1st International QEMU Users' Forum

Alpexpo

Grenoble, France

March 18th, 2011

Proceedings

Editors:

W. Mueller, U. Paderborn, Germany
F. Pétrot, TIMA, France

Table of Contents

Introduction

QEMU, as a fast open source software emulator, has reached significant acceptance in the embedded software development community. QEMU is nowadays used by many groups, and among them, several have also developed various extensions, like for example support for timing/power analysis and for cosimulation with SystemC.

The QEMU Users' Forum (QUF) 2011 will provide attendees with a forum to exchange ideas and to join forces for future activities. It is planned as a workshop to initiate discussions and to additionally provide beginners with basic QEMU information and an overview of some current advanced researches and applications through a set of invited presentations.

After a general introduction to QEMU, the invited speakers present their activities focusing on HW/SW cosimulation based on the QEMU and SystemC technologies. The afternoon is dedicated to short presentations with different applications from various international groups and experts. We hope that we have compiled an attractive program for everybody and wish you a workshop with fruitful discussions.

W.Mueller, U. Paderborn, Germany

F. Pétrot, TIMA, France

QEMU PROJECT OBJECTIVES AND TECHNICAL OVERVIEW

Nathan Froyd
CodeSoucery
9978 Granite Point Court
Granite Bay, CA 95746
United States
froydnj@codesoucery.com

Alexander Graf
Suse Linux Products GmbH
Maxfeldstr. 5
90409 Nürnberg
Germany

ABSTRACT

QEMU started out as an open source emulator for x86 Linux binaries. In the past eight years, QEMU has grown to include emulation of most major CPU architectures, system emulation capabilities, and virtualization capabilities.

We will discuss the changes implied by QEMU's rapid growth, its current architecture, and its prospects for future development.

QEMU AND SYSTEMC

Màrius Montón i Macián

CEPHIS
Escola d'Enginyera
Universitat Autonoma de Barcelona (UAB)
marius.monton@uab.cat

GreenSocs
http://www.greensocs.com
marius.monton@greensocs.com

ABSTRACT

QEMU is a powerful Virtual Platform with a great open community and lot of processor models, platforms and devices. We present our work on adding SystemC language as specification language for devices to this Virtual Platform. We present two different approaches to this joining, one oriented to systems with few SystemC devices (typically one peripheral described in SystemC and the rest of the system in QEMU) and one oriented to complex SystemC systems where QEMU is a small part of the simulation.

As conclusion, we will launch the idea of built a community around QEMU and SystemC where all participants of QUF or others developers could join it to share their contributions.

Combined Use of Dynamic Binary Translation and SystemC for Fast and Accurate MPSoC Simulation

Marius Gligor and Frédéric Pétrot

System-Level Synthesis Group
TIMA Laboratory
Grenoble, France 38031
{Marius.Gligor, Frederic.Petrot}@imag.fr

Abstract. In this paper, we present a simulation strategy that tries to combine the speed of the binary translation based ISSes with the accuracy of the event driven simulators. To have an accurate timing behavior for an instruction set simulator based on binary translation, we had to first solve timing issues in processor modeling, second define fast and precise cache models, and third solve the synchronization issues due to the different models of computation used in the ISSes and in the rest of the system.We have experimented our proposal using processors models provided by the QEMU framework to replace the existing ISSes and SystemC TLM as simulation environment for the whole platform.

1 Dynamic binary translation bases. QEMU

The binary translation represents the emulation of the instruction set of a processor by the instruction set of another processor using code translation. QEMU is a fast and portable emulator which emulates many architectures (X86, ARM, SPARC etc.) and runs on several architectures. To simulate guest systems, QEMU uses the dynamic binary translation. QEMU uses micro-operations as intermediate representation (IR) in the target to host code translation process. These micro-operations represent the instruction set of a simple 2-address abstract machine. This machine has three general purpose registers. For each of these micro-operations, a C function is hand coded and a unique integer identifier is associated with. These micro-operations are compiled to an object file.

Fig. 1 presents the QEMU simulation model. QEMU verifies if the sequence of target instructions starting at the address given by the program counter of the simulated processor has already been translated. In the case it was not translated before, the *binary translation* stage begins. The instruction corresponding to the program counter of the simulated processor is *fetched* from the target binary code. The fetched instruction is then *decoded* into several micro-operations whose identifiers are concatenated into a micro-operations identifier buffer. If the current instruction is not a branch instruction, the next target instruction is fetched and decoded. The binary translation stage ends after a branch instruction is decoded. The sequence of target instructions together treated by the binary translation stage forms a translation block. Once the block translation is finished, the *tiny code generator* generates a host function which is made of the

concatenation of the micro-operations compiled code corresponding to the identifiers stored in the buffer. The resulted host function is stored into a *translation cache entry* of the translation cache. Once the generated code is *executed*, the simulator verifies the existence of the translation corresponding to the new program counter. If the required translation already exists in the translation cache, it is directly executed.

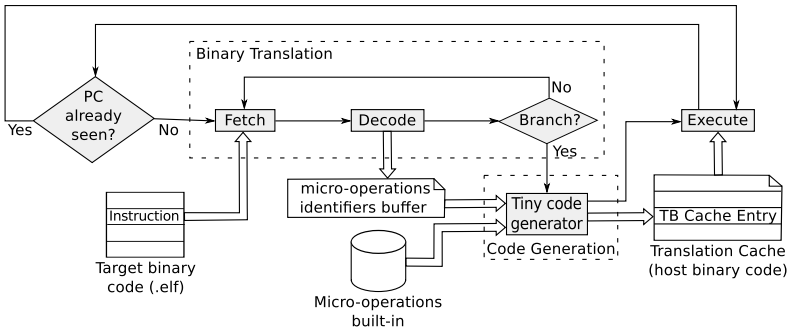


Fig. 1. QEMU simulation model

2 Multiprocessor modeling

Our simulator is presented from the multiprocessor, time, frequency and energy modeling points of view.

The goal of the binary translation simulators is to be very fast and functionally correct, but not to provide information about hardware execution time. Therefore, the implementation of multi-processor platforms simply call the processors interpretation function one after another in a predefined order. The simulation order of the processors is always the same. As our goal is to estimate timings, we need to have a mean to control (and measure) the scheduling of the processor execution. To do so, the default behavior of the initial simulators must be modified, at the price of reducing the simulation speed.

2.1 ISS wrapping and connection

For each processor in the platform, we instantiate a SystemC module wrapper (*iss_wrapper*) as depicted in Figure 2. The execution of each processor is performed in the context of the SystemC process (*SC_THREAD*) of its wrapper. This way, the processors are simulated concurrently. It is necessary that each processor is simulated in the context of its own SystemC process, so that they are simulated independently under the control of SystemC.

In order to avoid the binary translation of the same target code for each simulated processor, the processors that may share the same translation cache are encapsulated into a SystemC module, called *iss_group*. To ensure that the host code generated for a translation block is correct for all processors in the group, the processors in a *iss_group* must be identical.

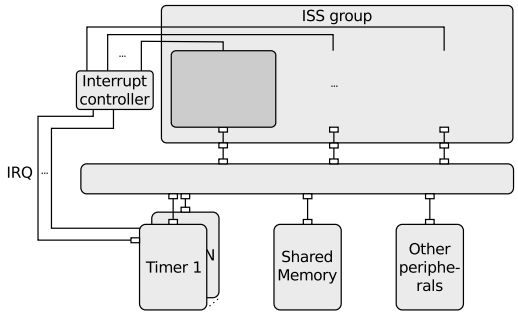


Fig. 2. QEMU - SystemC simulation platform

The processor wrappers are connected to an interconnect, through which they can communicate with other hardware components (traffic generator, timers, interrupt controller, memory, spinlocks *etc.*) also connected to it. All hardware components are implemented as SystemC modules. The interrupt lines of the different components (*e.g.* timers) are connected to the processor wrappers, which implement the interface between SystemC interrupt wires and the ISS. From the initial simulator, our approach uses only the processor models with, if required, their MMUs (Memory Management Unit). All other devices are externalized and implemented as SystemC modules for accuracy reasons.

2.2 SystemC synchronization points

In our model, a processor is simulated while it does not communicate with the world behind its caches and the initial simulator does not stop it. When an instruction/data cache miss or an I/O occurs, the processor simulation stops and the processor wrapper synchronizes itself with the rest of the SystemC platform by consuming the time (SystemC *wait* function) required by the real processor to execute the instructions simulated since the last synchronization.

So, unlike the cycle accurate simulators where the cycles of the ISSes are simulated concurrently, the proposed binary translation based ISS simulate a group of cycles before synchronizing with SystemC and allowing other ISSes to execute. The number of cycles in these groups may vary from one to many cycles, depending on the simulated target code. The simulation of these cycles takes zero SystemC time.

The synchronization also takes place after a predefined period of time without synchronization. For the target processor instructions designed for the synchronization of the software running on a SMP architecture, a SystemC synchronization should also be generated (*e.g.* exclusive load and store).

The processors simulation order depends on the time consumed by the processors at synchronizations. A synchronization condition may occur at any time during the simulation of a translation block (*e.g.* cache miss), thus the unscheduling does not respect anymore the border of the translation blocks.

3 Time modeling

The binary translation based simulators do not have a time notion for the simulated platform. For the timeout events required by the timer devices, they usually use different external mechanisms and sometimes asynchronous to the simulated platform. For instance, QEMU uses one of the host timers (hpet, rtc etc.), that generates an alarm signal on the host machine.

