Paravirtualizing Linux in a real-time hypervisor

Vincent Legout, Matthieu Lemerre

Paravirtualizing Linux in a real-time hypervisor

Vincent Legout and Matthieu Lemerre
CEA, LIST, Embedded Real Time Systems Laboratory
Point courrier 172, F-91191 Gif-sur-Yvette, FRANCE
{vincent.legout,matthieu.lemerre}@cea.fr

ABSTRACT

This paper describes a new hypervisor built to run Linux in a virtual machine. This hypervisor is built inside Anaxagoros, a real-time microkernel designed to execute safely hard real-time and non real-time tasks. This allows the execution of hard real-time tasks in parallel with Linux virtual machines without interfering with the execution of the real-time tasks.

We implemented this hypervisor and compared performances with other virtualization techniques. Our hypervisor does not yet provide high performance but gives correct results and we believe the design is solid enough to guarantee solid performances with its future implementation.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

General Terms

Design, Security, Performance

Keywords

Linux, virtualization, hypervisor, real-time, hard real-time system, microkernel

1. INTRODUCTION

Embedded systems have processing units often underutilized, loosing processing time which could be used by other applications. This is usually because a real-time system requires determinism, guaranteed response times and keeping the system as small as possible makes it easier to fulfill these requirements.

The Anaxagoros microkernel has been designed to meet these requirements. With security in mind, this system is built to execute hard real-time and dynamic non real-time tasks on the same hardware. But Anaxagoros being a minimalist microkernel, it cannot directly execute applications written for a fully-featured operating system such as Linux. The Anaxagoros design prevent non real-time tasks to interfere with real-time tasks, thus providing the security foundation to build a hypervisor to run existing non real-time applications. This allows current applications to run on Anaxagoros systems without any porting effort, opening the access to a wide range of applications.

Furthermore, modern computers are powerful enough to use virtualization, even embedded processors. Virtualization has become a trendy topic of computer science, with its advantages like scalability or security. Thus we believe that building a real-time system with guaranteed real-time performances and dense non real-time tasks is an important topic for the future of real-time systems.

Linux being as popular as it is, was our first choice when choosing the operating system to be virtualized. It eases the development of the virtualization process and with its large hardware support offers many possibilities for our non real-time tasks.

The remainder of this paper is structured as follows: first, section 3 details the design and implementation of our microkernel, Anaxagoros. Then section 4 dives into the design of our hypervisor. Section 5 presents our current prototype, the current version is able to run at least two virtual machines with non real-time tasks. To finish, section 6 presents the performance of our prototype. We compared some characteristics of the virtualized Linux system with the same Linux system without virtualization.

2. RELATED WORK

Virtualizing Linux has already been done by many, for example KVM [8] and Xen [5], two open source virtualization solutions. In KVM, Linux serves as the hypervisor whereas Xen is the hypervisor and can run on several operating systems including Linux. They both support many guest operating systems. However, their focus is not real-time systems and they do not offer real-time guarantees.

Running Linux on top of a microkernel has for example been done by L4Linux [2], a port of Linux to the L4 microkernel. Its purpose is also to run side-by-side Linux and real-time tasks. MkLinux [3] is another solution to host Linux on a microkernel. ChorusOS or Mach [4] can run multiple OS personalities (e.g. Linux) but do not focus on real-time.

And unlike L4Linux or KVM and Xen, Anaxagoros was not designed to be a hypervisor but a secure microkernel for hard real-time. And we use these security and real-time features to build the hypervisor.

Efforts were also made to support hard real-time in Linux, for example Xenomai [7] or Adeos [1], but these solutions only enhance Linux while our solution allows the use of multiple isolated Linux virtual machines together with real-time tasks on top of a secure microkernel, without relying on Linux.

3. ANAXAGOROS

Anaxagoros is an operating system kernel for safe execution of mixed-criticality hard real-time and non real-time tasks on a single hardware platform. This system strictly complies with resource-security principles. It has been designed and implemented for the x86 architecture. Efforts were made to make the kernel design small and secure, and the kernel currently has only 2587 lines of C and 1155 lines of x86 assembly (measured with sloccount).

Principles and global structure.

The goal of the system is to provide facilities to safely share resources between tasks, i.e. using shared services. It has strong security so as to prevent undesirable task interference, and security covers protection of the real-time requirements of the tasks. The use of shared services by hard real-times tasks is secured and easy.

Various mechanisms have been built in order to ensure that all the requirements are met, but there are out of the scope of this paper. We redirect the reader to [9, 10, 11] for more information about Anaxagoros.

Security and interface.

