mask of reliable CPUs
cat /sys/kernel/reliable_cpus
# 0x03 - CPUs 0 and 1 are reliable
# Make CPUs 0 and 5 reliable
echo "0,5" > /sys/kernel/reliable_cpus
# Make CPUs from 0 to 4 reliable
echo "0-4" > /sys/kernel/reliable_cpus
# Enable the error resilient scheduler
echo uniserver_sched > /sys/kernel/reliable_cpus
# Restore the default per CPU Linux scheduler
echo Linux_sched > /sys/kernel/reliable_cpus
```

Figure 39: Example of System Configuration

Each change, system wide or task specific, is audited in the kernel log to facilitate the monitoring and debugging of the system. When the system boots, all CPUs are marked as reliable assuming that the system starts in a fully reliable state. Also, the error resilient scheduler is disabled at startup.

By running a series of stress tests on relaxed CPUs under such a configuration with all or some of the features enabled, we observe performance comparable to the vanilla Linux in the majority of the combinations of

enabled features (17% overhead for computational intensive and 2% for I/O intensive applications with full-scheme system call, page fault, scheduler migration). The different overhead for different kind of workloads brings up the opportunity of educated protection policies that trade-off the degree of protection with the associated overhead.

Apart from performance, we see that under such a configuration, errors typically manifest at the user-level quite before the do at the system software level. System software is still affected, if we continue to lower the margins. Our mechanisms, thus, result to a configuration region where the system has indications that it is entering an unsafe operational state and the margins should be reconsidered.

5. Related Work

In this section we overview some of the existing work on the implementation of error resilient hypervisors related to the heterogeneous domains that we also adopted in our initial scheme as discussed above, while we also mention recent work on check pointing and restart which are among the candidates for enhancing the robustness if it is needed in the final software release.

5.1. Reliable Hypervisor

FTXen [13] is a fault tolerant implementation of the Xen hypervisor. Figure 40 shows the Xen hypervisor architecture. Xen supports two types of virtual domains, one privileged domain (*domain0)* which can access directly the Hardware and provides control interfaces to support and manage other domains. All other unprivileged domains are referred to as DomainU and every request that requires access to lower privilege levels is handled by the Xen hypervisor.

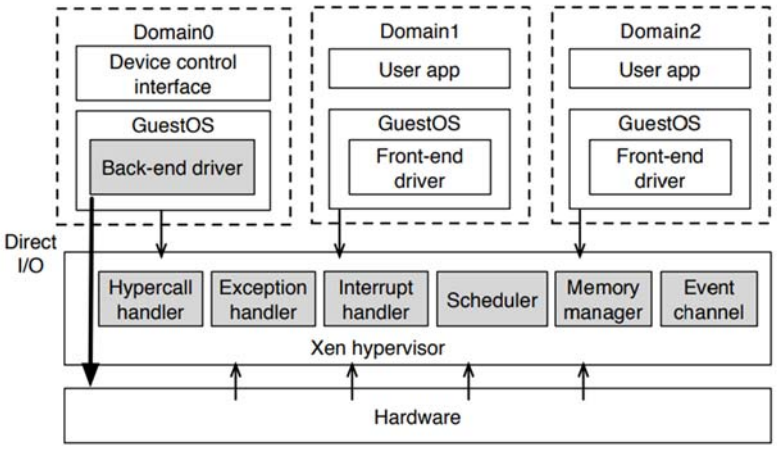

Figure 40: Xen Architecture

FTXen extends architecture of Xen and introduces an asymmetric logic by separating system CPUs to reliable and relaxed ones. Domain0 is pinned on the reliable CPUs, where DomainU to relaxed ones. In FTXen, updates to hypervisor critical data structures are restricted and only performed by reliable CPUs, thus when some relaxed CPUs have faults, the hypervisor is still intact and only the virtual machines assigned to these are affected.