Instead of using the host time, our platform uses the SystemC simulation time. The timer interrupts are generated by the SystemC timer modules. For time accuracy of the processor model, a few changes have been made to the initial simulator in order to model the time required to execute instructions. These changes are performed by adding information when the target processor binary code is translated into the host processor binary code.

The first annotation tries to make simulated processors accurate from the point of view of **the time internally required for instructions execution**. When the target binary code is translated, we add a micro-operation before the micro-operations of each translated target instruction. The added micro-operation increments the number of cycles of the instructions that have been simulated since the last synchronization. The increment uses the number of cycles of the instructions given by the datasheet of the simulated processor.

Another modification applied to the initial simulator for improving the time accuracy consists of presence verification and loading of the required cache line into the **instruction cache**. The verification and the loading are done in a function called by a micro-operation inserted at the beginning of each translated translation block and, inside a translation block, before the first instruction of each instruction cache line. In case of an instruction cache miss, the currently simulated processor is synchronized with the rest of the SystemC platform by consuming the time required to execute the number of cycles obtained by the first annotation. After the synchronization, the processor wrapper sends a request over the interconnect for a burst transfer from memory of the implied cache line.

A similar modification refers to the main memory read/write data accesses. Each time a main memory location is read, its presence in the **data cache** is verified. The mechanism described for the instruction cache miss is also applied in the case of a data cache miss. The **I/O accesses** to SystemC modules also generate a SystemC synchronization followed by a request in the SystemC subsystem.

4 Experimental results

Table 1 presents the speedup and the accuracy of our simulator compared to a cycle accurate simulator using SOCLIB components and to a TLM simulator using interpretative SOCLIB ISSes and the rest of the hardware components from our simulator.

Table 1. Speedup and accuracy of our simulator vs. other simulators

	Simulation speedup	Accuracy
Cycle accurate simulator	300X	92 %
TLM simulator using interpretive ISS	2X	100 %

QEMU/SYSTEMC COSIMULATION AT DIFFERET ABSTRACTION LEVELS

Markus Becker
University of Paderborn/C-LAB
Fuerstenallee 11, 33102 Paderborn
markus.becker@c-lab.de

KEYWORDS

Embedded Software, Real-Time Operating Systems, Hardware-dependent Software, Simulation, Transaction-Level Models, Dynamic Binary Translation

ABSTRACT

As today's embedded software applications have a fast growing complexity early modeling and verification is a crucial to save design costs and time-to-market as well as to increase the software quality. This is especially applies to safety critical real-time properties which require the use of dedicated Real-Time Operating Systems (RTOS). Modeling such systems is always a trade-off between simulation accuracy and simulation performance.

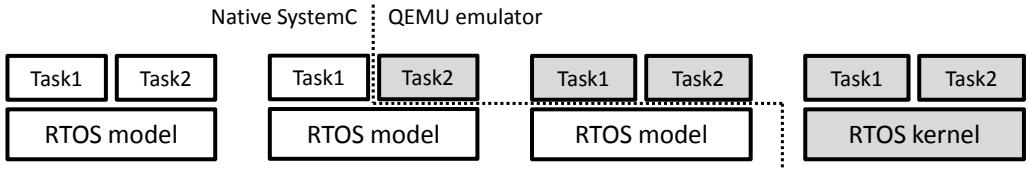


Fig. 1: QEMU/SystemC-based Refinement of Real-Time Software.

We present a mixed QEMU- and SystemC-based cosimulation environment for embedded real-time software [BZM10, MOZB10, BMX+10]. The environment comprises different levels of real-time software abstraction starting from a purely native source-level model for very early and fast simulations which can be refined towards an accurate but yet very efficient CPU specific model (see Fig.1). Modeling of timing aspects starts from coarse-grained source-level software segments with statically analyzed delay annotations towards dynamic cycle approximate estimations of the CPU specific binary code.

For this, our abstract Real-Time Operating System model in SystemC (aRTOS) [ZMG09][BZMU09] is combined with QEMU software emulator. For software refinement, we apply the two QEMU emulation modes. The QEMU system mode emulates a full CPU and its I/O for the execution of complete software stacks including operating system and drivers. In contrast, QEMU user mode emulates a CPU in user mode for the execution of single user programs on top of the host system. Existing QEMU/SystemC cosimulation environments are usually interfaced through the register accurate I/O emulation of QEMU's system mode. To provide a smoother refinement from the purely native model towards the full system emulation, we introduce an intermediate step interfacing QEMU and SystemC at the RTOS API. For this, system calls from the QEMU user mode are trapped and handled by the native SystemC RTOS model. Thus, the execution of the actual RTOS and device drivers can be avoided postponing the decision for a concrete RTOS implementation by the additional advantage of a very fast cosimulation at the same time. We outline the efficiency of our approach by first results comparing simulation speed with accuracy.

REFERENCES

[BZM10] M. Becker, H. Zabel, W. Mueller. A Mixed Level Simulation Environment for Stepwise RTOS Software Refinement. In L. Kleinjohann, B. Keinjohann (eds.), IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES 2010), Springer Verlag, Dordrecht, September 2010.

[MOZB10] W. Mueller, M. Oliveira, H. Zabel, M. Becker. Verification of Real-Time Properties for Hardware-Dependant Software. HLDVT2010, Anaheim, Juni 2010.

[BMX+10] M. Becker, W. Mueller, T.Xie, F. Fummi, G.Pravadelli. RTOS-Aware Refinement for TLM2.0-based HW/SW Design. DATE 2010, Dresden, March 2010.

[ZMG09] H. Zabel, W. Mueller, A. Gerstlauer. Accurate RTOS Modelling and Analysis with SystemC. In: W. Ecker, W. Mueller, R. Doemer (eds.) "Hardware Dependent Software - Principles and Practice", Springer Verlag, Dordrecht, January 2009.

[BZMU09] M. Becker, H. Zabel, W. Mueller, U. Kiffmeier. Integration abstrakter RTOS-Simulation in den Entwurf eingebetteter automobiler E/E-Systeme. In: Proceedings of MBMV'09, Berlin, Germany, March 2009.

[QEMU] QEMU Homepage. http://www.qemu.org.

[SystemC] SystemC Homepage. http://www.systemc.org.

Timing Aspects in QEMU/SystemC Synchronization

Davide Quaglia[1], **Franco Fummi**[1] **Maurizio Macrina**[2], **and Saul Saggin**[2]

[1] University of Verona, 37134, Italy
[2] EDALab s.r.l., 37134, Italy

Abstract. Modeling complex embedded platforms requires to co-simulate one or more CPUs, connected to hardware devices, running applications on top of an operating system. This work presents a QEMU-based co-simulation framework in which hardware devices can be either mapped on the actual hardware of the co-simulation host (e.g., an Ethernet chip) or modeled in SystemC, when they represent ad-hoc IP-cores under design. The work presents the timing aspects to be addressed to synchronize QEMU, executing software components, and a SystemC simulation kernel.

1 Introduction

In the design of ever complex embedded systems, a major task is handling several platforms consisting of di□erent processors and operating systems as well as a large amount of HW devices such as memory, DSPs, and I/O interfaces. CPU emulation, as the one provided by QEMU [1], can be used to reproduce the behavior of target processors while HW description languages and their simulation environments such as SystemC [2] can be used for the simulation HW devices under design. The reasons of the choice are: *1)* **QEMU already supports the use of host-mapped devices** *2)* **SystemC supports the HW description at many abstraction levels, and** *3)* **QEMU source code is available and easy to understand and modify.**

Figure 1 shows the framework described in this work. QEMU hosts both the user application and the operating system; they access the hardware components by using device drivers and interrupt service routines which read and write device registers through the memory-mapped I/O. The SystemC HW registers are mapped on specific memory locations and read/write operations on this locations are re-directed to SystemC. Also interrupt signals generated by SystemC models are sent to QEMU.

Communication between QEMU and SystemC simulator is established through inter-process communication primitives (i.e., a socket) between a new component of QEMU, named *QEMU-SystemC Wrapper***, and a modified version of the SystemC simulation kernel; the exchanged messages have the purpose not only to transmit data and interrupt signals but also to keep the simulation time synchronized between the simulation kernels.**

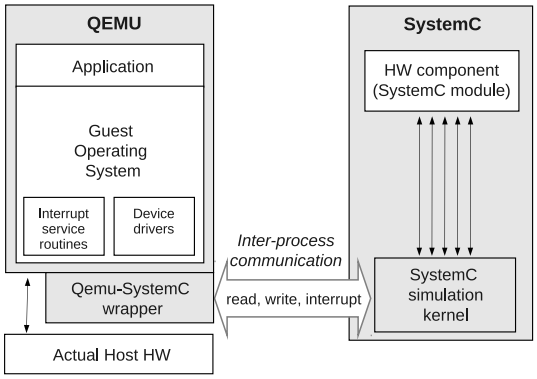


Fig. 1. The proposed co-simulation framework.

