KairosVM: Deterministic Introspection for Real-time Virtual Machine Hierarchical Scheduling

Kevin Burns, Antonio Barbalace, Vincent Legout, Binoy Ravindran
Department of Electrical and Computer Engineering
Viginia Tech, Blacksburg, VA
{kevinpb, antoniob, vlegout, binoy}@vt.edu

***Abstract*—Consolidation and isolation are key technologies that drove the undisputed popularity of virtualization in most of the computer industry. This popularity has recently led to a growing interest in real-time virtualization, making this technology enter the real-time system industry. However, it has several issues due to the strict timing guarantees contracted. Moreover supporting legacy software stacks adds another level of complexity when the software is a black box. We present KairosVM, a latency-bounded, real-time extension to Linux's KVM module. It aims to bridge the lack of communication of the real-time requirements between the guest scheduler and the host scheduler, exploiting virtual machine introspection. The hypervisor captures the real-time requirements of the guest by catching previously added undefined instructions. Our evaluations show that KairosVM's overhead is negligible when compared to existing introspection solutions thus can be used in real-time.**

I. Introduction

Lots of existing real-time systems have been running for years on the same platform since deployment. However, this hardware is obsolescent and updating the software to take advantage of newer processors is often not an option. Virtualization is thus an appealing solution for these legacy systems. It is therefore gaining traction in a variety of fields, including real-time control. With such systems, the overheads of a virtual platform must be carefully analyzed before promising certain real-time guarantees, as these guarantees have the potential of not being met due to hidden latencies in the design.

When using virtualization to consolidate existing real-time software stacks, there are many issues that need to be addressed. First of all, maintaining the same level of real-time guarantees (soft, hard, or best-effort) contracted in the virtualized software is mandatory. Furthermore, traditional real-time scheduling techniques do not apply in virtual environments where multiple schedulers coexist, in the hypervisor and in the virtual machines.

Several solutions have already been proposed to schedule virtualized systems using the hierarchical real-time scheduling theory, e.g. RT-Xen [1]. It accounts for one scheduler in the hypervisor and one in each guest operating system. However, they require either modifications of the guest operating system or an accurate offline analysis. There are many cases when we want to reuse the previous existing software stack without modifications. For example, the entire code-base could not be available. Also, organizations do not want to pay an exorbitant amount of money to rewrite and revalidate what they already wrote many years ago.

In order to consolidate unmodified real-time software stack we propose techniques to introspect the software to monitor real-time scheduler calls. Our solution targets modern operating systems with a traditional user-space/kernel-space division, where real-time applications are implemented as user-space threads. This work can easily be extended to one address-space real-time operating systems, like VxWorks. Our solution will track the scheduling calls on the guest systems, while interacting with a real-time scheduler in the hypervisor to provide the same guarantees contracted on each of the virtualized software stacks.

We implement this mechanism on top of the existing ChronOS 3.4 operating system [2] based on the 3.4.82-rt100 Linux kernel. The modified ChronOS would act as the hypervisor. We are proposing the adoption of a plugin infrastructure, where each type of virtualized software stack (e.g. RTAI [3], LITMUS^{RT} [4], *PREEMPT_RT* [5], etc.) would have its own plugin. Thus we reduce the effort of the system integrator by providing an interface to either write a new plugin or extend an existing one. Next we will show how we implemented our proposed mechanism for a particular combination of software stacks. Specifically, we used ChronOS Linux as a guest. We show how this solution is suitable for a real time system and has lower overhead than other filtering solutions like Nitro [6].

Our contribution includes a virtual machine introspection (VMI) mechanism and a pluggable interface to support different RTOS. The interface can be used to feed the right information to the host OS scheduler in order to make the best scheduling decision. Our lightweight introspection mechanism allows the host scheduler to reconstruct the real-time state of the guest. To the best of our knowledge introspection techniques are being used in many other projects, specifically related to security, but we are the first to propose its use in real-time systems. It is important to emphasize that this paper addresses an architecture for real-time schedulers and not actual real-time scheduling algorithms, this is left as future work.

Section II introduces previous works related to real-time virtualization and virtual machine introspection. Section III discusses virtualization solutions, our task model and ChronOS. Sections IV and V detail our contributions: our event trapping approach and our implementation for x86 systems. How this approach can be integrated in KairosVM is discussed in Section VI, while Sections VII and VIII evaluate our solution and conclude.

II. Related Work

While introspection techniques have not before been used to inform the host scheduler, several informed scheduling platforms have already been proposed. Many propriety real-time hypervisor solutions are available on the market. Wind River, National Instruments, TenAsys, and RTS all sell real-time virtual machine management solutions. However, these proprietary solutions tend to use strict partitioning schemes. Dynamic real-time scheduling is left to research based tools and solutions.

The scheduling theory to schedule real-time virtual machines is based on hierarchical scheduling (e.g. [7], [8], [9]). This theory has first been developed such that multiple applications with several real-time tasks can be executed in the same platform. But it can be used in the context of virtualization where an application is a virtual machine. In hierarchical scheduling, the host operating system is responsible for scheduling the virtual machines. And each virtual machine has a local scheduler to schedule its own tasks. Several solutions has been developed based on this theory.

Xen offers a paravirtualization approach and it thus a popular target hypervisor when researchers focus on the scheduling translation problem (e.g. [10], [11]). The most advanced solution based on Xen is RT-Xen [1], [12]. They first implemented in [1] four existing hierarchical servers: Defferable, Periodic, Polling, and Sporadic and show that they improve the real-time guarantees of the guests compared to the existing Xen schedulers. Then, in [12], RT-Xen was modified to support compositional scheduling frameworks (CSF) [13].