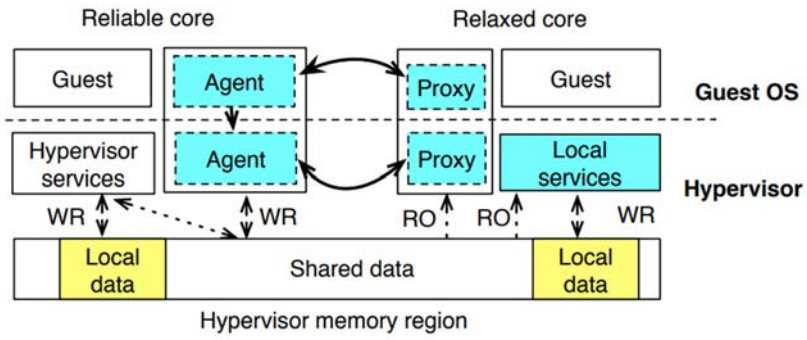

Figure 41: FTXen System Architecture

Figure 41 shows the inner communication of Domains running on relaxed cores with Domain0. By making the Domain0 responsible for every DomainU, the system can recover reliably from a potential fault of relaxed CPUs.

FTXen is based on the Xen hypervisor which is offering an asymmetric virtualization solution (separation of virtual domains to Domain0 and DomainU). On the contrary, our implementation is KVM-based inherits Linux Kernel and by nature is symmetric. This makes it more challenging to exploit the reliability-based heterogeneity of CPUs and protect them from hardware errors, as KVM treats all virtual machines equally to the host operating system in terms of reliability.

Shown in Figure 42, Hive [14] provides fault containment for shared-memory multiprocessors through the notion of cells. Each cell manages a subset of the system's memory and processors like a separate operating system, focusing on its own correctness.

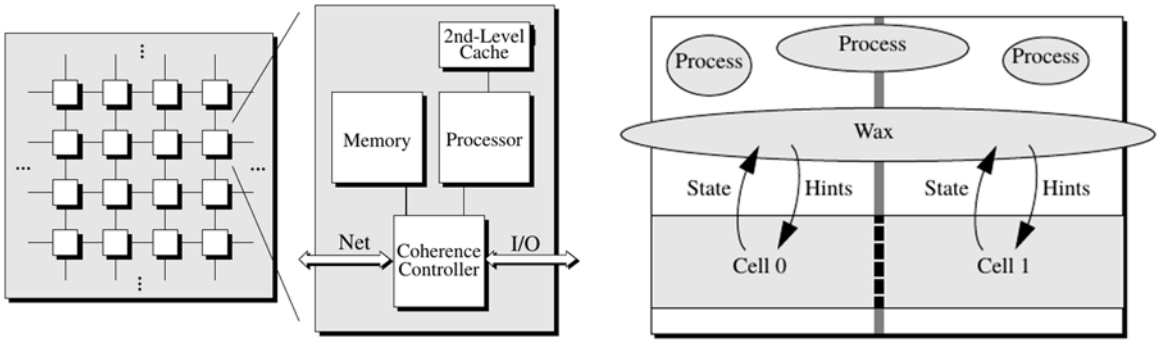

Figure 42: Hive System Architecture of Each Cell and Hive Cell Management

Under this system design, which is similar to a distributed system, each cell can isolate faults manifesting to itself from contaminating other cells. Hive uses a software strategy that discards the state of a faulty cell and page write protection for this purpose. Resource management, synchronization and load balancing between cells is controlled by a user space process which maintains global view of the system, notifies each cell and applies the appropriate policies.

Cellular Disco [15] offers resource isolation similar to Hive, but resource management and fault isolation is provided through a virtualization layer, leading to better resource management and scalability. The system consists of semi-independent cells with the properties of a virtual cluster. The protection of the virtual machine monitor layer assumes that the hardware is designed with recovery mechanisms. In case of failure the hardware notifies the good nodes to initiate their own software recovery strategy and determine which virtual machines are affected.

