TESTING EMBEDDED SYSTEMS SOFTWARE USING OPEN SOURCE VIRTUAL PLATFORMS

Andrea Bastoni
SPRG - University of Rome "Tor Vergata"
Roma, Italy
bastoni@sprg.uniroma2.it

Patrizio Boschi
patrizio.boschi@gmail.com

Fabrizio Batino, Christian Di Biagio, Luca Recchia
MBDA-Italy S.p.A.
Roma, Italy
{fabrizio.batino, christian.di-biagio, luca.recchia}@mbda.it

ABSTRACT
This paper studies the use of open source virtualization systems to perform functional testing of embedded systems software directly on developer machines. Problems related to software testing on embedded platforms are analyzed and an open source virtual target platform architecture is presented. A development process is devised to extend the standard emulated device set of virtualization solutions to include specific-purpose virtual devices. A prototype implementation of the proposed architecture is presented and functional as well as performance issues are analyzed.

KEY WORDS
Virtualization, testing, embedded systems

1 Introduction

The commonly used software development process is a difficult task to perform when the developed software has to be run on an embedded system.

The main difficulties lie in the testing phases, as typical target machines differ sharply from development machines, which are usually normal consumer boxes.

Furthermore, due to cost, availability or developing reasons, the target system may not always be ready by integration or testing time. These factors cause heavy bottlenecks when multiple developers work on the to-be-developed application and need to concurrently access the target system for testing and integration.

1.1 Motivation

When solutions like remote testing, cross-testing and target platform assignment among developers are not viable due to, for example, the above-mentioned constraints, the problem of *application testing* on a target platform can be faced by using a *virtual platform* (or *virtual machine*, *guest machine*) based approach.

The *virtual platform* approach to embedded software testing could be particularly successful in all those industrial contexts where hardware is co-developed with software, and hardware platform prototypes cannot be promptly available to software developers. By using a *virtual target platform*, every developer could access one or more virtual target platforms directly on his/her workstation (usually a desktop PC). This could greatly reduce testing and integration phase duration, as real target assignment problems (and subsequent bottlenecks) can thus be avoided.

Moreover, the virtual machine (virtual target) can be configured to *exactly* reproduce the target configuration, and specific hardware devices, usually available only on the real target, can be fully emulated in the virtual machine. Therefore, virtual machines can provide earlier availability and better accessibility to the target platform, thus allowing an increase in productivity.

1.2 State of the art

At present, existing commercial virtualization-based solutions are well established and used for embedded software development and testing.

Virtutech Simics for Virtualized System Development [18, 22] and *CoWare Virtual Platform* [9] are well-known commercial products for hardware and software integration and testing on virtual target platforms. Commercial VMware [15] solutions are also common in the industrial world to perform testing on generic x86 target platforms (i.e., no specific hardware devices are needed on the target platform).

The open source world does not have a tailored solution for embedded software development yet, but it offers a full range of virtual machines: Xen [23], KVM [20], QEMU [5], KQEMU [4], Linux VServer [21] and OpenVZ [13] are common and widespread products. However, not all of them can be successfully employed to perform functional testing of embedded software or to emulate specific hardware behaviours. In fact, many of these products are developed for completely different purposes (e.g., server consolidation, enforce applications isolation...).

The rest of this paper is organized as follows. In sec-

tion 2, we present requirements and objectives of virtual target platforms. In section 3, we analyze two virtualization technologies that can be exploited to realize virtual target platforms. In section 4, we present our virtual platform architecture and we tackle non-functional and performance problems of the architecture; in this section we also describe our virtual device development process. In section 5, we present a prototype implementation of the architecture, we show how functional testing can be performed on virtual platforms and we discuss prototype performance. In section 6, we briefly present debugging techniques on open source virtual platforms. We draw conclusions conclude in section 7.