The L4 microkernel has also been used has a hypervisor. Microkernels and traditional hypervisors share many characteristics, so this is a natural use case. L4 also has a variation called L4 Fiasco which is hard real-time capable. Fiasco has also been used as a real-time hypervisor to observe the need of a "flattened hierarchical scheduler" [14]. The Fiasco microkernel offers an interface that allows user level schedulers to change parameters in the kernel level scheduler policy. The authors use paravirtualization techniques (vmcalls) to "flatten" the guest and host schedulers. This is done by allowing the guest real-time operating system to choose the underlying priority used on the host (Fiasco) via the interface it provides. That is, the guest has task by task control over the VCPU process' priority. The results of this technique indicates an overhead incurred by the vmcalls that are used to do the priority switch.

Finally, several solutions has been developed using Linux as the host. Using KVM, Kiszka [15] patched Linux with *PREEMPT_RT* to support real-time guarantees and then used a fix-priority approach. This work showed that Linux has promise as a real-time hypervisor. In that, using a real-time fixed-priority scheduling policy (SCHED_FIFO) on both the guest and host can lead to reasonably low overhead and reduced scheduling jitter for some use cases. However, there are still problems to be solved, namely if there are still running tasks on the native Linux install there may be priority inversions.

Nitro [6] is a virtual machine introspection framework designed to monitor every system call made in the guests and detect malicious activities. Thus Nitro is not suitable for real-time systems because of the latency associated with catching every system call made in the guests.

III. Background

Common practice in building real-time systems dictates that the software stack must be deployed directly on the hardware in order to provide the expected time guarantees. The real-time operating system and applications have full control of all hardware resources in the system. Thus the OS maintains a consistent view of the time. Applications are usually tuned for a specific platform in order to meet time constraints imposed by the activity they are carrying on. Furthermore the full software stack undergoes extensive testing on the specific hardware in order to prove its ability to comply to some real-time properties.

A. Virtualization

When moving such a software stack to a virtualization environment, it will not directly control the hardware anymore, losing its capability to provide time guarantees. Hardware resources are usually emulated, especially timing devices, critical for a real-time software stack. Moreover, on the same virtualization platform, many software stacks execute concurrently on different virtual machines. Therefore hardware devices are eventually shared among virtual machines. This sharing happens at the hypervisor level that proxies or time-partition the access to the shared resource. Because of that, the flow of time perceived by the software running in a virtual machine is different from the one perceived by the same software running on real hardware.

In a virtualization environment the hypervisor (or virtual machine monitor) schedules the virtual machines for execution on the available processors. When making scheduling decisions, the hypervisor, Linux in KVM/QEMU, treats the software stack in the VMs as a black box. For the scheduler, a VM hosting a real-time software does not differ from a VM hosting a general purpose operating system. Therefore application's declared real-time properties are not propagated down the software stack. Consolidating real-time software stacks on genuine virtualization solutions as KVM/QEMU will not let the user retain the same real-time properties that he/she was observing on bare hardware.

In order to consolidate real-time systems into a virtualization environment the propagation of real-time properties down the software stack will enable to make more informed scheduling decisions at each software layer. When the software stack can be modified, e.g. when the source code is available, information passing from the virtual machine to the hypervisor can happen via hypercalls or function calls (i.e. paravirtualization). When the software stack is not modifiable there is no easy way to communicate between VM and hypervisor. Virtual machine introspection can be leverage as a way to live gathering behavioral information of the VM. This creates an unidirectional communication channel in which a stream of events flows from the VM to the hypervisor but the inverse communication path is not enabled.

Offline scheduling analysis can be used to avoid the need of communication between VM and hypervisor as demonstrated by previous work [1]. Such work is limited to periodic tasks. Therefore the focus of this paper is on virtual machine introspection as an aid for real-time hierarchical scheduling. The approach proposed do not requires offline analysis of the

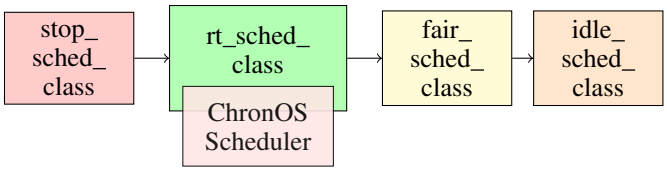

Fig. 1. Linux's scheduling classes ordered by their priority. The ChronOS scheduler is nested in the RT scheduling class

virtualized software stack and is not limited to periodic task sets. In the following we present a reference task model and an overview of ChronOS Linux, our real-time operating system.

B. Task Model

In this paper we do not focus on a specific task model. The natural model of ChronOS Linux is the aperiodic model, due to the best-effort scheduling policies that come with it. Despite this, we will often make references to the periodic task model. In this model a set of periodic tasks $T = \{t_1, t_2, .., t_n\}$ are characterized by release time r, period p, worst case execution time e, and deadline d. Thus a task t_x is defined by the tuple (r_x, p_x, e_x, d_x). We assume that all tasks in T are initially released at the same time $r_i = 0, i = 1..n$ and all deadlines are equal to periods $d_i = p_i, i = 1..n$. The term job refers to each periodic release of a task.