2 Synchronization

Each tool, i.e., QEMU and SystemC, contains a part of the whole simulation model and performs simulation on it; tools are executed concurrently by the host operating system. Therefore a synchronization mechanism is needed to assure that the same simulation time is kept by both tools. For instance, if a simulator reads/writes data from/to another simulator whose simulation time is di□erent, then the e□ect of such data in the first/other simulator may not be correct. Synchronization is based on the concept of *shared location* **which is seen as a memory area by SW components in QEMU and as a HW register by SystemC modules. In the rest of the Section di□erent synchronization approaches are presented. All of them are based on the exchange of messages between simulators through a socket; each message contains the unique identifier of the shared location, the command (e.g., read value, write value, interrupt number), the simulation time of the simulator which generated it and the data (in case of write operation). Synchronization is performed by using blocking read/write operations on the socket. In fact, a blocking socket operation can interrupt simulation both in QEMU and in the SystemC kernel.**

2.1 Exact approach

In the exact approach each shared location between the SystemC model and the QEMU guest SW component is considered as a variable shared between two concurrent processes inside an operating system. Only one simulator is running while the other is blocked before an instruction which involves co-simulation. The running simulator executes until a co-simulation event is to be scheduled; at this point it blocks itself and notifies its simulation time to the other simulator which is blocked on another co-simulation event; the simulation times are compared and the simulator with the lower one is re-started and performs the corresponding co-simulation event while the other is blocked. This approach assures the perfect

```
while( T < Tmax )
{
  read messages coming from the other peer
  and enqueue them (together with local events)

  pick all the next events from the queue
  with the same timestamp and execute them (for
  each executed event which is old, issue
  a warning notification to the user)

  for each executed event for which a message has
  been sent to the peer, wait for its response

  update T to the timestamp of the executed event
}
```

Fig. 2. Pseudo-code of the synchronization algorithm (Approach 1).

alignment of simulation times but it requires that just one tool is running at each time. The next approaches allow concurrent execution at the cost of more complex algorithms; therefore, they can fully exploit multi-core architectures.

2.2 Approach 1

Figure 2 shows the pseudo-code of the first approach of the synchronization algorithm; it is implemented both in the SystemC kernel and in the QEMU Wrapper. Both instances of the algorithm start by reading messages coming from the other peer tool. The corresponding events are put in a time-sorted queue. In SystemC kernel this queue also contains internally-generated events either involving internal modules or requiring to send a message to the peer. Then, the peers execute all the queued events belonging to the same time. An event to be executed could be old if it comes from the other peer whose simulation time is lower. In this case, its timestamp is updated to the current time before execution and a warning notification is issued to the designer since simulation output may not be correct. Finally, both tools wait for an acknowledge for each request of a read or write operation and increase the simulation time. In case of read command, the acknowledge also bears the requested value. If a tool receives a request with a lower simulation time, it sends immediately the acknowledge. If the request has a higher simulation time, the receiver serves it when the same simulation time is reached; since the acknowledge is delayed the tool which made the request is forced to wait thus leading to its realignment.

2.3 Approach 2

The second approach is based on the first one but introduces a further mechanism to re-align simulation times. If a tool has a higher simulation time than the one

of the received command, it sends immediately the acknowledge and issues a warning as in Approach 1; then it blocks itself after having asked the requester to be notified when it reaches the same simulation time.

2.4 Approach 3

To use this synchronization approach each tool has to know the timing of future events involving data exchange with the other tool. This approach is based on the definition of some synchronization checkpoints. The first checkpoint is fixed in the setup phase, tools exchange the time of the next event involving co-simulation, i.e., read/write/interrupt operations. A special message (called NEXT in the following) is used. The tool receiving a NEXT message records the time of the other-peer next event in its event queue.

After this setup phase the tools start execution till the first checkpoint. At the checkpoint, the enqueued NEXT message blocks the receiver waiting for the read/write message of the other tool. The other tool sends the corresponding message, computes the next co-simulation event, sends a new NEXT message and waits for a NEXT message coming from the co-simulation peer to update its queue with the new checkpoint. On the other side, after handling the event, the tool waits for the NEXT event message of the peer, computes its next co-simulation event and sends a NEXT event message.

Once tools have updated their event queue with the new NEXT messages, the next synchronization point is defined and both events executed till the next checkpoint.

In case of QEMU, the evaluation of the next co-simulation event requires the analysis of the code to be executed looking for instructions involving read/write operations on shared locations. The presence of the cache of the already translated code simplify this task. In case of SystemC, only the next event in the queue is known and it could not be a co-simulation event. For this reason, a *virtual clock* **has been introduced to generate periodic synchronization events inside SystemC simulator; if the next event is not a co-simulation event, the virtual synchronization event is used. The virtual clock frequency is set by the user in SystemC at the beginning of the co-simulation session; the higher is the frequency, the slower is the simulation but a too low frequency could lead to misalignment of the simulation times. The right trade-o depends on the simulation model.**

References

1. QEMU Emulator, http://fabrice.bellard.free.fr/qemu.
2. "IEEE Std 1666 - 2005 IEEE Standard SystemC Language Reference Manual," *IEEE Std 1666-2005*, pp. 1–423, 2006.

Program Instrumentation with QEMU

Christophe Guillon

STMicroelectronics
christophe.guillon@st.com

Abstract. An instrumentation interface for the QEMU [1] cross-platform emulation tool has been developed along with some associated instrumentation plugins. This interface has proven useful for program analysis and optimization in a cross development context.

1 Introduction

QEMU [1] is an emulation tool used in a number of applications such as virtualization, processor simulation or Linux process emulation.

In addition to these use cases, QEMU has the potential to be a powerful program instrumentation and analysis tool such as some existing profiling, or bug detection tools. Though, with QEMU, these capabilities can be made available to cross-platform development in addition to native development.

Indeed, the process used by QEMU for executing the emulated program or system is a two steps dynamic binary translation: first the guest machine code is translated into a generic intermediate representation, then, this representation is in turn translated into some host machine code before execution. At the intermediate representation level, one can insert machine independent instrumentation operations. In addition, one can provide an interface such that the user can develop an external instrumentation plugin independently of the QEMU core.

2 Contribution and Related Work

The contribution of this work is:

- to bring program instrumentation capabilities to QEMU through dynamically loadable plugins, allowing both instrumentation and cross-execution;
- to develop some typical plugins for program analysis;
- to report use cases of actual development with QEMU user-mode in a cross development context for several targets (Intel x86, ARM v7, ST40).

Concerning program instrumentation, Intel developed Pin [2], a sophisticated dynamic instrumentation tool which features a large number of program analysis plugins. Valgrind [3], which is also a dynamic instrumentation tool, is widely used under Linux thanks to its powerful memory leak detection plugin. The objectives of these tools are comparable. Both were developed first on Intel processors and it

is possible to port and run them on other architecture such as ARM. Though, by leveraging on QEMU, instrumentation is made possible also for cross-platform development. I.e. one can develop on a host Intel machine for a different target platform such as ARM or ST40 and cross-validate, cross-debug or cross-optimize thanks to the powerful and retargetable QEMU binary translation technology.

Concerning previous work on instrumentation with QEMU, Gligor and al. [4] developed an interface with a co-simulation environment. By using a generic instrumentation interface, the development of such a bridge to an external process could be decoupled from the QEMU core framework.

3 Instrumentation Interface

The proposed instrumentation interface is based on a user provided loadable library, an instrumentation plugin, that defines several optional entry points that are activated by QEMU during the translation and execution of the guest application.

Depending on the type of information required by the instrumentation plugin the user can interface with QEMU in mainly two ways:

Execution Time The simplest way is to provide a block_execution_event() function. In this case the user plugin is called at each translated block execution. For instance, generating an execution trace is done by emitting the block program counter and size from this function.

Translation Time Another way is to define the block_translation_event() function called after each guest code block is decoded into the intermediate representation and before the actual code generation takes place. In this case the user plugin can insert instrumentation code directly into the translation block. For instance, generating a trace of memory access addresses consists in inserting effective address dumps at each `load` or `store` operation.

4 Applications

The following instrumentation plugins have been developed in the context of application development and optimization with QEMU user-mode:

- a *trace plugin*, that features full program counter trace or symbolic function trace, used for application debugging,
- a *profile plugin*, that generates function profiles for application tuning or profile directed compiler optimizations,
- a *cache simulation plugin*, used to generate instruction cache conflict information for the compiler or instruction cache statistics for the user.

The QEMU user-mode emulation environment with the proposed instrumentation capabilities and the previously mentioned plugins have been used intensively for the development of several applications for some embedded targets.

The framework was used for the TraceMonkey[1] Just-In-Time compiler ST40 cross-development and its cross-analysis between ST40 and Intel platforms.

The environment was also used for the development of an Open64[2] C/C++ ARM compiler validated by cross-executing and instrumenting benchmark suites.

Last, it allowed cross-development and cross-optimization of an embedded browser based on the WebKit[3] engine for both ST40 and ARM targets.

Validation on actual hardware evaluation boards for these applications was necessary only very lately, for the qualification phase.

5 Experiments

An evaluation of the performance impact of the instrumentation interface has been done on a set of benchmarking applications from the Spec2000 [5] suite.

A very simple plugin that counts the number of executed instructions in the emulated program has been implemented. The objective here is to measure the overhead of the instrumentation interface itself, hence the minimalist plugin. Figure 1 shows that, on an Intel i386 architecture, the process emulation (labeled *qemu*) of the set of benchmark applications is measured as 15 times slower than the native execution (*native*). Using the *Execution Time* interface (*offline*) for calling our counter update function adds a 30% overhead to the bare emulation time. Using the *Translation Time* interface (*inline*) to inline the counter update directly into the translated code divides by ten the instrumentation cost, reducing to only 3% the total overhead. Sometimes, the instrumentation overhead cannot be differentiated from the measurement noise.