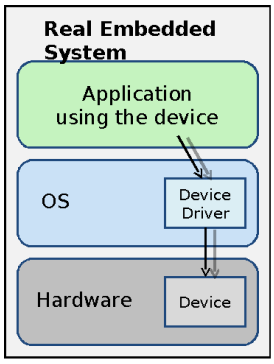


Figure 1. Real embedded system schema

2 Goals and requirements of a target platform emulator

2.1 Accurate hardware and driver abstraction

The main requirement of a virtual platform is to *faithfully* emulate specific-purpose target platform hardware (CPU(s), memory, devices). This emulation can be performed on developer boxes using virtualization technologies.

Furthermore, the operating system configuration of the virtual platform (virtual machine *and* specific purpose hardware emulation) should closely match the configuration of the target platform. In particular, the operating system *driver set* should be identical. The need of a coherent driver set is due to two different key factors: *driver bugs* and *driver programming interface* implementation.

As pointed out by some authors [3] and also as confirmed by our experiments [7], the `/driver` section of the Linux kernel accounts for the *majority of the bugs* of its whole source code. If the virtual target platform uses a different driver sets from the real target platform, then there is a nonnegligible probability that target driver-related faults will not be detected while testing the application using the driver set of the virtual target platform.

In different driver sets, drivers could also considerably change as the *driver programming interface* is not very strict about the features that should be implemented.

2.2 Increase target platform availability for functional testing

The target platform is the best place for performing functional and non-functional application tests. Unfortunately, it is often unfeasible to assign a real (embedded) target platform to each developer and, therefore, the target platform must be shared among developers, leading to heavy bottlenecks that could reduce productivity.

The direct availability of a *virtual target platform* on the developer's box decreases the amount and the effort of functional testing that should be performed on the real target platform. Bottlenecks in the scheduling of the real target platform (Figure 1) can be greatly reduced if the developer is able to perform most of testing phases on virtual embedded systems (Figure 2), interleaving testing on the platform and code editing (using her / his favourite programming tools).

3 Platform virtualization and emulation

Platform virtualization and emulation are broad terms referring to the process of hiding the physical resources of a computing platform to the operating system, and of exposing instead an abstract or emulated (possibly different) platform.

Solutions like paravirtualization (e.g. Xen paravirtualization [17]) and operating system level virtualization (e.g. OpenVZ [13], Linux VServer [21]) can guarantee good performance in many contexts (e.g. server consolidation), but these solutions create a platform abstraction that is similar yet *not* identical to a real one.

Paravirtualization and operating system level virtualization are therefore unsuitable for the aims of this article. In fact, it would be meaningless to perform functional testing on a virtual platform whose devices and Board Support Package (BSP) drivers differ from the real one.

Instead, *system emulation* and *full virtualization* technologies (briefly described in next paragraphs) allow for a faithful emulation of real devices and real drivers and are therefore suitable solutions to our problem.

3.1 System emulation

System emulation allows to simulate both hardware and software components of a system. When performing system emulation, all CPU instructions executed by the virtual machine are *translated and emulated* on the host CPU. Furthermore, every low-level operation on hardware devices is simulated as well.

In spite of the huge performance overhead inherently implied by this technology, system emulation gives the opportunity of faithfully simulating the behaviour of a complete physical platform.

Examples of open source system emulators are Bochs [12] and QEMU [5], which emulate x86 32 and x86 64 architectures. QEMU supports additional host and guest system based on SPARC, PowerPC, MIPS and ARM architectures. QEMU also comes with a set of emulated device models, thus allowing to run non-modified guest systems which rely on widespread devices or legacy devices.

3.2 Full Virtualization

A significant performance improvement with respect to system emulation can be achieved by avoiding the translation of all CPU instructions.

Full Virtualization technologies allow the complete decoupling of virtual machines applications and kernel from the physical hardware (as in system emulation), but also allow virtual machines to directly execute most of their instructions on physical CPU(s).

Virtual machines native execution is only possible if the virtual machine (*guest*) architecture is identical to the host (*Virtual Machine Monitor*) one. Given the widespread diffusion of the x86 architecture, virtualization of x86 systems over x86 hosts is the most common form of full virtualization.