C. ChronOS

ChronOS Linux [2] is a scheduling framework developed inside the Linux scheduler. It implements traditional real-time scheduling algorithms as well as best-effort policies while fully supporting multicore platforms.

As illustrated in Figure 1 the Linux scheduler is implemented as an ordered list of scheduling classes. Every time a scheduling decision must be made, the Linux scheduler scans the class' list and stops at the first class that has ready tasks to be executed. The list is navigated from the highest priority class (`stop_sched_class`) to the lower priority class (`idle_sched_class`). These two classes are not meant to be used from user space applications but are reserved to specific kernel threads. The other two classes are designed to schedule all kind of threads. The `fair_sched_class` implements a time-sharing scheduling policy (historically managed with the `nice` command in UNIX). The `rt_sched_class` implements the POSIX `SCHED_FIFO` and `SCHED_RR` by mean of a multi-level priority queue indexed with a bitmap. There are 100 levels or priorities. In kernel space, 0 is the highest real-time priority while 99 is the least (which is the opposite in user space). In a multiprocessor platform Linux maintains for each processor a ready queue per scheduling class.

ChronOS scheduler is not implemented as a Linux scheduling class but as an extension of `rt_sched_class`. A ChronOS scheduling queue lives at a fixed priority level n in the aforementioned class, see Figure 2. For the sake of completeness it is important to note that with this architecture real-time tasks can be divided as *system critical*, *hard real-time* and *soft real-time*. Despite such division hold, ChronOS real-time guarantees are dependent upon the real-time guarantees

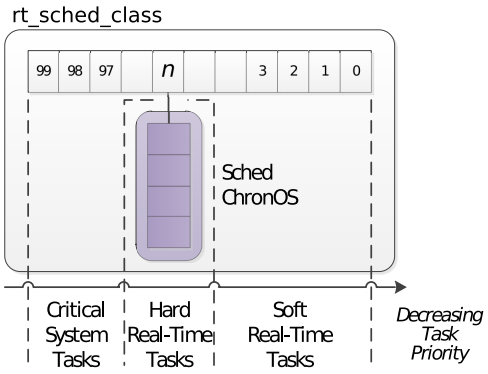

Fig. 2. Real-time scheduling classes in ChronOS Linux. ChronOS' scheduled tasks are meant to be at `SCHED_FIFO` priority n

```
long begin_rtseg(int tid, int prio, int
    max_util, struct timespec* deadline,
    struct timespec* period, unsigned long
     exec_time);
long end_rtseg(int tid, int prio);
long add_abort_handler(int tid, int
    max_util, struct timespec *deadline,
    unsigned long exec_time);
long set_scheduler(int scheduler, int prio
    , unsigned long cpus);
```

Fig. 3. Chronos application programming interface (without mutexes)

provided by the `rt_sched_class` in which it is embedded, and the *PREEMPT_RT* patches. In addition ChronOS has a pluggable interface: each plugin is a different scheduling policy.

In ChronOS a real-time task is a user-space thread scheduled at priority n within the `SCHED_FIFO` policy. ChronOS semantic requires to mark each job with the library calls `begin_rtseg()` and `end_rtseg()`; to mark its start and end respectively. In ChronOS terminology a job is more generically referred segment. The concise application programming interface without mutexes is depicted in Figure 3.

The function `add_abort_handler()` allows the user to be notified if a job exceed is deadline; `set_scheduler()` allows to select which ChronOS' scheduling policy to use. From a system software point of view all of the APIs are system calls but the first three map to the syscall number `__NR_do_rt_seg` while the latter to `__NR_set_scheduler`. Why this matter will be more clear in the next Section.

Figure 4 pictures a periodic task in ChronOS in pseudo C code. The real-time segment of each job must be written between the invocations of `begin_rtseg` and `end_rtseg`. In order to make the task periodic the programmer should insert a sleep call (`usleep()` here). When the task exits the real-time segment its `SCHED_FIFO` priority will be increased in order to be awakened at the exact release time. The library function `usleep()` calls the syscall `sys_nanosleep` (that maps to the number `__NR_nanosleep`). All the other functions in the code above are user-space only (despite `pthread_setschedparam`); they do not provide any ad-

```
void periodic_task()
{
  param.sched_priority=TASK_START_PRIO;
  pthread_setschedparam(0, SCHED_FIFO,
                        &param);
  for (job=0; job<MAX_JOBS; job++) {
    dead = find_job_deadline();
    begin_rtseg(TASK_RUN_PRIO, dead);
    // real-time segment
    end_rtseg(TASK_CLEANUP_PRIO);
    next = find_next_job_release();
    usleep(now - next);
  }
}
```

Fig. 4. Example of periodic task in Chronos

ditional information for the real-time scheduling.

IV. Event Trapping

In a real-time operating system with per process separate address spaces (e.g. Linux with real-time extensions), a real-time task will be eventually created in user space. To make the kernel-space scheduler be aware of such a task, and its real-time properties, system calls are exploited.

When this software stack is moved into a virtual machine, a naive approach to get the hypervisor informed of real-time scheduling events, is to configure the hypervisor to trap system calls. Because our approach assumes unmodified virtualized software stacks, we based our work on hardware enabled full-virtualization in order to support legacy systems. Already existent virtual machine introspection (VMI) libraries offer syscall tracing functionality for hardware assisted fully virtualized environments. Syscall events are caught by the processor virtualization extensions, Intel's VMX or AMD's SVM on x86 architectures. Every time the software in the virtual machine executes a *syscall* or *sysenter* instruction the hardware virtualization extensions (VTx) interrupts the guest and context switches to the hypervisor. At this point, the aforementioned VMI libraries add code in order to check the syscall number against a list of registered syscalls. In the case of an hit in the list some notifications or logging functions are called, otherwise the control is returned to the virtual machine.