Anaxagoros is a microkernel, and thus comes with some services (e.g. memory management service). However, Anaxagoros can also be seen as an exokernel [6] because it's API is very low-level and does not abstract physical resources. For example, an user task can create its own address space and control how pages are managed inside this address space. We use the low-level functions to virtualize Linux.

Anaxagoros comes with built-in services (e.g vga, keyboard, . . .). These services can be used by any task and respect all the security principles of the system.

Anaxagoros defines three independent entities : *address space* (separates memory rights), *threads* (separates CPU access rights), and *domains* (separates all other kinds of rights). The usual *task* concept is obtained by combination of one thread, one domain, and one address space.

4. DESIGN

This section details the design of our project. The previous section dealt about Anaxagoros, and we used Anaxagoros features to build a hypervisor in order to run Linux virtual machines. The hypervisor is the piece of software inside Anaxagoros that manages the virtual machines, for example when one virtual machine wants to communicate with the host system. Our contribution is to paravirtualize Linux without changing Anaxagoros.

Paravirtualization.

Virtualization is the idea of creating a virtual operating system that run on top of another operating system. The virtual operating system is called a guest, the other one being the host. In a paravirtualized environment, the guest knows it does not run on hardware unlike full virtualization where the guest behaves like it runs on real hardware.

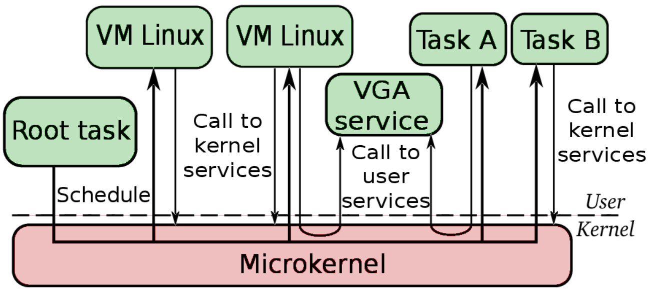

Figure 1: Tasks and virtual machines cohabitation

When a system is virtualized, it does not have full access to the hardware. In a x86 system, a guest usually does not run in the most privileged ring (i.e. where the system needs to be to access all the resources) and thus needs the host to perform those operations. In a full virtualized environment, the guest would try to perform the operation, it would fail and the failure would be captured by the host. On the contrary, a paravirtualized guest can directly tell the host to perform an operation on its behalf, without the need to try the operation first. A communication from the guest to the host is called a hypercall.

We choose paravirtualization because it makes it easier to build an hypervisor (no need to emulate all the hardware in the host). However, it makes it impossible to run an unmodified guest. Paravirtualization is also usually faster than full virtualization.

In our system, the guest system runs in the same protection ring as the user space application. But modern processors (VT-x (vmx) for Intel and AMD-V (svm) for AMD) introduced another ring to ease virtualization and allowing the guest to have a better access to the hardware. We do not use these technologies.

Figure 4 illustrates how native tasks and virtual machines cohabit. Only the microkernel is in kernel space while tasks and virtual machine are in user space and need to communicate with the kernel. The root task is the task responsible for creating the other tasks and the virtual machines. This task, as Anaxagoros exports the definition of the scheduling policy, is responsible for allocating CPU time to the tasks.

Hypervisor.

Inside Anaxagoros, we designed a hypervisor to manage the execution of the guests. This hypervisor is responsible for the following tasks :

- Hypercalls: The hypervisor receives hypercalls from the virtual machines and handle them. According to the type of hypercall it receives, various operations are performed. Once the hypercall is done, the execution returns to the guest.
- Interrupts: When a guest is running, interrupts can happen. It is the responsibility of the hypervisor to redirect these interrupts to the correct interrupt handler. Without virtualization, this task is done with the *Interrupt descriptor table* (IDT), but the guest

cannot update the hardware IDT. When a guest wants to update its IDT, it sends an hypercall to the host. The host then updates the shadow IDT of the guest. The host maintains a shadow IDT for each guest.

- Memory: The guest does not have a direct access to the whole memory. At its creation, an amount of memory pages are dedicated to the guest, pages the guest can access. To build its mapping, the guest needs the host, and thus hypercalls are responsible for creating, updating or deleting page directories and pages tables. The guest can update the CR3 register via a hypercall. TLB flushes are possible with a hypercall and the TLB is automatically flushed when switching between guest and host or between two guests.

Security.

Each virtual machine must be isolated from another and must not interfere with other tasks. To achieve this goal, we put each virtual machine in its own domain, its own address space and in its own thread. The root task, responsible for creating the virtual machine, creates a new domain, a new address space, then prepares the new address space by mapping and copying the Linux code and the switch to the new address space. Thus each virtual machine has its own domain, and is clearly separated from the other tasks and virtual machines.