The two main techniques used to achieve full virtualization of guest machines are *trap-and-emulate* and *binary translation* [1]. In 2006, AMD [2] and Intel [16] started developing architectural changes in their CPUs to support an easier virtualization on x86 architecture. This changes are commonly referred to as *hardware support to virtualization* and are included in most modern x86 processors.

VMware [15] and Parallels [19] are commercial products that use full virtualization techniques. Xen HVM [23] and KVM [20] are open source solutions which use full virtualization by exploiting the "hardware support". KQEMU [4] uses full virtualization and directly executes guest user-code on the real hardware, while it runs guest kernel-code through a QEMU emulation.

4 Open source virtual platform architecture

The proposed virtual platform architecture makes use of a layered structure (Figure 2) to decouple developer's machine hardware from the virtual hardware seen by each virtual embedded system.

The platform uses open source virtualization solutions (KVM, KQEMU, Xen HVM and QEMU) to implement the core virtualization layer, while we extended the emulated device set to include specific-purpose hardware device emulations. This allows to show to guest operating system a faithful emulation of the embedded platform hardware.

In particular, we extended the QEMU virtual device set, as all the analyzed virtualization solutions exploit QEMU device model to realize emulated device abstractions. The virtual device set needs to be extended as

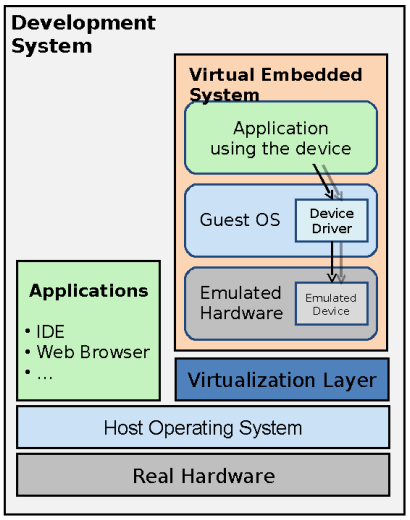


Figure 2. Virtual platform system schema

QEMU only offers a limited choice of emulated hardware devices and these devices are mostly *COTS* hardware devices, while specific-purpose hardware devices found on embedded platforms are not available.

Therefore, most of the difficulties in realizing an open source virtual platform lie in the development of suitable device emulations that can be exported in virtual embedded systems. However, by firstly developing the device emulator in QEMU, it is possible to perform this process only once, as QEMU virtual device can be easily adapted to fit other virtualization solutions

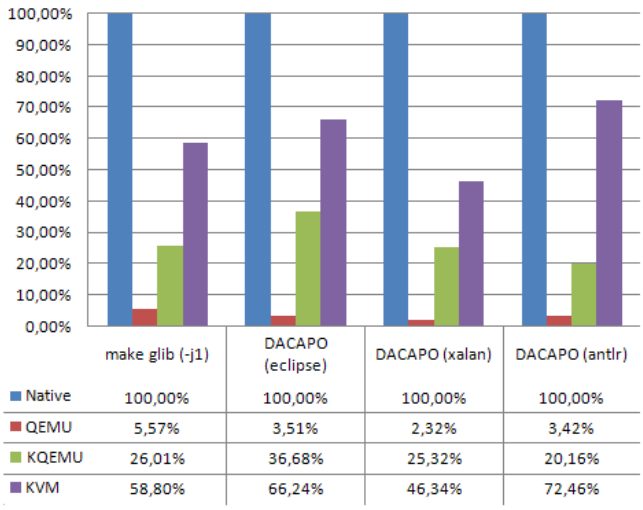
4.1 Performance and non-functional aspects

Through the virtualization of the physical (embedded) platform we mainly aim at performing *functional* software testing; yet, there are situations where *non-functional* and performance-related aspects must be taken into account in order to perform correct testing. In those situations, the huge overhead paid by QEMU as system emulator may invalidate the testing of the software and therefore the use of virtual platforms is not advisable. However, if full virtualization techniques are employed, some non-functional and performance testing activities can still be performed on virtual platforms.

To obtain an overview of the performance loss of system emulation with respect to full virtualization, we performed a series of tests using application-level benchmarks (*glibc-2.18* building, DaCapo benchmark[6] and Whetstone benchmark [10]).