There are two main drawbacks in adopting a syscall tracing approach in a real-time virtualization setup. *First* the number of system calls that occur in a virtual machine can be difficultly bounded. Thus the number of syscalls trapped by the hypervisor can be significant and the trapping will add a non-negligible overhead to the virtualized system (refer to Section VII). Moreover because hard, soft and non real-time tasks can potentially generate system calls, not all syscalls are interesting for our purpose of trapping scheduling related events. *Second* not all real-time operating systems have a per process separate address space therefore system call tracing is not the right solution to our problem. Instead function calls must be trapped.

In addition to the two observations above, most of the VMI libraries comes as user space software tools. Thus they are not directly usable in kernel space where the scheduler resides.

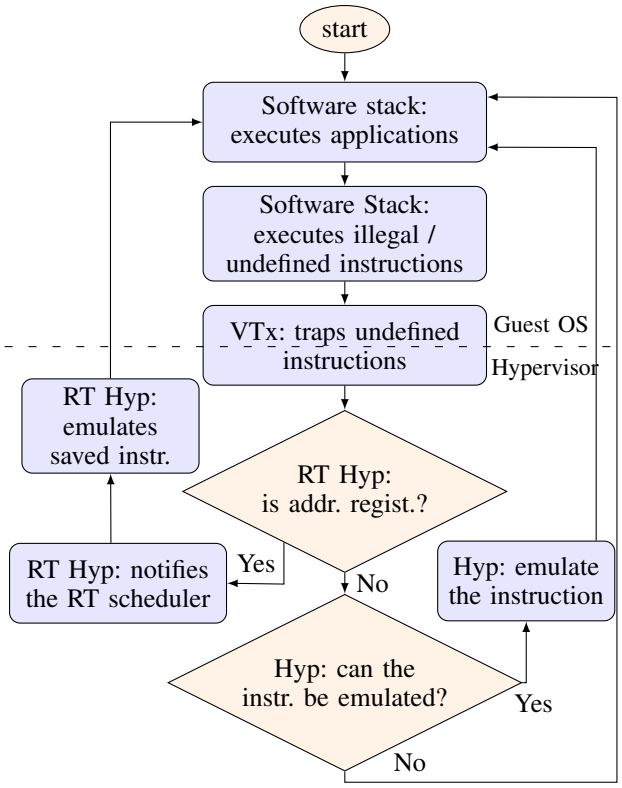

Fig. 5. KairosVM's events trapping mechanism. KairosVM injects an illegal/undefined instruction at the addresses where each real-time scheduling function is located. If the undefined instruction was not injected by KairosVM the hypervisor will continue its original execution path

A. The KairosVM Approach

In order to create a more deterministic and less intrusive introspection framework we decided to do not trap each syscall. Instead, we propose to live inject illegal/undefined instructions in the guest. An illegal/undefined instruction will be inserted at any point of interest, i.e. each scheduling related function. When the processor will fetch such instructions, it will trigger a hyperswitch and pass the control to the hypervisor that in turn will invoke KairosVM. This process is illustrated in Figure 5 that includes the execution flow for illegal and undefined instructions already existent in the guest software stack.

KairosVM, "RT Hyp" in Figure 5, is suddenly invoked by the hypervisor when an illegal/undefined instruction has been trapped. If the address at which the fault happened does not correspond to any of the addresses at which we were interested in, the hypervisor will handle it. The hypervisor attempts to emulate the instruction at the faulty address and then returns the control to the virtual machine. Whereas when the address is one at which we inserted an illegal/undefined instruction, KairosVM will first notify the real-time scheduler, and then emulate the instructions that we had overwritten with the illegal/undefined one. At that point KairosVM passes the control to the guest software.

Note that the proposed mechanism removes both drawbacks innate in available VMI libraries. Our approach traps only the real-time scheduling events we are interested in, without adding any other overhead. Because these events will

be due to the real-time activity they can be accounted in a real-time hierarchical scheduler. We will show in Section VII that our trapping adds a bounded overhead to the execution. Furthermore, KairosVM approach is more generic, it can actually be exploited to trap syscalls, interrupts and obviously function calls, because illegal instructions can be inserted mostly everywhere.

In order to trace the activity of a real-time operating system running in a virtual machine, the scheduler in the hypervisor should know which is the scheduling system call interface in use. This information is much more helpful if related with the notion of which entity, task, thread or process, triggered it.

In KairosVM we provide mechanisms to identify the entity that triggered an event and auxiliary methods to manipulate function arguments. We believe that these functionalities, added to the hardware context identification facilities provided by KVM, will facilitate and foster the development of hierarchical schedulers in Linux/KVM.

V. x86 Implementation

We implemented an initial prototype of the KairosVM introspection engine on Linux kernel version 3.4.82-chronos *PREEMPT_RT* and KVM/QEMU version 0.12.3 with Intel x86 64bit VMX as a target platform. The current implementation includes an initialization stage and a runtime, hypervisor-side, trapping and handling stage.