Both method are acceptable compared to the emulation cost itself. The later method is quite efficient at reducing the overhead and can be used for lightweight instrumentation such as simple counters.

6 Future Work

The plugin mechanism needs to be activated also in QEMU system-mode in order to profile kernel code or to instrument system level execution.

The instrumentation interface surely needs some extensions or improvements. A natural way to improve this is to develop new plugins, for instance a memory checker or a bridge to a co-simulation framework.

Some typical services such as program symbol resolution could be provided by the QEMU core, as of now, the symbol table loading and the resolution must be done by the plugin itself.

A regression test suite has to be developed in order to ensure robustness of the instrumentation interface with respect to multi threading or system call reentrency for instance.

[1] http://wiki.mozilla.org/JavaScript:TraceMonkey
[2] http://www.open64.net
[3] http://www.webkit.org

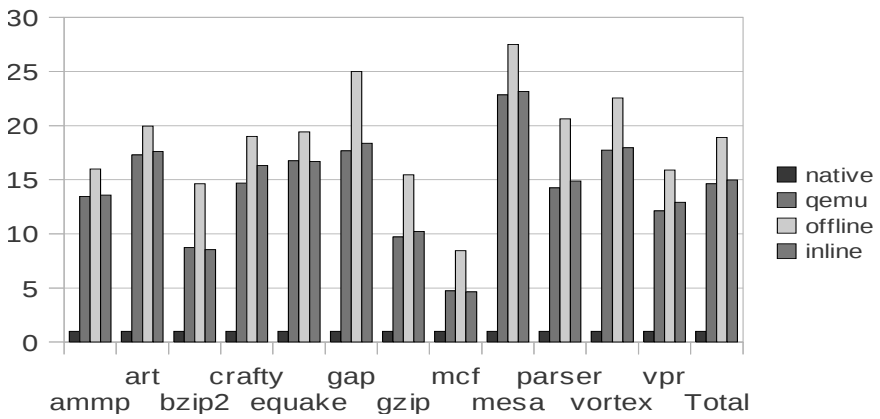


Fig. 1. Normalized execution time for some benchmarks compiled for i386 and emulated with QEMU-i386. Time is normalized to the first set, the native execution time; the second set is the bare QEMU emulation time; the third set is the emulation time with an offline counter update function (*Execution Time* instrumentation); the last set is the time with and inlined counter update (*Translation Time* instrumentation).

7 Conclusion

An instrumentation interface has been developed and used in the context of application cross-development and optimization with QEMU user-mode.

The interface was found useful for cross-developing several applications on several platforms.

Future work should bring more use cases for this cross-instrumentation tool.

References

1. Bellard, F.: Qemu, a fast and portable dynamic translator. In: Proceedings of the annual conference on USENIX Annual Technical Conference. ATEC '05, Berkeley, CA, USA, USENIX Association (2005) 41–41
2. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. PLDI '05, New York, NY, USA, ACM (2005) 190–200
3. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. PLDI '07, New York, NY, USA, ACM (2007) 89–100
4. Gligor, M., Fournel, N., Pétrot, F.: Using binary translation in event driven simulation for fast and flexible mpsoc simulation. In: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis. CODES+ISSS '09, New York, NY, USA, ACM (2009) 71–80
5. SPEC: Cpu benchmark suites, http://www.spec.org/cpu2000 (2000)

Using QEMU in timing estimation for mobile software development

Antti P. Miettinen[1], **Vesa Hirvisalo**[2], **and Jussi Knuuttila**[2]

[1] Nokia Research Center, Finland, antti.p.miettinen@nokia.com
[2] Aalto University, Finland, firstname.lastname@tkk.fi

Abstract. We present our research on using QEMU as a platform for software development that targets mobile hand-held devices. For such devices, understanding the timing of software execution is essential for energy consumption estimation. Traditional cycle-accurate simulators are orders of magnitude slower than real hardware, and thus unsuitable for software developers. Our research shows that embedding an abstract hardware simulator within QEMU achieves simulation times and accuracy required for software development.

1 Introduction

Software development targeting mobile devices is by its nature cross-development. The use of functional simulators in such development environments is an industry standard practice. High simulation speed is crucial for the simulators used in these setups as the productivity of the software development is greatly a□ected by the speed of iteration associated with contemporary software engineering practices.

However, understanding power and performance characteristics of the software is very di□cult when the simulation is only functional. The simulation time on a development host can be a severely misleading indicator for the performance of the software on a real target device. For mobile devices, especially the energy consumption of the software is a critical metric of interest. Considering modern hardware and software, energy consumption can be estimated if the timing of the execution can be simulated [1]. Our research has addressed simulating the timing behavior of software by using QEMU as our research tool.

2 Our contribution

Our research contribution is fitting a practical timing simulator of modern multi-core hardware within a functional emulator without a significant performance loss [2]. Our modified version of QEMU, called pQEMU [3],

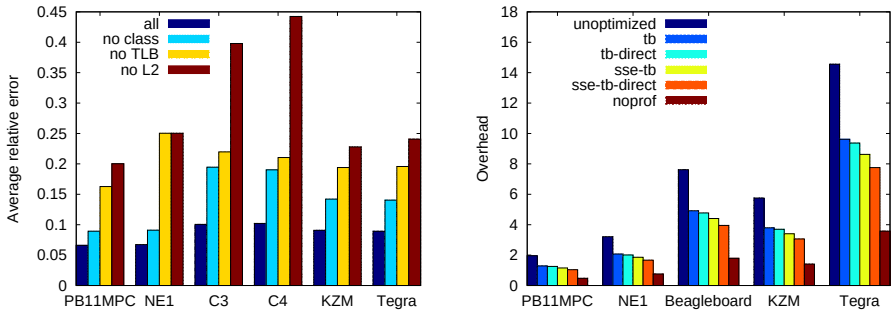
Table 1: Test platforms and simulation overhead

Platform	CPU	Speed	Cache (L1, L2)	Instrumentation				
				none	classify	L1	L2	TLB
PB11MPCore	ARM11	210MHz	32k+32k, 1M	0.3	1.3	2.1	2.1	8.5
Naviengine NE1	ARM11	400MHz	32k+32k, none	0.5	2.1	3.4	3.4	14
Beagleboard C3	Cortex-A8	500MHz	16k+16k, 256k	1.0	4.3	6.9	7.0	28
Beagleboard C4	Cortex-A8	720MHz	16k+16k, 256k	1.4	5.8	9.4	9.5	39
KZM	Cortex-A9	500MHz	32k+32k, 512k	0.9	3.8	6.1	6.1	25
Tegra	Cortex-A9	1GHz	32k+32k, 1M	2.4	9.7	15	16	63

simulates the timing of instruction executions and memory latencies. Instruction execution timings are simulated using instruction classification and weight coe□cients, while memory latency is simulated using a set-associative cache and TLB simulator.

In order to keep the overhead of the simulation as low as possible, we perform multiple optimizations. These include instrumentation at basic block granularity using direct TCG load-add-store triplets and SIMD vectorized cache simulation. We have also experimented with using multi-core parallelization for running the cache simulation in parallel, but we found it to o□er relatively modest speedups at significant implementation di□culty.

While highly accurate hardware modelling can cause a dramatic increase in simulation overhead, reasonable accuracy can still be achieved with a light-weight simulation. Our method is based on calibrating the



(a) Estimation error (b) Simulation optimizations

Fig. 1: Estimation error and simulation overhead with optimizations

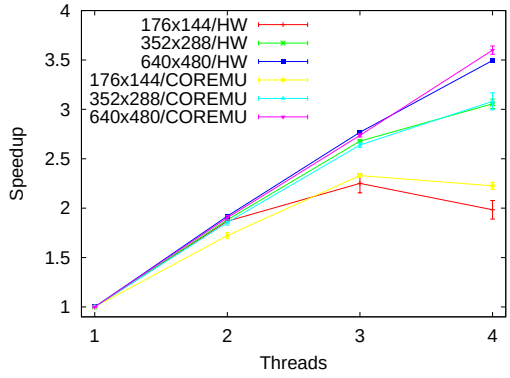


Fig. 2: Comparing performance scaling of x264 with different video frame sizes in real hardware and inside simulation.

simulator with timing measurements from real hardware. Table 1 shows our test platforms and illustrates how the simulation overhead increases in our non-optimized implementation when more details are added to the simulation. Conversely, Figure 1a illustrates how the timing estimation error is affected by the simulation detail. The effect of our optimizations on the overhead is shown in Figure 1b.

3 Related work

Traditional cycle accurate simulators are typically several orders of magnitude slower than real hardware, whereas purely functional simulators can almost reach the speed of the simulated hardware. This has led to the development of cycle-approximate instruction set simulators [4, 5].

Extending functional simulators with timed models is often motivated by the needs of hardware development [6, 7]. As our concern is software development, simulation performance is a prime concern. Instead of accurate hardware modelling, we aim towards hardware abstraction and rely on calibration for getting the accuracy to an acceptable level.

4 Future work

Current QEMU system mode emulation does not utilize host cores for simulating the target cores in parallel. As the number of processor cores is rising in cell phones, laptops, etc., we see this direction of research

and development vitally important for the future. Such a research and development e□ort is already taking place in the COREMU project [8].