In the *glibc*-building benchmark the `glibc-2.18` source is compiled (using `gcc-4.1.2` compiler) through the `make all` command.

The DaCapo benchmark suite (2006-10-MR2) "consists of a set of open source, real world applications with non-trivial memory loads"[14] which executes in-



	make glib (-j1)	DACAPO (eclipse)	DACAPO (xalan)	DACAPO (antlr)
Native	100,00%	100,00%	100,00%	100,00%
QEMU	5,57%	3,51%	2,32%	3,42%
KQEMU	26,01%	36,68%	25,32%	20,16%
KVM	58,80%	66,24%	46,34%	72,46%

Figure 3. Native, QEMU, KVM, KQEMU performance with `glibc` build and DACAPO benchmark

side a Java Virtual Machine (JVM). We have selected from the DaCapo benchmark suite the benchmarks `antlr`, `eclipse`, `xalan`.

The Whetstone benchmark generates a computationally-intensive workload by means of floating point operations.

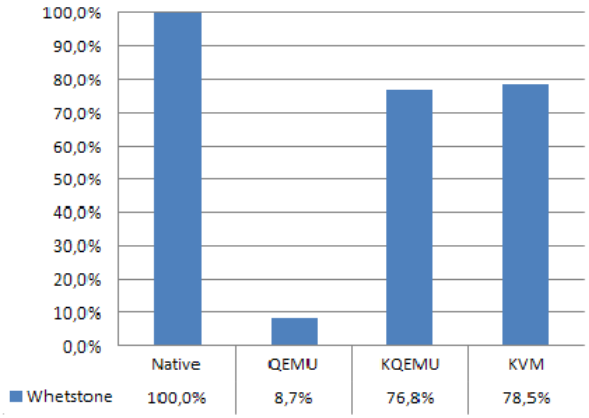
The tests are realized on a developer machine with an Intel Core2 Quad Q6600 (2.4 GHz) processor, 1 GB RAM DDR2 and 250GB SATA disk. The operating system is a Gentoo Linux with 2.6.24 kernel.

As it can be seen in Figure3 and 4, KVM is able to reach between 50 and 80% of non-virtualized (native) performance, while KQEMU only reaches between 20 and 40% in many benchmarks.

However, the full virtualization solutions always perform better than system emulation. In fact, the results obtained by QEMU are significantly lower if compared with KVM and KQEMU: system emulation solutions pay the huge overhead of instruction emulation and QEMU never obtains more than 10% of non-virtualized performance. A more detailed and complete survey of the performance of the analyzed virtualization solutions can be found in [7].

Virtualization technologies also affect the performance of virtual (emulated) devices. In fact, nominal performance of virtual devices equal those of real devices, while real performance are heavily influenced by the virtualization solution in use [8].

Although the very high performance overhead, QEMU is, at the moment, the only viable open source choice when the target architecture is not the x86 architecture. QEMU can in fact (as system emulation technology) emulate ARM, SPARC, MIPS and PowerPC guests over x86, x86_64 and PowerPC hosts. In addition, if same devices are supported by different architectures, then QEMU emulation of the devices only needs a minimal effort to be



	Native	QEMU	KQEMU	KVM
Whetstone	100,0%	8,7%	76,8%	78,5%

Figure 4. Native, QEMU, KVM, KQEMU performance using Whetstone benchmark

ported on guest with different architecture.

We do not delve futher into the issues related to emulation of different architectures as our target systems are Single Board Computers (Concurrent Technologies VP417) with an Intel Core 2 Duo L7400 processor, and developer boxes are x86 desktop PCs. We can therefore exploit full virtualization techniques to achieve better performance than system emulation.

A limitation of the proposed virtual platform approach is that it cannot be used when precise timing estimations (e.g. testing of hard real-time applications) are required for performing software testing. In fact, full virtualization technologies and system emulators such as QEMU can only reproduce the "*semantics*" of the requested guest instructions, while precise timing estimations would require also the emulation of the precise "*syntax*" of the real instructions. To solve these kind of problems, cycle-accurate simulators (such as PTLsim [24]) should be used.