A. Initialization

In order to being able to trap events, illegal or undefined instructions must be injected in the guest software stack. The injection should happen before the guest's real-time application starts but after the guest OS has been completely initialized. Furthermore, injection requires to find the entry point of each function that we would like to monitor. Although the scheduling interface of a real-time operating system is known in advance it is not always trivial to find its functions entry points. Different techniques can be used based on how far we can introspect. Disassembling, symbol listing and exploiting debug symbols can be all used offline on the guest binaries thus before the system starts executing a real-time application. Within the Linux kernel, various authors [16], [17] suggest to exploit *System.map*, which contains the kernel symbol list, and usually accompanies each installed Linux kernel. The same list can be obtained by reading `/proc/kallsyms`.

The current prototype targets Linux in the VM: a setup in which the guest OS has a per process address space. Thus we decided to trap syscalls, and we assume the syscall number of the functions we want to monitor are known. To identify the entry point of one of such functions we look up the address of the `sys_call_table` symbol from *System.map*, and then we get the nth entry starting from that address, where n is the syscall id that we would like to monitor. If *System.map* is not available the lookup is slightly more complicated: it requires to read the syscall entry point from (virtualized) hardware registers (IDT table, or MSR registers, respectively for `syscall` and `sysenter` instructions), and disassemble from the syscall entry point until the first `call` instruction, which embeds the `sys_call_table` address (note that this is highly Linux version specific).

With the knowledge of the function entry point we are saving the first m bytes at that address while overwriting the first few bytes with an undefined instruction. We set m equal to 15 as KVM/QEMU is doing when emulating. We are maintaining the syscall number, the address of the function, and the m bytes of binary code in a single data structure. Structures are stored in an hash table indexed by the hashed function address. In order to maintain a minimal and constant lookup time we store a single structure per table entry. In case of collisions KairosVM is rehashing the table, this is acceptable because during initialization there are no real-time constraints to be meet.

B. Event Trapping

To be compliant to our design, Figure 5, we choose to use *undefined instruction 0*, *ud0*, as the injected instruction that will cause the trap to the hypervisor. We decided to use *ud0* instead of the well known *ud2*, commonly used in the Linux source code to signal bugs (`BUG()` macro, i.e. assert in kernel space), in order to reduce false positives. The *undefined instruction 0* is an Intel undocumented instruction, but documented by AMD [18], that corresponds to the two bytes {`0x0F, 0xFF`}.

Whenever the software in the virtual machine hits one of the injected *ud0* instructions, as a consequence of a scheduling syscall, it context switches to the hypervisor. Because KVM is by default configuring the hardware virtualizaiton extensions to trap on undefined instructions, in order to emulate instructions that are not supported on the platform, we didn't have to enable undefined instructions trapping.

After the hypervisor gets called for an undefined instruction it passes the control to KairosVM which checks if the faulty address is present in the hash table. If it is not present it returns the control to the hypervisor. Otherwise, our prototype first dereferences the system call parameters and notify the ChronOS scheduler (Section VI), then emulates the instructions overwritten by the inserted *ud0* (by using KVM's `x86_emulate_instruction()`), and returns the control to the virtual machine.

Note that *ud0* is two bytes long and in x86 assembly there are (few) instructions which lenght is one single byte. Therefore the number of instructions that must be emulated can be maximum two. In order to account for determinism, we measured the emulation time for all of the one byte instrcution and we find it mostly constant. Despite that we observed that the first instruction of each function we trapped is a `push` type of instruction. This proves the suitability of our system in a real-time environment.

VI. KairosVM Integration

The events that KairosVM will trap are dependent upon the real-time operating system installed in a virtual machine. Because we would like to create the foundations to support any virtualizable real-time operating system we adopted a plugin interface. We aim to develop a plugin for each supported OS, we currently support ChronOS only.

A. ChronOS Introspection

In ChronOS, to trap the real-time scheduling events we monitor the following syscalls: `__NR_nanosleep`, `__NR_do_rt_seg` and `__NR_set_scheduler`. (We already described ChronOS internals and how an application should be written in Section III.) Removing the latter the other two syscalls are heavily used in real-time: they are called for each (periodic) job. Specifically the syscall `__NR_do_rt_seg` is called at the release and completion of each job.

The information carried by these events can be used by the scheduling algorithm in the hypervisor to better allocate CPU time to the guests. However the event per se is not enough, its arguments, that are carrying the real-time characterization of the segment, must be extracted from the syscall context. Therefore, in the ChronOS plugin, for `begin_rtseg` as an example, we extract from the virtual CPU registers its arguments (tid, priority, utility, deadline, period and wcet) that are located at the address stored in the *rsi* register. Such information, augmented with the entity triggering the event, i.e. the `task_struct` pointer of the current thread in Linux, is passed to the hypervisor scheduler (ChronOS in our setup). Because these features are OS ABI dependent we implemented them in the plugin.

As stated in Section I this work does not contribute any scheduling algorithm. Nevertheless with the information gathered by introspecting the guest OSes, new scheduling algorithms can be implemented in the hypervisor in order to meet real-time properties contracted in the guest software stack. These new algorithms can be based on hierarchical scheduling theory as introduced in Section II. Contrary to existing solutions, like RT-Xen [1], KairosVM enables a scheduling algorithm to get informed when tasks finish their execution earlier than expected or when new tasks arrive, fostering dynamic decisions.

B. Plugin Architecture

KairosVM integrates with KVM and ChronOS, and offers a pluggable interface in order to support the introspection of various real-time operating system, which differs by the way scheduler related events and properties are communicated by applications to a kernel.