An interesting topic for future research is whether execution driven simulation is able to provide meaningful performance estimates for multithreaded workloads while maintaining high simulation speed. We have done preliminary studies on the topic. Figure 2 show results of our experiment, where the performance scaling of x264 inside COREMU is compared to the scaling in real hardware. To better understand the applicability of QEMU, more experimentation is needed with true multicore emulation.

References

1. Miettinen, A., Hirvisalo, V.: Energy-efficient parallel software for mobile hand-held devices. In: Proc. USENIX Workshop on Hot Topics in Parallelism (HotPar). (2009)
2. Miettinen, A., Hirvisalo, V., Knuuttila, J.: Execution-driven simulation of non-functional properties of software. In: Proc. European Simulation and Modelling Conference (ESM). (2010)
3. Hirvisalo, V., Knuuttila, J.: Profiling for QEMU, Technical Report ESG-pQEMU-1 (2010) Aalto University, http://esg.cs.hut.fi/publications/esg-pqemu-1.pdf.
4. Cmelik, B., Keppel, D.: Shade: a fast instruction-set simulator for execution profiling. SIGMETRICS Perform. Eval. Rev. **22** (1994)
5. Franke, B.: Fast cycle-approximate instruction set simulation. In: Proc. Int. Workshop on Software & Compilers for Embedded Systems (SCOPES). (2008)
6. Monton, M., Portero, A., Moreno, M., Martinez, B., Carrabina, J.: Mixed SW/SystemC SoC Emulation Framework. In: Proc. IEEE Int. Symposium on Industrial Electronics (ISIE). (2007)
7. Gligor, M., Fournel, N., Pétrot, F.: Using binary translation in event driven simulation for fast and flexible MPSoC simulation. In: Proc. IEEE/ACM Int. Conference on Hardware/software codesign and system synthesis, New York, USA (2009)
8. Wang, Z., Liu, R., Chen, Y., Wu, X., Chen, H., Zang, B.: COREMU: A Scalable and Portable Parallel Full-system Emulator. Technical report (2010) Parallel Processing Institute, Fudan University.

QEMU - A Crucial Building Block in Digital Preservation Strategies

Dirk von Suchodoletz, Klaus Rechert, and Achille Nana Tchayep

Albert-Ludwigs University Freiburg, Hermann-Herder Str. 10, 79104 Freiburg i. B., Germany

Abstract. Emulation as long-term preservation strategy depends on usable, reliable and sustainable emulators to preserve old or obsolete original digital environments. The most common approach to achieve this is to preserve the hardware layer of the different computer architectures. The open source project QEMU is in many ways a digital preservation tool that the community of memory institutions depends on.
The requirements for emulation in preservation differ from those that exist for most virtualization tasks. QEMU has to bridge a widening gap between past and current digital environments by keeping the emulated hardware constant and reliable over time. To ensure reliability special test suites should be run on updated versions of QEMU. A number of software and unit tests could be run totally automated using the QEMU VNC and monitor interfaces on prepared system images of all preservation relevant original environments.
QEMU as a cooperative endeavor should extend its user base into the digital preservation domain. It could bring (financial) support and thus foster more sustainability.

Emulators in Digital Preservation Emulation is an extremely versatile and durable solution for retaining access to any kind of digital content. For dynamic and interactive digital objects like educational software, research applications or computer games, emulation is actually the last remaining option, as these digital objects usually cannot be migrated [9, 10] to a more current format.

When compared to the emulation of applications, emulating hardware is a much more efficient and effective approach in terms of the number of emulators required. For instance an application emulator like Scumm-VM[1] supports a small range of specific older computer games in contrast to virtually all objects for X86 systems supported by QEMU. Nevertheless, the longevity of emulation solutions in digital preservation is major concern: While X86 emulators and virtualization tools have been available for more than 10 years they have not yet proven their long-term sustainability or availability. The future of software development in the virtualization domain is thus worryingly difficult to predict.

While the virtualization and emulation community is more interested in system behavior simulation and provision of test environments for modern/current

[1] Script Creation Utility for Maniac Mansion Virtual Machine, `http://www.scummvm.org` [1/8/2011]

operating systems, the digital preservation community depends on long-term availability of old hardware in virtualization/emulation software. This hardware has to be an exact replicate and be correct and accountable in regards to its code-base stability.

Emulation does not avoid migration, but moves it to a more abstract level. As computer development advances, emulators as mostly platform dependent applications also need to be adapted. The major challenge with this state of affairs is in updating the "outer" software shell of the emulator application without changing any inner components at all. This has been accomplished very well in the case of MESS[2] for "dead" architectures like the old Apple Macintosh or home computers of the 1980s and early 1990s. In contrast to this commercially available tools such as the VMware X86 virtualization suite have mostly ignored the long-term preservation domain. Such commercially available tools have changed their emulated hardware configurations significantly. In the course of this they have deprecated old operating systems like Windows 3.X from Version 4.X on. Additionally, the container formats of the virtual hard disk have been regularly altered, rendering current *Workstation* versions unable to access containers of earlier 3.X versions [10].

QEMU as a Digital Preservation Tool Emulation in digital preservation is seldom used standalone in direct user interaction but often part of a whole workflow integrating other tasks like object type determination or verification. As many workflows imply repeated tasks executed on several hundred up to million objects of similar type they need to be able to be run unattended. There is potential for completely automated migration workflows deploying original tools and software environments to generate, for example, PDF output from Ami Pro document input. Thus preservation frameworks require predetermined and reliable ways to interface to emulators [6].

From the feature set point of view, QEMU is "digital preservation aware" as it implements the different interfaces which could be used for automated migration-by-emulation services as outlined before. Beside the traditional direct GUI in- and output a VNC interface is available which offers an appropriate abstract layer to automatically interface to display content and provide automatic keyboard and mouse input replacing the human user [7].

Another important feature in QEMU is the "monitor" interface implementing an API to emulator controls like power and reset buttons or on removable devices. It offers a channel to send commands during emulator operation for e.g. mounting removable devices, sending special keystrokes like CTRL-ALT-DEL and allows suspending and shutting down of the emulated environment.

A X86 emulator has to consider a range of additional challenges beside reproducing the deprecated hardware for actual computer architectures. As long as the number of objects to be processed is manageable, or just a few individual

[2] The Multi Emulator Super System is a source-available project which documents the hardware for a wide variety of (mostly vintage) computers, video game consoles, and calculators through software emulation, `http://www.mess.org` [1/8/2011]

users interact with emulation environments, the required computing power and wall clock time consumed for those processes is minor. If large scale preservation systems are to be run, and preservation planning tools like PLATO [1] are to be used to evaluate runtimes and give reasonable cost estimates, more information is needed [3] such as the memory consumption or CPU load.

For large scale migration scenarios requirements like predictability and accountability of actions taken play an important role. The time required for an action (calculated e.g. in wall time or CPU cycles consumed) should be predictable to give archive operators a base to calculate costs and the amount of time consumed for a particular preservation actions [3]. Additionally the monitor interface should be more usable to operate complete guest sessions by sending sequences of keystrokes or mouse events.

Preserving the Emulated Hardware Digital preservation depends on stable virtual machine hardware, because of this the emulator plays a crucial role. It bridges the widening gap between the digital past and todays current environments. As an operating system like DRDOS 6.0 or Windows 3.1 is not maintained any more, there will be no support for newer hardware like actual gigabit network adaptors or high resolution 3D graphic cards. Thus, the reproduction of a certain 20 year old software environment depends on emulators like QEMU providing exactly the hardware configuration in use in those times. In order to be able to deploy such an environment in 20 years from now the emulated hardware must remain exactly the same across time e.g. the virtual 386 ISA bus machine of the early nineties being equipped with a NE2000 network adaptor, 16 bit Soundblaster sound card and simple VGA should remain unchanged.

Until now the version history of QEMU was pretty good maintaining the once introduced hardware components but rather volatile regarding the reliable support of the older operating systems like Windows 3.X, 95 or 98. For some versions the floppy support did not work correctly, in others Windows 9X could not be started at all and in newer QEMUs the mouse does not operate as expected e.g. caused by the changeover from the Bochs to the SeaBIOS. A major issue is the *virtual hardware available driver mismatch*: At the moment the Cirrus Logic VGA is unusable in Windows 3.11 and the PIIX3 IDE is not proper recognized by the Windows 95 or official Intel drivers. Such changes and flaws are a major problem for digital preservation as the long-term reliability and stability of emulators and the hardware they emulate is crucial. As long as there are no abstract emulator-specific video or IDE drivers provided for outdated operating systems a complete reproduction of original environments depends on the proper hardware emulation.

Automated Emulator Testing Software testing is often a very elaborate task involving a lot of manual and repetitive labour. As the part of the QEMU community interested in the support of older operating systems is comparatively small we suggest other means to solve the problem. The tests should be run

mostly automatically by running a large number of different original environments and checking on certain components, like:

- Booting off different media like hard disk, floppy, optical media images and network.
- Checking on screen resolution, network availability, proper keyboard translation and mouse support.
- Booting the operating systems which are required to work properly to set up preservation workflows. Check that they reach a defined level.
- Starting applications and execute certain actions.