5. IMPLEMENTATION

The section details the implementation of our hypervisor. The focus of the current implementation was to make it functional, to get insights on what needs to be done in the Anaxagoros kernel and Linux port for an ideal implementation.

5.1 Linux

When building our hypervisor, we wanted to run a traditional operating system in the virtual machine. We choose Linux because it is well-known, it is free software and it offers various commodities to ease the development of our prototype.

Paravirtualized Linux supports three open-source hypervisors: KVM [8], Xen [5] and lguest [12]. Their authors sat down together and build an interface (named *paravirt_ops*) to make it easy to use Linux as a virtualized system. Another interface (*Virtual machine interface*) was earlier proposed by VMware, but never went into the kernel.

We use *paravirt_ops* to ease the development of our hypervisor, thus changes in the Linux kernel are relatively small. It provides function pointers to override functions which must be updated when Linux is paravirtualized. For example, *write_cr3()* is not possible and must be overridden to make a hypercall. Non-modified function pointers keep the same behavior. *paravirt_ops* is a useful interface to port Linux to a new hypervisor, using it is simple and it keeps the number of lines of code added small.

The current port of Linux is based on the lguest port. The core modifications are located on a *arch/x86/anaxagoros* folder with two files: *boot.c* and *i386_head.S*. *boot.c* contains most of the *paravirt_ops* functions and *i386_head.S* contains *anaxagoros_entry()*, the function called to initialize the *paravirt_ops* functions pointers. A header file located in *include/linux/* has also been added.

Others modifications are in *arch/x86/kernel/head_32.S*, to add Anaxagoros as a possible host, and in *kernel/printk.c* to tell *printk()* to use our own function to print a kernel message (thus Linux needs to be built against a static library provided by Anaxagoros).

We used Linux 2.6.36 without any additional patches other than those needed to add Anaxagoros as a host. Other versions of Linux may work if they provide *paravirt_ops*. We introduced a new configuration option: *ANAXAGOROS_GUEST* which must be activated in order to recognize Anaxagoros as a possible host. Building Linux with *make allnoconfig* and adding manually the following options is the easiest way to create a working configuration: *EMBEDDED*, *PHYSICAL_START* (0x400000), *PHYSICAL_ALIGN* (0x100000), *CONFIG_BLK_DEV_INITRD*.

5.2 Boot

This paragraph explains how the virtual machine loads Linux and starts. First, the root task creates a new domain with its new address space. The Linux ELF executable (vmlinux) is loaded into the address space at the correct location. One page is filled with a *boot_params* structure containing the setup parameters. Finally, it jumps to the Linux entry point (found in the ELF file).

Linux can then boot. During the boot, Linux will try to perform lots of operations it is not authorized to do. This is where the hypervisor enters the game.

The important part is to update the *hdr.hardware_subarch* entry in the *boot_params* structure to tell Linux it is going to be virtualized on an Anaxagoros host. Without this change, Anaxagoros code in Linux will not be executed and Linux will not boot. In order to recognize Anaxagoros as a potential host, we added a new entry in the *subarch_entry* list (file *arch/x86/kernel/head_32.S*), and when Linux detects that the *hdr.hardware_subarch* field matches Anaxagoros, it jumps to our own initialization function *anaxagoros_entry()*. This function performs the initialization hypercall so that the hypervisor can tell Linux which physical pages are accessible. Linux can then initialize all the *paravirt_ops* functions so that paravirtualization is set up.

5.3 Hypervisor

This paragraph details how the hypervisor is implemented. The userspace application (root task) is less than 600 lines of code and the kernelspace hypervisor less than 450.

Hypercall.

When the guest cannot make an operation by itself, it sends an hypercall to the host which does the operation and returns to the guest. We have twelve hypercalls. Table 1 lists the most important ones.

A hypercall is a software interrupt. A new entry in Anaxagoros's IDT has been created, arbitrarily at 0x30 (48). This number is chosen arbitrarily and must be greater than 31. A hypercall has five arguments (because the x86 architecture has five registers available: *eax*, *ebx*, *ecx*, *edx* and *esi*). The first argument is the hypercall number, the other depend on the hypercall. Once the hypercall finishes its task, it returns to the virtual machine using the *iret* instruction.

Interruption.