Focusing on recovering device drivers on Linux, the work on [16] describes a shadow driver mechanism based on Nooks [17] which provides transparent recovery under driver failures. A shadow driver layer introduces a shadow driver for each driver class which protects all same class drivers without the need of knowing implementation details of the actual drivers. Under normal execution a shadow driver passively monitors the requests to an actual driver. In case of failure, the shadow driver becomes active, intercepts the whole kernel-driver communication and replays the configuration and pending requests to a new fresh driver instance. After the completion of this recovery phase it switches back to passive mode.

5.2. Hardware Mirror Memory

Mirror memory [18] uses a doubled chip to back up the data immediately with extra hardware on the motherboard. When an error occurs, corrupt memory is quickly replaced by mirror memory. This technology is

widely used in high-end servers, such as HP's ProLiant server [19] memory, Dell's PowerEdge [4] series, APM Xgene [20] etc. Hardware mirror memory can also adapt to hot-plug memory technology [21]. Assisted by hot-plug memory, damaged memory can be replaced while the machine continues running. The new memory modules are synchronized nearly instantly with mirror memory, resulting in zero downtime. Mirror memory often generates too much overhead, however, and is consistently more expensive than native memory.

Memory mirroring really works only when supported by software at different levels, i.e. the data, application, or system level. These hardware solutions are thus limited to specific physical machines, and cannot be controlled at a sufficiently flexible or fine-grained mirroring level. Certain applications only duplicate a partial amount of critical data structures, like financial transactions, in memory. These data structures are usually duplicated with memory-mapped files, which incurs delays in backups as well as large overheads.

5.3. Software Mirror Memory

Memory-based reliability is also considered a part of the entire system's availability. Redundancy has been applied at different levels of granularity, such as the hardware, thread, and instruction levels. SWIFT [22], a software-only fault-detection technique, duplicates a program's instructions, inserting explicit validation codes to compare the results of original instructions and their corresponding duplicates. CRAFT [23] later improved SWIFT's approach by adding extra hardware structures. In order to direct the level of high reliability, PROFIT adjusts the level of protection and performance at fine granularities based on SWIFT. However, SWIFT incurs unwanted performance overhead as the number of instructions can be easily doubled, mainly due to the full duplication of instructions. Another typical reliable system model is dual-machine VM replication, in which a backup server is synchronized to the primary host. There is already a wealth of research on VM migration and VM replication.

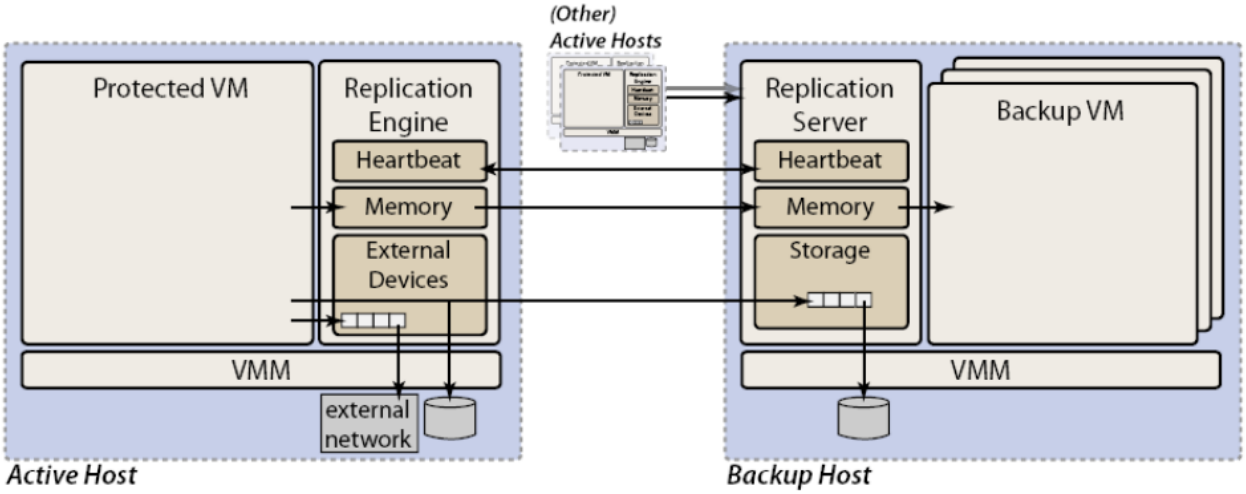

Figure 43: The Architecture of Remus

Remus: Figure 43 shows the architecture of Remus [24], in which the state of the primary VM is frequently recorded and transmitted to the backup server during execution. In a previous study's evaluation of Remus, Linux kernel compilation time was doubled and SPEC-Web [9] benchmarks suffered more slowdown when doing 40 checkpoints per second using a 1Gbit/s network connection for transmitting changes in the memory state. In order to improve the performance and scalability of Remus' checkpoint solution, ReNIC [24] provides an architectural extension to Single Root I/O Virtualization [25] (SR-IOV) for efficient I/O replications, but this requires new hardware-assisted I/O virtualization (such as SR-IOV). OCEAN enforces on-chip SRAM reliability

with a fault-tolerant buffer. It can optimally select the buffer size to minimize the energy overhead, with timing and area constraints, but it can only be applied to uniprocessor VMs and is highly architecture-specific.