Currently at the University of Freiburg, a test framework for QEMU and other VNC enabled emulators has been created using the results of research into the automation of interactive sessions [7, 5]. The idea is to begin by interactively recording a particular workflow using a modified VNCplay,[3] such as the boot process of an old operating system from hard disk or installation medium. Such a recording can serve as base for the generation of a machine script for a later completely automated repetition. The recordings are comprised of an ordered list of interactive events like mouse movements or key strokes which are passed on to the emulated environment through the VNC interface. Each of these events is linked with an expected outcome which can be observed as a state of the emulated environment. Before this effect is seen, the next event cannot occur. The link with expected outcome of a previous event is necessary since certain actions will take different amounts of time to run.

The Black-box-Testing-principle [4] was selected to perform the emulator-testing because all the tests are realized without knowledge of the emulator's code or internal structure. Thus the selected emulator will be viewed as a Black-Box with only the inputs and outputs are taken into account. Through the VNC interface events (input data) are sending to the emulator and which produces one or more outputs. Like in a ordinary software test these output is compared to the expected outputs. To ensure reliability all the same tests cases are performed on every updated emulator version. In this way it is easier to check if the previous emulated systems, applications, and hardware in the new emulator version are running properly. This principle is knowing in the software engineering as regression test [4].

Thus the framework in development defines and executes usage scenarios to verify the behavior of the emulator on a predefined checklist. This list should be compiled in a way to tell as precisely as possible which component failed if a certain task does not perform as expected. It is to be sorted in such a way as to start with small tasks such as booting a simple DOS from floppy disk, ISO and hard disk before starting more complex operating systems like Windows 9X or running complex games in high screen resolutions on top of them. All results are compared to expected, pre-recorded outcomes. The emulator testing framework is programmed in Java to be included into the existing toolsets dealing e.g. with software management for the original environments. In a further step it will be

[3] See `http://suif.stanford.edu/vncplay` [1/8/2011]

extended to verify other VNC enabled X86 emulators like the newest version of Dioscuri [8, 2] with the same set of tests.

Conclusion and Future Development In many aspects QEMU seems as a optimal digital preservation tool, implementing most of the APIs required for (automated) digital preservation actions missing only more advanced monitoring features. Being Open Source and possessing a large supporting community it has a good chance of long-term sustainability in contrast to commercial solutions. Nevertheless it lacks the reliability required for durable preservation solutions yet [11]. The original hardware needs to be supplied in a way that the original operating system drivers could directly be used as long as no other options (for example, high resolution screen output) are made available. Here the different user communities should cooperate more tightly e.g. to discuss and develop automated quality assurance for well defined test sets of standard (deprecated) environments. Another important issue is the long-term code serviceability and the definition of crucial hardware components to support, e.g. for migrating deprecated machines directly to the emulator.

To support the QEMU project the digital preservation community should extend the model used to finance the Dioscuri development. A joint effort of the national memory institutions could bring in relevant contribution without over straining the resources of single partners [12].

References

1. Christoph Becker, Hannes Kulovits, Mark Guttenbrunner, Stephan Strodl, Andreas Rauber, and Hans Hofman. Systematic planning for digital preservation: evaluating potential strategies and building preservation plans. *International Journal on Digital Libraries*, 10:133–157, 2009.
2. Evgeni Genev. Vnc-interface for java x86-emulator dioscuri. October 2010.
3. Brian Hole, Li Lin, Patrick McCann, and Paul Wheatley. Life: A predictive costing tool for digital collections. In Andreas Rauber, Max Kaiser, Rebecca Guenther, and Panos Constantopoulos, editors, *7th International Conference on Preservation of Digital Objects (iPRES2010) September 19 - 24, 2010, Vienna, Austria*, volume 262, pages 359–364. Austrian Computer Society, 2010.
4. ISTQB. International softaware testing qualification board (istqb). Online, *http://www.istqb.org*, 2010.
5. Klaus Rechert, Dirk von Suchodoletz, and Randolph Welte. Emulation based services in digital preservation. In *JCDL '10: Proceedings of the 10th annual joint conference on Digital libraries*, pages 365–368, New York, NY, USA, 2010. ACM.
6. Klaus Rechert, Dirk von Suchodoletz, Randolph Welte, Felix Ruzzoli, and Isgandar Valizada. Reliable preservation of interactive environments and workflows. In Mounia Lalmas, Joemon M. Jose, Andreas Rauber, Fabrizio Sebastiani, and Ingo Frommholz, editors, *Research and Advanced Technology for Digital Libraries, 14th European Conference, ECDL 2010, Glasgow, UK, September 6-10, 2010. Proceedings*, volume 6273 of *Lecture Notes in Computer Science*, pages 494–497. Springer, 2010.

7. Klaus Rechert, Dirk von Suchodoletz, Randolph Welte, Maurice van den Dobbelsteen, Bill Roberts, Jeffrey van der Hoeven, and Jasper Schroder. Novel workflows for abstract handling of complex interaction processes in digital preservation. In *Proceedings of the Sixth International Conference on Preservation of Digital Objects (iPRES09)*, 2009.
8. Jeffrey van der Hoeven. Dioscuri: emulator for digital preservation. *D-Lib Magazine*, 13(11/12), 2007.
9. Jeffrey van der Hoeven and Dirk von Suchodoletz. Emulation: From digital artefact to remotely rendered environments. In *Proceedings of the Fifth International Conference on Preservation of Digital Objects (iPRES08)*, pages 93–98, The British Library, St. Pancras, London, 2008. The British Library.
10. Dirk von Suchodoletz. Requirements for emulation as a long-term preservation strategy. Online, *http://eprints.rclis.org/18984/*, July 2009.
11. Dirk von Suchodoletz. A Future Emulation and Automation Research Agenda. In Jean-Pierre Chanod, Milena Dobreva, Andreas Rauber, and Seamus Ross, editors, *Automation in Digital Preservation*, number 10291 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2010. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, Germany.
12. Dirk von Suchodoletz, Klaus Rechert, Jeffrey van der Hoeven, and Jasper Schroder. Seven steps for reliable emulation strategies – solved problems and open issues. In Andreas Rauber, Max Kaiser, Rebecca Guenther, and Panos Constantopoulos, editors, *7th International Conference on Preservation of Digital Objects (iPRES2010) September 19 - 24, 2010, Vienna, Austria*, volume 262, pages 373–381. Austrian Computer Society, 2010.

MARSS-x86: A QEMU-Based Micro-Architectural and Systems Simulator for x86 Multicore Processors

Avadh Patel, Furat Afram, Kanad Ghose
Department of Computer Science
State University of New York at Binghmaton

Marss-x86 (Mico ARchitectural and System Simulator for x86) is an open-source full system simulation framework developed at SUNY Binghamton for fast *cycle-accurate* simulation of single-core, multi-core and heterogeneous cores configurations. Marss is built on QEMU to support full-system simulations running unmodified OSes and applications. The simulated processor model used in Marss is derived from PTLsim [1], which simulates cycle accurate out-of-order cores. We have extended this core model to simulate heterogeneous cores in one chip, including in-order and out-of-order cores. Marss provides a new event based memory hierarchy simulation which models coherent caches supporting MESI and MOESI protocols and both bus and switched on-chip interconnections in detail. Marss implements a MMIO based interface to communicate between the VM and the simulator to allow seamless switches between emulation and simulation. The other system level components models used in Marss are derived from QEMU and include functional models of chipsets and I/O devices like network adaptors and disk drives. This allows real-time IO simulation with cycle accurate core models (homogeneous or heterogeneous) s collection framework allows users to capture detailed statistics data of multiple regions in single run including separate user level and kernel level statistics. With all these features, Marss provides a unique tool for future micro-architectural research.

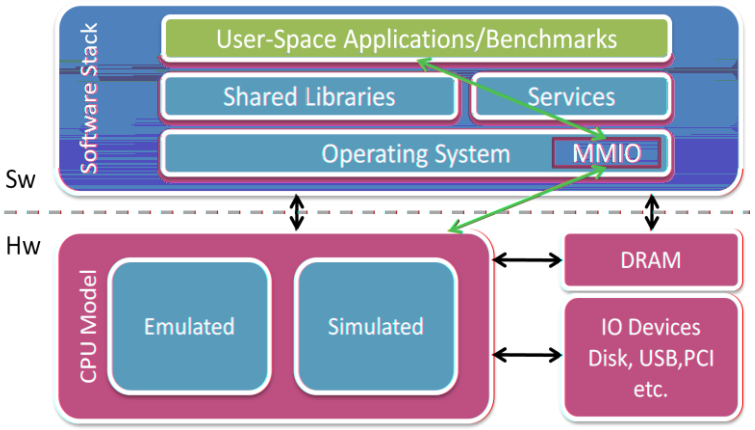


Figure 1: Overview of the Marss framework and communication between various modules

The Marss framework consists of a QEMU based simulator which integrates a significantly modified and augmented PTLsim based core simulation model and new memory hierarchy models. Figure 1 shows a high level overview of Marss framework and communication between various hardware modules and VM. As Marss is built on top of QEMU, many components of QEMU like emulated IO devices, user interface are directly reused in Marss. Marss integrated into the QEMU based CPU emulation models. The simulated models share the CPU Context with emulated models to retain the same interface between CPUs and other modules of the system. This permits a seamless

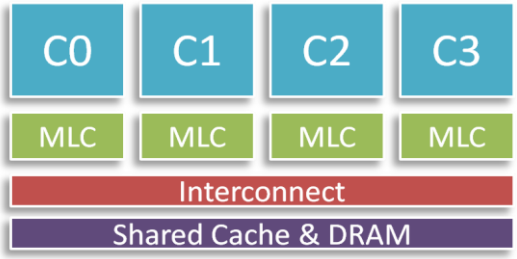


Figure 2: Overview of Simulation Models

switch between emulated and simulated CPU models. Marss also implements a MMIO device interface (Figure 1), which provides a communication channel between the simulated/emulated hardware components. Using this channel, software running in VM can send control signals or communication messages to the simulator. Figure 2 shows an overview of the CPU simulation model containing 4 cores, private L1 and L2 (MLC) caches, on-chip interconnect, shared cache and on-chip DRAM controller. We have extensively modified the core framework to allow users to simulate heterogeneous configurations, consisting of in-order and out-of-order cores on the same chip that shares cache and other resources. The heterogeneous framework also allows users to simulate different cache configurations. The out-of-order core model contains various structures like fetch queue, issue queue, re-order buffer, load-store queue, multiple register files, branch predictors etc. It also implements cycle-accurate simulation for a superscalar pipeline including register renaming, out-of-order issue, store to load forwarding, load-store aliasing, etc. A new statistics collection framework in Marss allows users to capture separate statistics of user mode and kernel mode activities in a single simulation run. Users can also set up the simulator to capture statistics of various regions of interest, for example for specific functions or loops, and collect them in a single simulation run.