4.2 Using QEMU for virtual device development

As already introduced in previous sections, QEMU is an appropriate choice for the development of device emulators. Unfortunately, QEMU does not offer a clean development framework — development process, device-model-aided design, IDE — such as those of commercial counterparts (e.g., Virtutech offers a Device Modeling Language (DML) [11] to ease the development of device models). However, QEMU provides a complete set of API `C` functions that can be successfully exploited to develop complete device emulations.

Next, we will define a development process that can be used in the development of virtual devices in the QEMU environment.

Development strategy When realizing device emulations, the main objective is to develop a device abstraction that faithfully simulates the hardware device features; in

particular, the OS guest driver should not be able to spot any differences in the interaction between emulated or real device.

We consider the real device as a *black box* with a clear interface, which is generally well documented in either device datasheets or in the programmer's reference manual. The main components of this interface are in fact the hardware registers and the interrupt request lines offered by the real device.

The virtual device is then developed as a QEMU module which exposes the same interface of the real device, but that implements the device logic at *software level* through *callbacks* registered with QEMU API initialization functions. For instance, in the emulation of a PCI device, a callback is called whenever a virtual machine guest driver tries to access a register (PCI Configuration Register, Memory Mapped I/O Register) or an I/O port of the device. In the device emulation (QEMU host side), registers are in fact simulated using normal memory locations (from simple variables to complex structures and arrays). Every access to an emulated register is intercepted to decide whether *side effects* have to be emulated.

The development of a virtual device can often be simplified by ignoring all register transfer level details and subtle hardware level logic (thus reducing the amount of device hardware documentation to read) and by realizing in the emulation only the main hardware logic.

We have identified a three-phases development strategy (Figure 5).

1. A simple *stub* of the real device is built; this stub should at least be detected by the guest operating system. The stub generally includes the internal state of the device and all the registers (PCI, memory, etc.) that could be accessed by the operating system.

2. The driver — device communication infrastructure (command handling and callback registration) is added.

3. The device *core logic* features are developed and linked to internal state, registers and command handlers.

Writing a device emulator with strict functional and quality requirements can be a difficult task when complex devices are involved. At emulation level, complexity is mainly related to the interception and simulation of the side effects due to register access and bit-field modifications. We have found it useful to develop first only the most important features of the device, while adding non-core functionalities in a second phase.

5 Prototype platform using the Intel Watchdog 6300ESB device

We have successfully applied the strategy above in the development of a device emulator of the Intel Watchdog timer

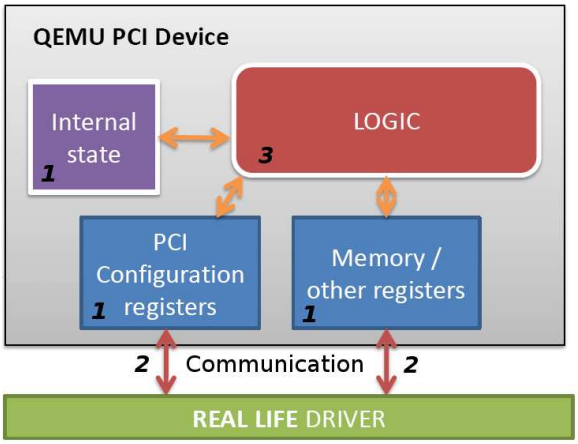


Figure 5. Development process using QEMU

(WDT) 6300 ESB found on our target x86 platforms and we have integrated the emulated device in the analyzed virtualization solutions. The result is a virtual platform architecture which is able to emulate WDT operations directly on developer's boxes.

The watchdog timer is a countdown hardware timer device that can be (re)programmed by the OS at periodic time intervals. If the OS fails to do so and the watchdog countdown timer reaches zero, the WDT reacts with some pre-programmed actions (e.g. rebooting the machine).

The watchdog timer is a good example of a device which is commonly found on most target platforms, but on none of the developer platforms; without a virtual platform, application testing which involves the WDT cannot be performed on developer platforms and need to be done on the target platform.