kMemvisor: Shown in Figure 44, kMemvisor [18] creates redundant virtual space via virtualization technology, then it inserts instructions (mirror instructions) through binary translation technology and replicates data to this space. Data can be recovered from the replica when errors happen. In this section, we will introduce the architecture and the process of the data replication and recovery. kMemvisor can be divided into two parts: memory management module, and code translation module. Memory management module monitors the page table related operations to create mirror page tables. Page table maps virtual addresses to physical addresses. Code translation management module takes charge of inserting mirror instructions. It identifies all memory writing instructions and replicates them. The difference between original instruction and replicated instruction is the destination address. Replicated instructions will write the same data to “mirror virtual address”. Mirror virtual address is a virtual address mapped to an additional physical area. This physical area is created by kMemvisor to store the redundant data.

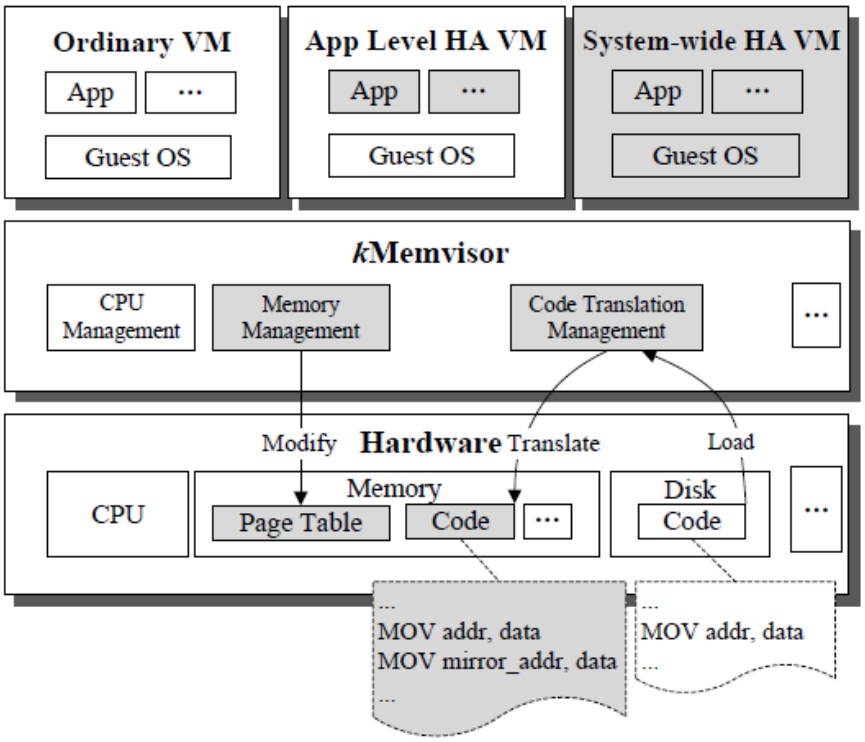

Figure 44: The Architecture of kMemvisor

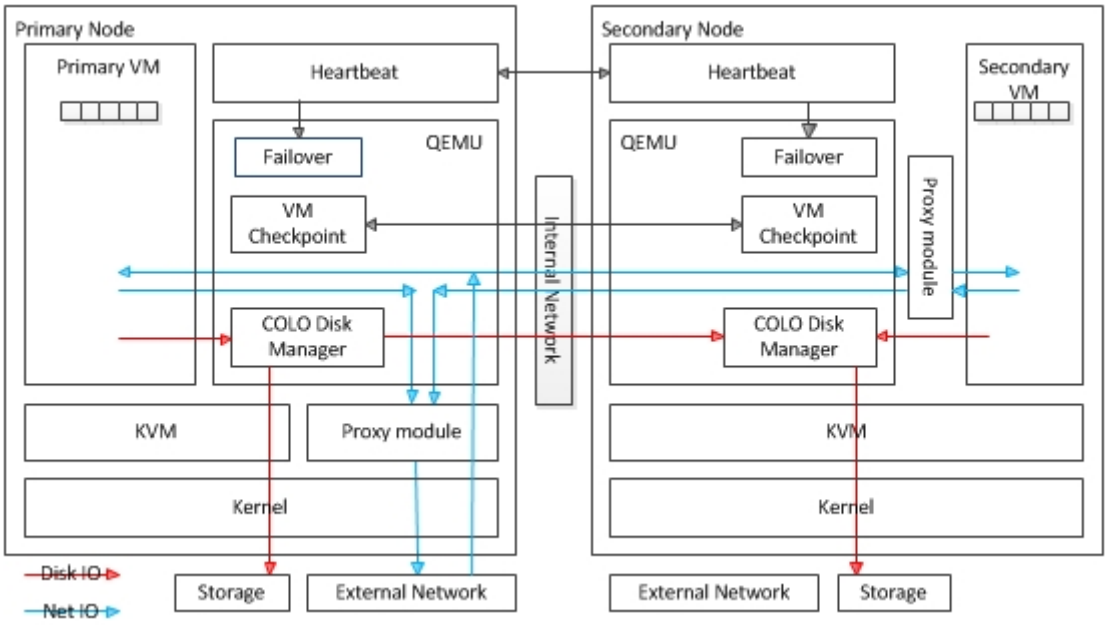

Figure 45: The Architecture of COLO

COLO: Shown in Figure 45, COLO [26] consists of a pair of networked physical nodes: The primary node running the Primary VM (PVM), and the secondary node running the Secondary VM (SVM) to maintain a valid replica of the PVM. PVM and SVM execute in parallel and generate output of response packets for client requests according to the application semantics. The incoming packets from the client or external network are received by the primary node, and then forwarded to the secondary node, so that both the PVM and the SVM are stimulated with the same requests.