One of the most challenging issues in developing Marss was to move the PTLsim based simulation model out of Xen and clean up complexities that were specific for Xen. PTLsim required users to have root level access to set up and run simulations, required the installation of a customized Xen hypervisor, thus impeding its wide deployment. PTLsim also did not support the full set of MMX instructions, coherent caches and other components needed for full system simulation. One of the biggest design requirements for Marss was to create a full system simulation framework that was complete, and easy to set up and use. QEMU was our first choice in this respect because of its ability to run unmodified OS without any hardware support, full user space emulation capabilities and fast emulation support. Additionally, QEMU also allows users to create snapshots which are used in Marss to create checkpoints of various benchmarks at user-specified locations. Marss also takes the virtual clock and the real clock by slowing down the virtual clock by the simulation speed of Marss (which is 1000x slower than a real high-end machine) and thus automatically delays various IO interrupts and timers of VM.

Marss also inherits one of the most attractive characteristic of QEMU by providing an extensible framework that permits quick integration of customized or any 3rd party simulation modules. Marss users have already integrated simulation modules like DRAMsim [2] to perform detailed DRAM simulations. There are ongoing efforts to integrate DiskSim [3] and FlashSim [4] into Marss. In general, Marss provides a unique full system simulation framework on top of QEMU for future micro-architectural and system level research focusing on multiple and heterogeneous execution cores.

References:

[1] PTLsim: x86-Cycle Accurate Processor Simulator, http://www.ptlsim.org
[2] DRAMSim, http://www.ece.umd.edu/dramsim/
[3] DiskSim, http://www.pdl.cmu.edu/DiskSim/
[4] FlashSim: A Simulator for NAND Flash-Based Solid-State Drives

Showing and Debugging Haiku with QEMU

François Revol

Laboratoire d'Informatique de Grenoble (LIG), France;
Laboratoire de Conception et d'Integration des Systemes (LCIS), France
Francois.Revol@imag.fr

Abstract. Haiku is one of those many alternative OpenSource Operating Systems under development around, mostly written on leisure time but also used as a research test bed. Like many others, it requires lots of debugging sessions, as well as a showcase platform. We will detail the various uses of QEMU for these purposes with the Haiku operating system.

1 History and Goals

Haiku is a Free Software operating system written from scratch, following the design principles of the BeOS, with binary compatibility as a target for Release 1. It focuses on desktop usage, with its own vision of the KISS principle (Keep It Smart and Simple), allowing it to stay small. Development started in 2001, and a first alpha version was released september 14th 2009. Haiku has a coherent design [13], and has been used to demonstrate several research prototypes at the University of Auckland, including a new GUI layout model [12], and automatic GUI help generation [10]. Haiku uses its own filesystem, BFS [11], to provide specific features on the desktop including very fast live queries, but also support many other filesystems.

2 Showing off Haiku

2.1 Virtual Machine Images

In addition to the CD image, the Haiku website [4] proposes several disk images for the most common virtual machines (VMware, VirtualBox, and of course QEMU). QEMU's comprehensive format support allows it to use any of those.

2.2 Live on the Internet

The Free Live OS Zoo website [2] allows people to experiment with many Operating System directly in their browser with a VNC Java Applet. It is built on top of QEMU and its internal VNC server. A similar application has been built tailored to demonstrating Haiku only [3], which adds several features, like the serial port debug output available as a `telnet:` URI as shown on Fig. 1, processor count selection, and soon audio streaming. The need for an absolute pointing device for use with VNC led to fixing the Wacom tablet emulation code in QEMU.

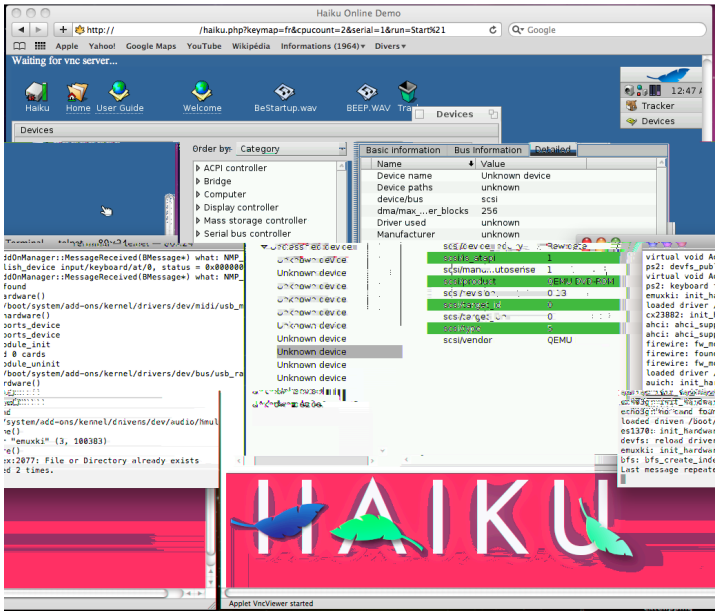


Fig. 1. The Haiku desktop running in QEMU shown in a Web browser by the VNC Applet, with serial debug output in a telnet client

3 Debugging the Kernel and Drivers

While using a traditional modular kernel, Haiku, like BeOS did a decade ago, already uses technologies being rediscovered on the desktop world, like preemptive and SMP kernel without giant lock, tickless timers, node monitoring... Development time greatly benefits from the flexibility provided by QEMU, starting from simple serial port redirection to inserting tests in the QEMU code itself to instrument the guest and assert its behaviour, the later being eased due to the Free Software nature of QEMU.

While Haiku runs very well on other virtual machines like VirtualBox or VMware, it strives at supporting as much real hardware as possible, and the diversity of devices emulated by QEMU allows better driver testing. While VirtualBox or VMware are more integrated with the host system, and their bias towards virtualization and guest additions might allow for better usage experience of the guest OS, QEMU is more suited to actual guest OS development from this perspective.

QEMU is also unique in the diversity of processor types and platforms it supports, allowing easy bootstrap of ports to PowerPC, ARM and other architectures. Haiku officially supports only the x86 architecture, but other ports are in various stages of completion, as shown in Table 1.

Haiku's Kernel Debugging Land provides many tools including a `gdb` stub, however the QEMU-provided stub has proven useful to debug the bootloader for

Table 1. Haiku ports and emulators used for development

Architecture	Platforms	Status	Emulators
x86	PC	100%	QEMU, VirtualBox, VMware
PowerPC	PowerMac, Pegasos...	25%	QEMU, PearPC [6]
ARM	Gumstix, BeagleBoard...	10%	QEMU
m68k	Atari Falcon, Amiga	10%	ARAnyM [1], QEMU-m68k [9]
MIPSel	Yeeloong[5]	2%	QEMU
X86_64	PC	2%	QEMU

the ARM port, with `gdb` reading the ELF version allowing to step through the C/C++ code. The Haiku build system setup doesn't build `gdb` along with the cross compiler, but it is a simple step to perform.

Interesting challenges facing Haiku include support for more time sources than just RDTSC, which causes problems in virtualized environments. Partial HPET support exists for timers which will require testing, first on QEMU before real hardware.

4 Debugging QEMU

OS diversity also means more hardware usage patterns, sometimes uncovering bugs in hardware chipsets, like the ATI IXP bridge not sending IRQs for their 8253/54 PIT (and APIC as well it seems) timer chip emulation when using mode 0, which BeOS and Haiku use to implement tickless timers.

Likewise, the diversity of guests systems which Haiku participates in allows for better emulation. For example a bug in the HD audio device caused an interrupt storm in the Haiku driver which wasn't triggered by other OSes drivers, because the Haiku driver handles codec status changes outside of the IRQ handler, unlike drivers in other OSes [8].

Also, when developing the `haiku.php` script, an absolute pointing device was required to keep both cursors in sync in the VNC applet, however the Wacom tablet emulation code had several issues, like wrong coordinate scaling and buttons reporting, as well as boggus asynchronous mode leading to high guest cpu usage. This simple fix [7] took several years to get through though.