A plugin is a standard Linux module which can be loaded and unloaded at runtime. In order to register a KairosVM plugin the developer has to call `rt_plugin_register()` in the module intialization routine. To deregister a plugin `rt_plugin_unregister()` must be called. Within the registering function a structure `rt_plugin` must be passed. The `rt_plugin` has to be populated with the name and a range of versions of the RTOS supported by the plugin, probe, initialization and exit functions. The probe function is meant to check if the OS in the VM is exactly the one which the plugin was made for. The initialization routine will account for introspect the software, insert the undefined instructions, and register the notification routines (see Section V-A). The exit function cleans up the guest software from the inserted faulty instructions. The notification functions are meant to be called during the real-time execution (see Section V-B), they should decode the syscall arguments, convert the information for the hypervisor-level scheduler and notify it.

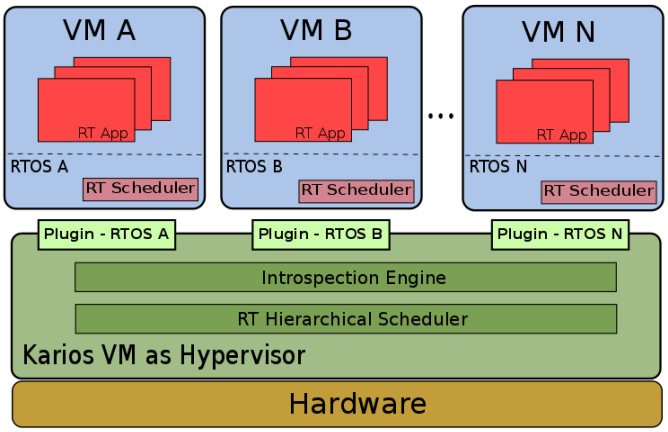

Fig. 6. KairosVM overall architecture

It is worth stressing that writing a plugin requires to know the internals of the RTOS running in the virtual machine. Despite that, KairosVM helps the plugin developer by providing her/him with routines for registering and deregistering events at a specific address, complemented by the fact that when an event occurs KairosVM will check it for genuinity and eventually call the registered notification function. When this function will return KairosVM will take care of the emulation and the rest of handling. Figure 6 illustrates the overall KairosVM architecture including the plugins. This figure includes three different guest operating systems (RTOS A, B and C): each of them requires a different plugin.

VII. Evaluation

To evaluate our introspection mechanism, we performed several experiments. We run the tests on an Intel Xeon E5520 processor with eight cores at 2.27GHz, and 16GB of RAM. The host is running Ubuntu Server 10.04 with Linux kernel 3.4.82, ChronOS 3.4 patched, as the hypervisor. We used KVM/QEMU version 0.12.3. On the guest, we used Ubuntu Server 10.04 with Linux kernel 3.0.24, patched with our ChronOS 3.0 patches. We compared vanilla KVM, KairosVM and Nitro on a set of benchmarks. Nitro [6] represents the state-of-the-art in virtual machine syscall tracing tools, we used Nitro for Linux kernel 3.13.0-rc8 that we backported to 3.4.82. The KVM module was swapped between tests to include and exclude Nitro or KairosVM.

Tests were run on a VM with a single VCPU that is pinned to a single CPU. We used the `isolcpus` kernel command line argument to dedicate specific cores to the KVM processes. We also followed the advice from Kiszka [15] to raise the priorities of QEMU's threads to improve the guest responsiveness.

In the experiments, we used two notions of time in the virtual machine: *virtual time*, measured with `clock_gettime()`, and *physical time*, obtained by reading the machine time stamp counter register (TSC). We tweaked KVM in order to do not dynamically change the *TSC Offset* in the *VMCS* in order to obtain reliable physical time measurements. Furthermore, because the VCPU is pinned to a

single CPU, we are mitigating any possible inconsistencies due to subtracting TSC reads from different CPUs. Compared to virtual time, the physical time has an absolute connotation, while the former is relative, because the TSC does not pause when the virtual machine is idle.

We first compared the overhead due to syscall trapping by measuring single system call latency. We then measured the effect of these overheads on the performance of real-time applications with different non real-time workload.

A. Trapping Overhead

We compared the overhead due to syscall tracing in KairosVM with Nitro, and we relate these measurement against an execution on KVM without any tracing mechanism active and against an execution without any virtualization. The test examines the latency of 6 function libraries (with a corresponding syscall) that are commonly used in ChronOS real-time applications, but only `begin_rtseg` and `end_rtseg` have been trapped/traced.

Because Nitro is logging the tracing data in user space, on the Linux host, in order to make the comparison fair, we accurately remove such user space time from Nitro's results. We collected the physical time over 1000 experiments, we computed average, min and max, Figure 7 shows the results.

The results clearly indicate that Nitro has the higher overhead per syscall compared to other solutions. Bare KVM virtualization introduces between 1% and 30% of additional overhead compared to a non virtualized setup. KairosVM does not add any quantifiable overhead to the latencies of the syscalls that are not meant to be trapped compared to KVM. Differently, Nitro does that, because it is trapping every syscall. For the syscalls that have been trapped, KairosVM is between 7 to 10 times slower than KVM. Note that, the time required to trap a syscall in KVM is mostly equivalent to the syscall latency in Nitro for non traced syscalls. Therefore we conclude that this time is highly due to the hyperswitch. Nitro is substantially slower than KairosVM, being between 139 to 248 times slower then KVM. This is due to the asynchronous notification mechanism implemented in Nitro.