COLO receives the outbound packets from both the PVM and SVM and compares them before allowing the output to be sent to clients. The SVM is qualified as a valid replica of the PVM, as long as it generates identical responses to all client requests. Once the differences in the outputs are detected between the PVM and SVM, COLO withholds transmission of the outbound packets until it has successfully synchronized the PVM state to the SVM.

5.4. Checkpoint Setup

Fixed *C*heckpoint: To setup checkpoint, one solution is to use a fixed time interval to periodically check the native system and then replicate the native data. The value of that interval usually comes from the trade-off of backup latency, the important time slot. For example, Remus optimizes checkpoint signaling in two ways: First, it reduces the number of inter-process requests required to suspend and resume the guest domain. Second, it entirely removes XenStore from the suspend/resume process. In the original code, when the migration process desired to suspend a VM it sent a message to Xend, the VM management daemon. Xend [27] in turn wrote a message to XenStore, which alerted the guest by an event channel (a virtual interrupt) that it should suspend execution. The guest's final act before suspending was to make a hypercall which descheduled the domain and caused Xen [16] to send a notification to XenStore, which then sent an interrupt to Xend, which finally returned control to the migration process. This convoluted process could take a nearly arbitrary amount of time — typical measured latency was in the range of 30 to 40 ms, but we saw delays as long as 500 ms in some cases.