Interrupts are more complicated. Because we want to

Table 1: Hypercalls list

Name	Description
CPUID	Return processor information
LOAD_IDT_ENTRY	Update shadow IDT
READ_CR2	Return CR2 register content
READ_CR3	Return CR3 register content
WRITE_CR3	Update CR3 register
SET_PTE	Set page table entry
ALLOC_PTE	Allocate a new page table
RELEASE_PTE	Release a page table
SET_PDE	Set page directory entry
ALLOC_PDE	Allocate a new page directory
RELEASE_PDE	Release a page directory
FLUSH_TLB	Flush the TLB
DISABLE_IRQ	Disable guest interrupts
ENABLE_IRQ	Enable guest interrupts

Table 2: Linux segments

Index	Macro	Selector
12	__KERNEL_CS	0x63
13	__KERNEL_DS	0x6b
14	__USER_CS	0x73
15	__USER_DS	0x7b

know which virtual machine must receive the interrupt, we need to analyze the interrupt. We do that by keeping in the host which virtual machine is currently running, so that it is easy to redirect the interrupt.

The hypervisor should also deal with the guest virtual IDT, the IDT sets by the guest so that the host knows where it must deliver the interrupt. When a guest wants to update its IDT, the operation is replaced by an hypercall. A virtual IDT is never loaded into the IDT, it is just kept inside the host memory.

Segmentation.

Linux makes a limited usage of the segmentation, memory protection is essentially done via pagination. Thus our implementation of the segmentation is quite simple: Linux's segments are fixed offline and cannot change. Table 2 shows the four segments used by Linux. The two first segments are supposed to be on privileged ring zero but are here on ring three (*0x63* and *0x6b* instead *0x60* and *0x68*).

Other segments that must be handled are TLS segments. They change on every context switch and a hypercall allows Linux to update the *Global Descriptor Table*. TLS segments are not necessary to boot Linux but are used for example by the glibc to handle threads.

Pagination.

Linux must be configured to only use two-level paging (i.e. PAE must be disabled).

Each guest has an arbitrary amount (n) of memory pages for its own use. For the guest, these pages are numbered from 0 to $n - 1$. From the host point of view, these pages can be in different places in memory, because a translation between guest physical page numbers and host virtual page numbers is done each time a guest wants to update its page table.

When the guest wants to update its mapping, it uses hypercalls. To do so, we updated lots of functions located in the *pv_mmu_ops* structure (e.g. *write_cr3* or *set_pte*). The host does not track how the guest manages its pages, there are no virtual page directories or tables inside the host.

5.4 User space

User-space applications can use a ramdisk loaded into the memory at runtime. Inside, we tested several applications with success:

- Busybox: Statically built, busybox can be used inside the ramdisk. It offers a shell and various utilities.

- Other applications can be built with the klibc, a minimal standard C library.

- Services: Application can use Anaxagoros's services like the vga service. The Linux vga driver is not yet functional.

- Network: Network support is partly provided. If linux is built with the network driver, we can use the Linux driver to communicate with the outside world even if Anaxagoros does not support any network card yet. It should be possible to extend this feature to support other devices in the virtual machine. This feature is much appreciated because Linux comes with a large hardware support and it makes it possible to use the drivers in a Anaxagoros system.

  There are two options to load the driver, one can build it inside Linux so that the Linux version running has immediate support for the network card or ship the module into the ramdisk. The module can then be loaded if *busybox* is built to support modules (i.e. with the *modprobe* utility).

5.5 Limitations

Some features are not yet available or do not work properly on the guest Linux system:

- Clock: The clock management is minimal. Currently, the clock of the virtualized Linux is not correct, it starts at 0 and does not run fast enough because clock interrupts are not always delivered.

- I/O: Our hypervisor currently does not support any I/O virtualization. This feature could be added, but not until Anaxagoros supports more hardware than it does today.

6. PERFORMANCE

The status of the hypervisor and Anaxagoros is of a functional prototype. The focus of this initial prototype has been first security, second simplicity, but even if the implementation of many parts is not optimized, many operations already have correct performance.

We performed two kinds of tests, one to illustrate the performance of the hypervisor and one to show the real-time properties of Anaxagoros. We used both the Bochs PC simulator and real hardware and we give performance results for these two situations. However, we did not try the network driver with real hardware.

Table 3: Virtualization performance

	syscall	rand	ctx	fork	pipe	tcc
Native	71	138	0.74k	91k	4.51k	231M
Virt.	308	141	1.59k	1969k	6.25k	284M
Native	490	167	13.83k	774k	33k	458M
Virt.	2640	188	17.1k	2589k	38.9k	563M

Table 4: Real-time performance

	minimum	maximum	standard deviation
nop loop	$-1.2 * 10^7$	$6.3 * 10^7$	$1.2 * 10^7$
Linux VM	$-1.3 * 10^7$	$7.8 * 10^7$	$2.9 * 10^7$

6.1 Virtualization

The following results show how fast the virtualized Linux is compared to an unvirtualized Linux running on bare metal.