B. Real-Time Performance

To determine the effect of each introspection mechanism on real-time performance we used the *sched_test_app* benchmark which is part of ChronOS distribution. This is a real-time benchmark that simulates real-time workloads using the Baker model [19].

We ran this benchmark in two configurations. In the first one, a CPU intensive application is running alongside the benchmark. In the second configuration Bonnie++ [20] is used instead of the CPU intensive application. Bonnie++ is a benchmark suite that is aimed at performing a number of simple tests of hard drive and file system performance. Thus the second configuration generates much more system calls than the first one. In both configurations we used a script to automatically start the non real-time application, and load *sched_test_app*; the script then waits for *sched_test_app* to finish, collects the output and kill the non real-time application. Every data point in our experiments is averaged over 20

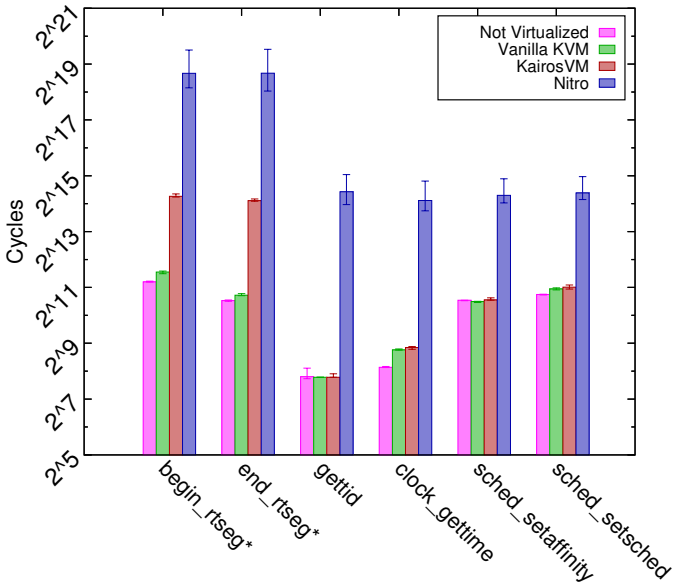

Fig. 7. Latencies of selected system calls (* indicates the system call being trapped/traced in KairosVM/Nitro)

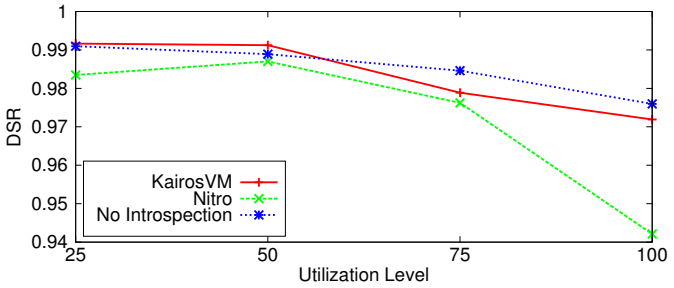

Fig. 8. Deadline Satisfaction Ratio for guest real-time application

repetitions. At each repetition we picked a different taskset with a number of tasks in the range $[20, 60]$.

Figure 8 and 9 show the Deadline Satisfaction Ratio (DSR) as returned by *sched_test_app* on vanilla KVM, KairosVM and Nitro for different utilizations (25, 50, 75 and 100). The graphs refer to deadline misses by comparing physical times. When virtual times are compared instead, we had mostly no misses. This is because the relative time in the virtual machine is not incremented while syscalls are being trapped. Figure 8 shows that the DSR for KairosVM and when no introspection is active are almost similar. On the other hand, the DSR for Nitro is at most 5% lower, especially when the utilization increases. This illustrates the fact that Nitro adds latencies that prevent the completion of the real-time tasks before their deadline. As KairosVM only traps syscalls related to real-time scheduling, the latencies are much lower and the difference between the KairosVM and the no introspection configuration is negligible. The same trends can be observed for the experiment with Bonnie++, in Figure 9, where Nitro never performs better than KairosVM and the configuration with no introspection.

Table I shows Bonnie++'s overall number of syscalls and the number of *write* syscalls done during each experimental run for different configurations. We report numbers for the two higher values of utilization that highlight the decrease in general purpose activities, lower utilization values would

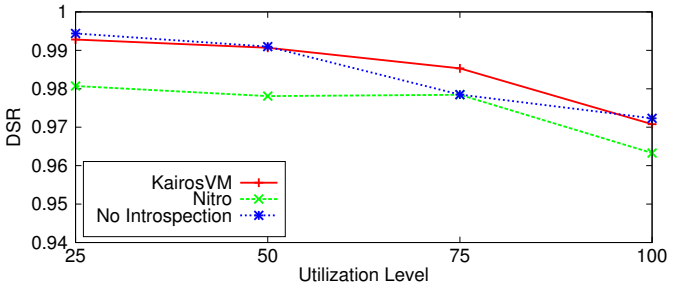

Fig. 9. Deadline Satisfaction Ratio for guest real-time application with Bonnie++ interference

TABLE I. NUMBER OF SYSCALLS AND WRITES ACCORDING TO THE UTILIZATION

Util	# of Syscalls	# of Writes	Case
75	4147015	3479204	KairosVM
	3236017	3175538	Nitro
	⋆ 4153681 ⋆	⋆ 3481426 ⋆	No Introspection
100	4065685	⋆ 3452094 ⋆	KairosVM
	1282703	1282300	Nitro
	⋆ 4066879 ⋆	3452492	No Introspection

add no insight. Results show that the number of syscalls for KairosVM and the no introspection configuration are similar, while the number of syscalls for Nitro differs. The number of overall syscalls and the number of *write* syscalls is decreasing when the utilization grows because less time is allocated to perform these syscalls and deadline misses can happen.