*Event-driven C*heckpoint: Although a fixed time value is useful in checkpoint setup, such a value cannot be aware of the dynamic allocation. So, some systems need the event-driven checkpoint to avoid the redundant replication. Event including the parity check, double check to the data of different VMs.

For example, COLO [26] initiates a server-client system in the PVM and SVM at exactly the same state and then stimulate them with the same incoming events. Then, the identical results should be produced for a specific interval, which depends on the deterministic execution performance of PVM and SVM. As long as both VMs generate identical responses to client requests. If the output diverges due to the accumulated results of non-deterministic instructions, then the SVM is no longer a valid replica of the PVM. At that point, COLO will initiate a coarse-grained lock-step operation: It replicates the PVM state to the SVM.

6. Conclusion and Future Work

This section presents a number of conclusions that we have drawn from the previous section. Most importantly, these conclusions also indicate opportunities for future directions for our work.

6.1. Conclusion

A major new development pioneered by UniServer lies in significantly extending the capability of existing state-of-the art software packages on ARM based micro-servers. Hardware operating in safe conditions needs less support for fault-tolerance than the hardware operating outside its nominal operating conditions. This inherent behavior offers opportunities for dynamic adaptation of the overhead incurred due to the heterogeneity and fault-tolerance operation of the system. In this deliverable, we have proposed an initial implementation of an error-resilient hypervisor for UniServer.

In Section 2 of this document we reported on experiments with error-injection in both the kernelspace hypervisor (KVM) and the userspace hypervisor (QEMU). Through various benchmarks, we were able to perform characterization of sensitive data structures from both parts of hypervisor.

Section 3 of the document discussed our design for a basic prototype of the hypervisor for heterogeneous reliability memory. Specifically, we partitioned the current physical memory by adding a less domain and we revised key memory syscalls: *kmalloc, mmap, alloc_page*. Finally, we enabled QEMU to support our reliable domain. For example, if QEMU creates the syscall backend, hypervisor can trap and redirect guest physical memory to the reliable domain.

In Section 4, we migrated the sensitive syscall() to the reliable CPUs which could be guaranteed by adjusting voltage/ frequency etc. The main points of interest are the entry points from non-critical to critical code, namely the system calls, interrupts and page faults. Through various modifications to the current UniServer kernel (APM kernel), we successfully migrate the key code execution to the reliable CPUs from our definition.

6.2. Future Work 1: Enhancement of Hypervisor for Storage

In the next months, we plan to evaluate the framework (error-resilient hypervisor) developed above with more applications and further refine it if needed.

One other possible solution that we could adopt is mirror memory in virtual space. A mirror memory is a memory where the system can flexibly duplicate and from which it can recover data natively on an unreliable system. The time when data from native memory will be duplicated to the mirror memory will be determined by explicit checkpoints. The two techniques (mirror memory and checkpoint) are widely used in the hardware backup approaches [26] and could help enhance the robustness.

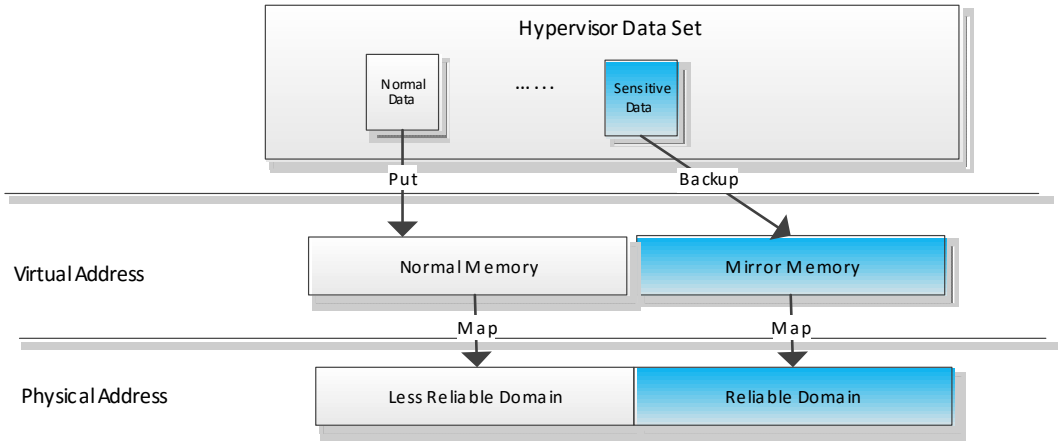

Figure 46: A Potential Implementation of Mirror Memory

Figure 46 shows a potential example of mirror memory. We hope that it creates the mirror space in virtual space for the same process. When an OS is initialized, a block of physical memory space is reserved as a mirror area in the less reliable domain. If occurring process write in the native space, mirror write instruction is then replicated in the mirror space, and redundant data is written by the mirror instruction.