We used two configurations, the Bochs PC simulator (Bochs does not simulate cache, and thus executes one instruction per cycle) and a Dell Precision Workstation 650 with an Intel Xeon (3.06 GHz) (512 KB L2 cache, 1 MB L3 cache) and 4 Go of DDR-SDRAM (266 MHz).

In table 3, the five first columns are representative lmbench tests, where each column gives the number of cycles required for the operation. The tcc test consists in a shell script executing the Tiny C compiler compiling itself 3 times.

The tests were run on Bochs (first two lines) and on the Dell Workstation (last lines). For each configuration, we tested two situations: first-line is with a non-virtualized Linux and second is Linux running inside Anaxagoros. For each simulation, we used the same Linux and userspace applications.

6.2 Real-time

Anaxagoros is a real-time system, and one of the reasons we build it is to have real time guarantees for the real-time tasks when they run together with non real-time tasks (e.g. Linux virtual machines).

To illustrate this point, we ran one real-time task and one non real-time task. We dedicate a fixed amount of CPU time to the real-time task on a periodic basis and this task is incrementing a counter. For the behavior of this task to be predictable, the final value of the counter must be the same for each execution, or the difference as small as possible. We ran this test in two situations, the non-real time task being a nop loop or a Linux virtual machine.

Table 4 represents the difference of the minimum and maximum counter increments relatively to the mean value (which was $\approx 5 * 10^8$), and the standard deviation on the Dell machine. For the 200ms timeslice, this represents an average variation of 5%, and a 18% maximum variation.

However on Bochs, the minimum and maximum values of the counter are respectively 3349028 and 3349102 (difference is 74) while the mean value is 3349072. The standard deviation is 29. The second thread workload had few effects on the execution time of the first thread. The maximum variation of workload relatively the 200ms timeslice is 22.1 ppm ($22.1 * 10^{-6}$), which is very low.

The high variations on the x86 can be explained by the presence of caches, system management mode, etc. But the Bochs measurements demonstrate that excellent results could be achieved with deterministic hardware, which validates our approach.

7. CONCLUSIONS

Paravirtualizing Linux in a real-time systems proved to be not as difficult as we thought, mostly because Linux has mechanisms to ease the porting on a new hypervisor. We obtained a working prototype capable of running two Linux virtual machines in parallel with non real-time tasks. Tests have been run to compare the performance of the virtualized Linux with a non-virtualized Linux and results showed decent performance.

Future work will concentrate on improving the Anaxagoros/Linux interface. For instance, currently Linux uses a custom system call (with a custom software interrupt) for many hypercalls, rather than the standard capability invocation mechanism [10] used in Anaxagoros. We need to extend Anaxagoros with a "virtualization service" that Linux could use.

Also, we need to use and improve the Anaxagoros mechanism to batch hypercalls. Currently, Linux needs to do one hypercall per modification inside a page table, which makes virtual memory operations extremely slow.

Finally, there are many work left to do with Anaxagoros; among them is support for multicore computer, and an effort we have started to use formal methods to prove the security-related parts of Anaxagoros.

8. REFERENCES

[1] Adeos. http://home.gna.org/adeos.

[2] L4linux. http://os.inf.tu-dresden.de/L4/LinuxOnL4.

[3] Mklinux. http://www.mklinux.org.

[4] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for unix development. 1986.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03. ACM, 2003.

[6] D. R. Engler. The design and implementation of a prototype exokernel operating system. Master's thesis, Massachusetts Institute of Technology, 1995.

[7] P. Gerum. Xenomai - implementing a rtos emulation framework on gnu/linux. 2004.

[8] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In *Ottawa Linux Symposium*, July 2007.

[9] M. Lemerre. *Intégration de systèmes hétérogènes en termes de niveaux de sécurité*. PhD thesis, Université Paris Sud - Paris XI, 10 2009.

[10] M. Lemerre, V. David, and G. Vidal Naquet. A communication mechanism for resource isolation. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, 2009.

[11] M. Lemerre, V. David, and G. Vidal Naquet. A dependable kernel design for resource isolation and protection. In *Proceedings of the First Workshop on Isolation and Integration in Dependable Systems (IIDS'2010)*, Paris France, 2010. ACM.

[12] R. Russel. Lguest: Implementing the little linux hypervisor. In *Ottawa Linux Symposium*, July 2007.