VIII. CONCLUSION

Consolidation of multiple legacy real-time software stacks on the same hardware requires real-time aware virtualization. We present KairosVM, a new hypervisor built to support existing software stacks and guarantee their real-time requirements. As we aim to support legacy systems, we cannot use paravirtualization. We thus introduced a new introspection mechanism in the host operating system to extract the real-time requirements of the guest. Our approach uses undefined instructions to trap into the hypervisor. Evaluations showed that state of the art solutions, like Nitro, are less suitable for real-time because of the overhead generated on each syscall. On the other hand, our introspection solution only traps syscalls related to real-time scheduling, such that the overhead is negligible compared to that of no introspection.

This paper is a first step in building a real-time aware hypervisor where multiple guests can coexist while respecting their real-time properties. We gave some pointers in Section VI on how the information from guests can be used to dynamically schedule the system. This allows the scheduling algorithm to better allocate the processor time when tasks terminate their execution earlier than expected. This could be exploited to improve the schedulability of the system or reduce the energy consumption. In future work, in addition to ChronOS, we would like to support different guests RTOS (e.g. RTAI [3], LITMUS^{RT} [4], *PREEMPT_RT* [5]) through the use of our plugin infrastructure described in Section VI-B.

REFERENCES

[1] S. Xi, J. Wilson, C. Lu, and C. Gill, "RT-Xen: Towards real-time hypervisor scheduling in Xen," in *Proceedings of the Ninth ACM Int. Conf. on Embedded Software (EMSOFT)*, 2011, pp. 39–48.

[2] M. Dellinger, P. Garyali, and B. Ravindran, "ChronOS Linux: A best-effort real-time multiprocessor linux kernel," in *Proceedings of the 48th Design Automation Conference (DAC)*, 2011, pp. 474–479.

[3] "RTAI - the RealTime Application Interface for Linux." [Online]. Available: http://www.rtai.org

[4] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "LITMUS-RT: A testbed for empirically comparing real-time multiprocessor schedulers," in *Proceedings of the 27th IEEE International Real-Time Systems Symp. (RTSS)*, 2006, pp. 111–126.

[5] "CONFIG_PREEMPT_RT patch set." [Online]. Available: https://rt.wiki.kernel.org

[6] J. Pfoh, C. Schneider, and C. Eckert, "Nitro: Hardware-based system call tracing for virtual machines," in *Proceedings of the 6th International Conference on Advances in Information and Computer Security (IWSEC)*, 2011, pp. 96–112.

[7] S. Saewong, R. R. Rajkumar, J. P. Lehoczky, and M. H. Klein, "Analysis of hierarchical fixed-priority scheduling," in *Proceedings of the 14th Euromicro Conf. on Real-Time Systems (ECRTS)*, 2002, pp. 173–.

[8] R. Davis and A. Burns, "Hierarchical fixed priority pre-emptive scheduling," in *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS)*, 2005, pp. 10 pp.–398.

[9] F. Zhang and A. Burns, "Analysis of hierarchical EDF pre-emptive scheduling," in *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS)*, 2007, pp. 423–434.

[10] M. Lee, A. S. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik, "Supporting soft real-time tasks in the Xen hypervisor," in *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2010, pp. 97–108.

[11] P. Yu, M. Xia, Q. Lin, M. Zhu, S. Gao, Z. Qi, K. Chen, and H. Guan, "Real-time enhancement for Xen hypervisor," in *Proceedings of the IEEE/IFIP 8th International Conference on Embedded and Ubiquitous Computing (EUC)*, 2010, pp. 23–30.

[12] J. Lee, S. Xi, S. Chen, L. T. X. Phan, C. Gill, I. Lee, C. Lu, and O. Sokolsky, "Realizing compositional scheduling through virtualization," in *Proceedings of the IEEE 18th Real Time and Embedded Technology and Applications Symposium (RTAS)*, 2012, pp. 13–22.

[13] I. Shin and I. Lee, "Compositional real-time scheduling framework with periodic model," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 30:1–30:39, May 2008.

[14] A. Lackorzyński, A. Warg, M. Völp, and H. Härtig, "Flattening hierarchical scheduling," in *Proceedings of the Tenth ACM International Conference on Embedded Software (EMSOFT)*, 2012, pp. 93–102.

[15] J. Kiszka, "Towards linux as a real-time hypervisor," in *Real Time Linux Workshops (RTLWS)*, 2009.

[16] B. D. Payne, M. De Carbone, and W. Lee, "Secure and flexible monitoring of virtual machines," in *Proceedings of the 23rd Annual Computer Security Applications Conf. (ACSAC)*, 2007, pp. 385–397.

[17] Y. Fu and Z. Lin, "Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2012, pp. 586–600.

[18] "The netwide assembler: Nasm." [Online]. Available: https://courses.engr.illinois.edu/ece390/books/nasm/html/nasmdoc2.html

[19] T. P. Baker, "Comparison of empirical success rates of global vs. partitioned fixed-priority and EDF scheduling for hard real time," Tech. Rep., 2005.

[20] R. Coker, "Bonnie++ benchmark suite." [Online]. Available: http://www.coker.com.au/bonnie++/