Finally, considering the checkpoint, it requires massive testing data and analysis for predicting when an SDC error may occur. As we discussed in Section 2 we have started collecting such statistics for interfiling the time that tikes the system to crash after the detection of an error. For each checkpoint, the error-resilient hypervisor can backup sensitive data to mirror memory. Our motivation is to minimize record log and file in each checkpoint and reduce the frequency when triggering the checkpoint.

6.3. Future Work 2: Enhancement of Hypervisor for Execution

Realizing a robust and error resilient hypervisor is a non-trivial undertaking. It requires a lot of testing, careful design, good knowledge of the underlying hardware, deep understanding of the kernel primitives and subsystems as well as studying the bibliography for existing approaches. In the current version, we have identified some limitations, which we are working to alleviate.

Dynamic adjustment of the isolated CPUs according to the reliable CPUs needs to be introduced to the kernel. Apart from page faults, we currently ignore all other software exceptions that may occur because they are either fatal to the application (e.g. division by zero) or unlikely to happen in production (e.g. debug). Each exception is more or less a special case and a uniform solution to all may not be applicable. Besides the expected functionality of the migration approach of page faults, workqueues seem to be inefficient at least for the batch processes with full CPU utilization. As an alternative, we plan to reuse the same mechanism we have used for system calls based on the *sched_setaffinity()* function and migration threads.

The migration of page faults is currently a primary source of performance degradation. We are exploring ways to minimize this, while keeping the desired functionality. One approach we plan to investigate is to delay the migration of the task back to the relaxed CPUs when the rate of page faults is higher than a system defined threshold. This also opens an opportunity for policy managers to configure the reliability level of the system.

sched_setaffinity () is the only system call that may execute on relaxed CPUs and lead to system failure in case of error. By default, the migration of a task is performed by the migration thread of the sender CPU which

is a relaxed CPU when entering a system call. One way to further reduce the failure surface is to use migration threads from reliable CPUs only to handle the migration process in both entry and exit of system calls.

Considering new features, there is still code that can be migrated away from relaxed CPUs. A more fine-grained per-CPU clock interrupt manipulation may be feasible. Also, the time keeping of the system can be pinned to reliable CPUs if necessary. Another path worth exploring in order to improve performance is to increase the task priority for the duration of the migration. System calls are quite often in the critical path of the application. This is yet another control knob for policy managers to consider when taking decisions.

Finally, the actual CPU change from reliable to relaxed and vise-versa is not currently included in the kernel. A user-space tool used by the policy manager performs the necessary changes and then notifies the system with the new configuration. In future versions, this functionality will instead become part of the kernel. Moreover, the above API is indicative and subject to change depending on the user requirements and the inclusion of new features.

7. References

[1] H. Irfan, "Virtualization with KVM," *Linux J.,* p. 166, 2008.

[2] Intel Corporation, "Intel 64 and IA-32 architectures software developers manual. combined volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C," September 2014.

[3] P. Padala, ".Playing with ptrace, Part I,," *Linux Journal,* vol. 2002(103), no. 103, p. 5, 2002.