The watchdog emulator is developed by using QEMU. We create a full replica of the target OS configuration and filesystem in a QEMU virtual machine and we use KQEMU and KVM to speed up QEMU execution. Tests using Xen HVM are currently ongoing.

As already noted in section 4.1, the use of a full virtualization technology to speed up the execution of QEMU device emulators does not affect the development of the device emulators themselves.

The WDT PCI Configuration Registers are implemented through simple arrays of variables, which are initialized with values taken from the datasheets, from the target machine (e.g. `lspci`) or from the reverse engineering of the Linux driver. All the arrays are included in a `C` structure which represents the watchdog device in the emulation. This structure also includes an amount of information related to WDT logical status and available resources (e.g. the current value of the countdown timer, a pointer to QEMU virtual timers...).

When the virtual target platform boots up, QEMU calls our device emulator C init-function, which in turn initializes the device and registers all the callbacks with

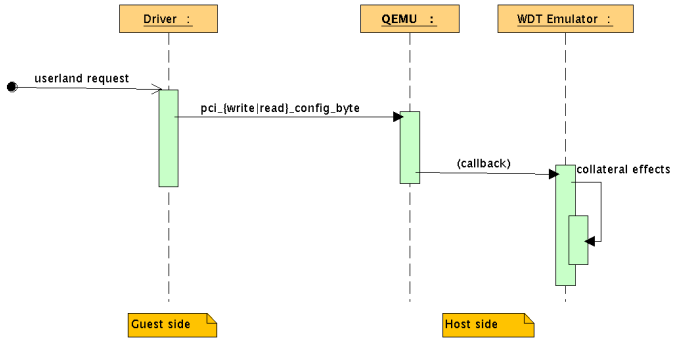


Figure 6. Execution flow of a request to WDT emulator

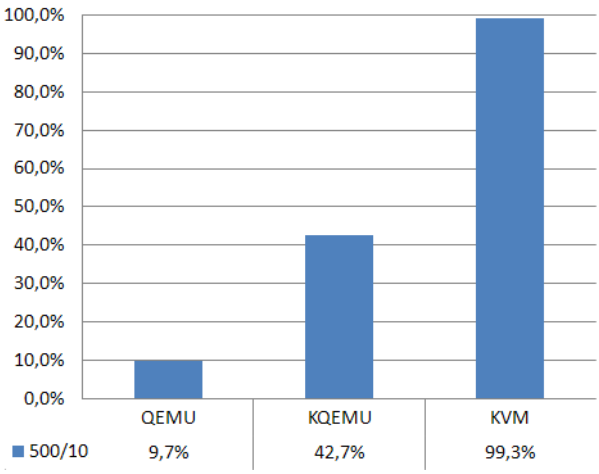


Figure 7. QEMU, KQEMU, KVM performance using WDT benchmark

QEMU. QEMU does not offer a mechanism to dynamically load a device model after virtual platform initialization and any new device initialization function should be statically called from inside QEMU boot function or should be indirectly called by adding a new command line parameter.

The WDT hardware timer logic is simulated by using software timers. Their time resolution ($\approx$ 100 ms) is in fact more than sufficient to effectively support application requirements. The typical execution flow of a request directed to the WDT emulator is shown in Figure6.

The relatively low complexity level of the WDT hardware (only 24 registers) has led to a lightweight watchdog emulator which can fully emulate all the features of the real hardware. The size of the WDT emulator is approximately 600 effective lines of code (SLOCs) with 150 lines of code comments. The ratio between Linux WDT 6300ESB driver size (around 360 SLOCs) and WDT emulator size complies with the estimated ratio between the size of Linux device drivers and their correspondent emulation in QEMU [7].

Testing The testing of the platform including the newly developed device emulator can be accomplished either from the *host* side or from the *guest* side. In the former (in the QEMU environment), the device emulator is seen as a component of the QEMU system and the tests performed here are mainly unit testing and debug of the various developed functions.