[4] DELL, "PowerEdge server," [Online]. Available: http://www.dell.com/en-us/work/shop/category/servers.

[5] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conference*, 2005.

[6] L. Group, "Virsh command reference," [Online]. Available: http://libvirt.org/virshcmdref.html.

[7] Parsec, "Parsec benchmark," [Online]. Available: http://parsec.cs.princeton.edu/.

[8] CloudSuite, "A benchmark suite for cloud service," [Online]. Available: http://cloudsuite.ch/.

[9] C. SPEC, "SPEC Benchmark," [Online]. Available: https://www.spec.org/benchmarks.html.

[10] P. K. A Baliga, "Lurking in the shadows: Identifying systemic threats to kernel data," in *SP'07. IEEE Symposium on. IEEE, 2007: 246-251.*, 2007.

[11] S. Hajnoczi, "QEMU internals: How guest physical RAM works," [Online]. Available: http://blog.vmsplice.net/2016/01/qemu-internals-how-guest-physical-ram.html.

[12] Nairobi-embedded, "Devices and RAM memory regions," [Online]. Available: http://nairobi-embedded.org/050_devices_and_ram_memory_regions.html.

[13] X. Jin, S. Park, T. Sheng, R. Chen, Z. Shan and Y. Zhou, "FTXen: Making hypervisor resilient to hardware faults on relaxed cores.," in *HPCA*, 2015.

[14] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu and A. Gupta, "Hive: Fault containment for shared-memory multiprocessors," *ACM SIGOPS Operating Systems Review,* vol. 29, no. 5, pp. 12-25, 1995.

[15] K. Govil, D. Teodosiu, Y. Huang and M. Rosenblum, "Cellular disco: Resource management using virtual clusters on shared-memory multiprocessors," in *ACM SIGOPS Operating Systems Review*, 1999.

[16] M. M. Swift, M. Annamalai, B. N. Bershad and H. M. Levy, "Recovering device drivers," *ACM Trans. Comput. Syst.,* vol. 24, no. 4, pp. 333-360, November 2006.

[17] M. M. Swift, B. N. Bershad and H. M. Levy, "Improving the reliability of commodity operating systems," in *ACM SIGOPS Operating Systems Review*, 2003.

[18] B. Wang, Z. Qi, H. Guan, H. Dong, W. Sun and Y. Dong, "kMemvisor: flexible system wide memory mirroring in virtual environments," in *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing (HPDC '13)*, New York, NY, USA, 2013.

[19] HP, "ProLiant server," HP, [Online]. Available: https://www.hpe.com/us/en/product-catalog/servers/proliant-servers.hits-12.html.

[20] APM, "X-Gene: World's first ARMv8 64-bit server on a chip solution," [Online]. Available: https://www.apm.com/products/data-center/x-gene-family/x-gene/.

[21] D. Hardy, M. Kleanthous, I. Sideris, A. Saidi, E. Ozer and Y. Sazeides, "An analytical framework for estimating tco and exploring data center design space," in *International Symposium on Performance Analysis of Systems and Software*.

[22] OpenStack, "OpenStack metering using ceilometer," [Online]. Available: https://www.mirantis.com/blog/openstack-metering-using-ceilometer/.

[23] OpenStack, "Open source software for creating private and public clouds," [Online]. Available: https://www.openstack.org/.

[24] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, San Francisco, 2008.

[25] Y. Dong, X. Yang and J. Li, "High performance network virtualization with SR-IOV," *Journal of Parallel and Distributed Computing,* vol. 72, no. 11, pp. 1471-1480, 2012.

[26] Y. Dong, W. Ye, Y. Jiang, I. Pratt, S. Ma, J. Li and H. Guan, "COLO: Coarse-grained lock-stepping virtual machines for non-stop service," in *Proceedings of the 4th annual Symposium on Cloud Computing (SOCC '13)*, New York, NY, USA, 2013.

[27] Xend, "Xend bussiness solution," [Online]. Available: http://new.xend.com.ph/.

[END OF DOCUMENT]