On the guest side, the device emulator is seen as a real device integrated in the virtual platform and this testing phase can be carried out only through the exported device interface.

The testing of the device through its interface is generally done in two steps: first, we use ad-hoc user space applications which indirectly test all device emulation features through the OS driver interface. In detail, the user-space application opens the device front-end (`/dev/watchdog` node) and repeatedly issues all the `ioctl` commands supported by the Linux driver and by the WDT device. For each command, the behaviour of the emulated WDT device is compared to the behaviour of the real watchdog timer.

In the second phase of the testing process, we run the application suites commonly used to access real watchdog timer device features. Without a virtual platform, these applications need to be run on the real platform, while we were able to reproduce the identical real WDT behaviour on the virtual platform running on developer's machine.

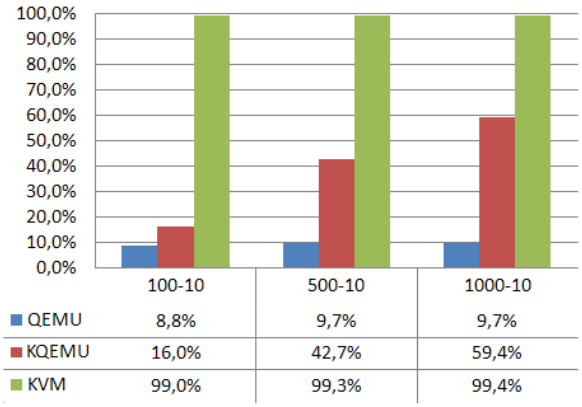
Performing testing activities on the virtual platform is just as easy as performing them on the real platform. In fact, the testing of the WDT emulator allowed us to discover a bug in the Linux watchdog timer driver which caused a non correct reinitialization of one register value upon rebooting.

Performance aspects Although performing functional testing on virtual target platform is our main target, we evaluated how different virtualization technologies affect performance of the platform.

We developed a benchmark that emulates an application using WDT. The benchmark performs one million runs, each run being composed by 500 floating point userspace operations (emulating the behaviour of Whetstone benchmark) and 10 kernelspace operations on the WDT device (10 register operations). It is important to underline that 10 register operations are a high number of operations for the WDT device as applications generally perform only 2 - 3 operations on the WDT.

The benchmark measures the minimum execution time needed to perform one run. The minimum execution time has been chosen as this is the value obtained when the test is not preempted during its execution. One run execution time in the non-virtualized system takes $64\mu s$, on average.

Figure 7 shows performance (normalized against non-virtualized Linux) of QEMU, KQEMU and KVM virtualization solutions. A comparison of Figure 4 and Figure 7 shows that operations on the virtual device does not substantially change the performance of KVM and QEMU virtualization solutions. In fact, KVM performance are very close to non-virtualized one (the lower KVM performance



	100-10	500-10	1000-10
QEMU	8,8%	9,7%	9,7%
KQEMU	16,0%	42,7%	59,4%
KVM	99,0%	99,3%	99,4%

Figure 8. QEMU, KQEMU, KVM performance using WDT benchmark (Increasing ratio of userspace operations)

in Figure 4 can be explained as our floating point load is less demanding than Whetstone's load of 50000 floating point operations), while QEMU slowdown (90%) is similar to the slowdown obtained in the Whetstone benchmark. KQEMU performance is interesting as KQEMU can directly execute only userspace instructions, while it needs to emulate all kernelspace instructions (access to WDT is performed in kernelspace). Therefore, as can be seen in Figure 8 (which shows performance for an increasing number of userspace operations — 100, 500, 1000 floating point operations, 10 device operations) KQEMU performance increases as the number of floating point userspace operations increases, while the performance of both KVM and QEMU remains stable. Thus KQEMU performance is heavily influenced by the number of userspace operations performed, while, as KVM and QEMU, is not affected by operations on the virtual WDT .

6 System debugging

On real target platforms it is often possible to perform system debugging through JTAG-like technologies. On virtual platforms, virtual hardware debugging can be easily accomplished by directly accessing the internal emulator state: it is for example possible to anytime investigate an emulated register status (a variable in the host machine).

Commercial solutions for virtual platform emulation offer advanced features for guest (virtual platform) debugging and analysis. For instance, Virtutech provides a mechanism (*reverse execution*) to *walk back* to a previous system state; this is particularly handy when something goes wrong and a trace of the faulting execution is needed.

QEMU does not offer the same debugging features of its commercial counterparts, but it provides a useful *GDB stub*, which makes it possible to perform guest system and on-line *kernel* debugging by using the classical GDB approach: a GDB session is started on the host machine (developer's box) and is connected to the GDB stub exported by the guest machine.

7 Conclusions and future work

Our work has tackled the problem of performing embedded software testing on target platforms by means of open source virtualization solutions. We have analyzed the problems of performing testing phases on target embedded platforms and we have shown how virtualization technologies can speed up the software development process by allowing functional testing to be performed directly on developer's boxes.

Our approach is novel in that we propose a virtual platform architecture based on open source virtualization solutions to decouple developer's machine hardware from the hardware seen by virtual embedded platforms. We have shown how QEMU can be used to develop specific-purpose hardware abstraction that can successfully exploited in the emulation of embedded target-systems, and we have presented a development process that can be followed to implement such virtual devices in QEMU.

We have also analyzed how to integrate QEMU device model in faster full virtualization systems such as KQEMU and KVM (on x86 architecture) and we have proposed a prototype platform with embeds a device emulator of the Intel Watchdog timer 6300ESB. This prototype can be effectively exploited to perform functional testing of applications that use the WDT.

Performance issues related to the use of virtualization technologies in software testing have also been analyzed and we have shown how virtual platform architectures are generally unsuitable for non-functional and performance testing as they introduce high overheads.

We have underlined how QEMU and open source virtualization technologies still lack a clean development framework and well documented APIs to reduce the device emulators' developing effort.

In the near future we plan to better investigate performance aspects of the target emulation. We are also evaluating Xen/PTLsim integration to perform cycle-accurate simulation of target x86 platforms and the use of open source virtualization solutions to emulate multicore physical targets.

References

[1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. *SIGOPS Oper. Syst. Rev.*, 40(5):2–13, 2006.

[2] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, September 2007.

[3] Andy Chou et al. An empirical study of operating system errors. In *Symposium on Operating Systems Principles*, pages 73–88, 2001.

[4] D. Bartholomew. Qemu: a multihost, multitarget emulator. *Linux J.*, 2006(145):3, 2006.

[5] F. Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proc. of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[6] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, New York, NY, USA, 2006. ACM.

[7] P. Boschi. Test di software per piattaforme embedded su sistemi virtualizzati open source. Master's thesis, University Roma Tre, 2009.

[8] F. L. Camargos, G. Girard, and B. D. Ligneris. Virtualization of linux servers: a comparative study. In *2008 Ottawa Linux Symsposium*, pages 63–76, July 2008.

[9] CoWare. http://www.coware.com/.

[10] H. J. Curnow, B. A. Wichmann, and T. Si. A synthetic benchmark. *The Computer Journal*, 19:43–49, 1976.

[11] J. Engblom. Virtutech DML Device Modeling Language. March 2009.

[12] K. P. Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux J.*, page 7.

[13] OpenVZ. http://openvz.org/.

[14] the DaCapo benchmark suite. http://dacapobench.org/.

[15] VMware Inc. http://www.vmware.com.

[16] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel(r) virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3):167 – 178, August 2006.

[17] P. Barham, et al. Xen and the art of virtualization. In *SOSP '03: Proc. of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[18] P. S. Magnusson et al. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[19] Parallels. http://www.parallels.com/.

[20] Quamranet Corp. Kvm: Kernel-based virtualization driver. White Paper.

[21] Stephen Soltesz et al. . Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, 2007.

[22] Virtutech. http://www.virtutech.com/.

[23] Xen. http://www.cl.cam.ac.uk/research/srg/netos/xen/.

[24] M. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. *Performance Analysis of Systems and Software, IEEE International Symmposium on*, 0:23–34, 2007.