5 Conclusion

Haiku provides various usage patterns to QEMU, allowing both to get fixes and thus better reliability. The availability of QEMU in most GNU/Linux distributions allows people to easily test and debug Haiku, attracting new developers. The many supported system emulations and the integrated GDB stub speed up bootstrapping ports to other architectures.

References

1. Atari running on any machine. http://aranym.org/.
2. The Free Live OS Zoo. http://www.oszoo.org/.
3. Haiku online demo php script. http://dev.haiku-os.org/browser/haiku/trunk/3rdparty/mmu_man/onlinedemo/haiku.php.
4. Haiku project. http://www.haiku-os.org/.
5. Lemote yeeloong notebook. http://www.lemote.com/en/products/Notebook/2010/0310/112.html.
6. Pearpc emulator. http://pearpc.sourceforge.net/.
7. QEMU x: Fixed wacom emulation. http://git.qemu.org/qemu.git/commit/?id=2ca2078e287174522e3a6229618947d3d285b8c0.
8. QEMU x: intel-hda: Honor WAKEEN bits. http://git.qemu.org/qemu.git/commit/?id=af93485cde810f3c2f488533e0b60c99eae5d01d.
9. Qemu m68k branch. http://gitorious.org/qemu-m68k/qemu-m68k.
10. Y. A. Chakravarthi, C. Lutteroth, and G. Weber. Aimhelp: generating help for gui applications automatically. In *CHINZ '09: Proceedings of the 10th International Conference NZ Chapter of the ACM's Special Interest Group on Human-Computer Interaction*, pages 21{28, New York, NY, USA, 2009. ACM.
11. D. Giampaolo. *Practical file system design with the BE file system*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1999.
12. J. Kim and C. Lutteroth. Multi-platform document-oriented guis. In G. Weber and P. Calder, editors, *Tenth Australasian User Interface Conference (AUIC 2009)*, volume 93 of *CRPIT*, pages 31{38, Wellington, New Zealand, 2009. ACS.
13. F. Revol. The Haiku Operating System. http://eurosys2010.sigops-france.fr/poster_demo/eurosys2010-final17.pdf.

PQEMU: A Parallel System Emulator Based on QEMU

Jiun-Hung Ding[1], Po-Chun Chang[2], Wei-Chung Hsu[2], Yeh-Ching Chung[1],

[1] Department of Computer Science, National Tsing Hua University, Hsinchu 30013, Taiwan, ROC

adjunhon@sslab.cs.nthu.edu.tw

ychung@cs.nthu.edu.tw

[2] Department of Computer Science, National Chiao Tung University, Hsinchu 30010, Taiwan, ROC

pochang0403@gmail.com

hsu@cs.nctu.edu.tw

Abstract. In this paper, we propose a novel design of a multi-threaded QEMU, called PQEMU, which can effectively deploy multiple emulated virtual CPUs on the underlying multi-core machine. The main idea of the design is to add a parallel emulation model to the execution flow of QEMU. To evaluate the design, we emulate an ARM11 MPCore by running PQEMU on a quad-core x86 i7 system and use SPLASH-2 as benchmarks. The experimental results show that the performance of PQEMU is, on average, 3.8 times faster than that of QEMU and is scalable on the quad-core i7 system for the SPLASH-2 benchmark suite.

1 Introduction

QEMU is a fast functional system emulator using the Dynamic Binary Translation (DBT) technique. With QEMU, single-core or multi-core computer systems can be emulated to run unmodified operating systems and application programs. However, the current QEMU is not designed for emulating multi-core systems. Since it runs multiple virtual CPUs in a round-robin fashion by using only one emulation thread, QEMU does not take advantage of the parallelism available in the underlying hardware. It incurs linear slowdown when the number of emulated cores is increased.

For computer architecture research, it is critical to have a full system emulator, such as QEMU, parallelized so that emulations of future multi-core architectures can actually run in parallel on the multi-core processors of current host machines. We have designed and implemented a parallel system emulator, called PQEMU, on top of the current QEMU. Although the implementation specifics are tuned for QEMU, the concepts of our work could be applied to other DBT based functional emulators.

The rest of this paper is organized as follows: Section 2 reviews the serial emulation model of QEMU. Section 3 describes the design of PQEMU. Benchmark results for the Serial QEMU and PQEMU are presented in Section 4.

2 The serial emulation model of QEMU

The current QEMU implementation has a severe performance limiter for emulating multi-core machines. Figure 1 presents a computer system emulated by the current QEMU including one emulation thread and one I/O thread. The emulation thread is composed of CPU, Memory and I/O functional models of one guest machine. No matter how many VCPUs (Virtual CPU) are created, only one emulation thread is activated to run all VCPUs in a round-robin fashion. Based on the serial emulation model, all emulated memory operations are considered exclusive and atomic. Hence the current QEMU emulates the hardware atomic operations of guest machines with locks. The I/O thread handles human interface devices (HID) and timer alarm from the host operating system. The host system events will update the status of guest I/O devices and trigger interrupts. The emulation thread can be interrupted by the I/O thread for signal delivery. Such a serial emulation model is simple and can be effectively applied to single-core emulations. However, such a model is inadequate for emulating multi-core systems.

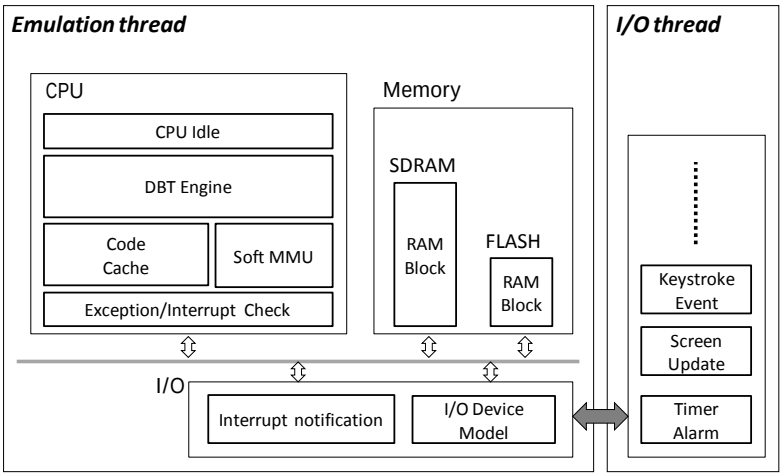


Fig. 1. A computer system emulated by QEMU

3 The design of PQEMU

To parallelize QEMU, we designed a parallel emulation model so that each emulated VCPU can be deployed on a separate core in the underlying multi-core machine for effective parallel execution. Each VCPU is 1-to-1 mapped to an emulation thread and is scheduled by the host OS independently. If the host machine has multiple cores and the number of cores is greater than or equal to the number of VCPUs, PQEMU can boost its emulation speed and more accurately emulate the characteristics of concurrent execution in multi-core systems. We will briefly explain the parallel emulation model in three dimensions: CPU, Memory and I/O.

For emulating multipleVCPUs, the main challenge of PQEMU is to parallelize the DBT engine in the current QEMU. The original DBT engine must acquire a big lock to serialize the

emulation of multiple VCPUs. PQEMU is designed to deploy fine-grained locks in the DBT engine to exploit parallelism available in the host machines and achieve high emulation efficiency. It is complex and error-prone to deploy fine-grained locks, we have carefully analyzed the global common resources used in the emulation flow in QEMU and proposed a DBT event synchronization model for two code cache designs: one is a unified code cache (UCC) design and the other is a separated code cache (SCC) design. In the UCC design, the global common resources are shared by all VCPU threads, while in the SCC design, the common resources are private to each VCPU thread. The SCC design prevents most DBT operations from the need of synchronization and thus minimizes synchronization overhead. However, the SCC design requires much more memory for private code caches. Otherwise, the UCC design has only one copy of the code cache, but requires more synchronization overhead for various DBT operations. If PQEMU needs to emulate a large number of VCPUs, the SCC design might be less desirable in terms of the memory overhead.

For emulating the shared memory for multiple VCPs, PQEMU must handle the translation of guest architecture's atomic memory instructions correctly and efficiently. PQEMU has implemented two atomic instruction groups based on the *swap* instruction and an exclusive mechanism based on *ldrex* and *strex* (which are similar to the load-link and store-conditional pairs on the MIPS ISA) defined in the ARM ISA. The *swap* instruction is similar to the exchange instruction *xchg* in the x86 ISA, except that *swap* can work on two distinct registers (one source, and one destination register). This design excludes the possibility of using *xchg* to directly emulate *swap*, unless both the destination and the source registers are identical. PQEMU realizes the *swap* instruction by an explicit spin_lock, and has implemented a global monitor to support the *ldrex* and *strex* based exclusion mechanismare.

The guest peripheral I/Os are emulated in QEMU by function callouts. A device callout is invoked once a VCPU raises an I/O request via MMIO read/write or IN/OUT instructions. A real I/O request is usually completed and uses an interrupt to inform the completion to the OS. In QEMU, an emulated I/O request is finished before the execution of the next guest instruction. Currently, PQEMU inherits this sequential I/O model from QEMU, based on an assumption that accessing guest MMIO registers by the guest OS is always exclusive, i.e. real I/O requests only appear in the kernel critical sections. Though modern kernels already follow this design convention, PQEMU still suffers when dealing withlong-latency I/O requests such as DMA operations or disk accesses. To exploit more parallelism with I/O, we suggest a dedicated host thread other than using VCPU threads to carry out these time-consuming operations. This suggestion is not to replace the original I/O model, but to complement it.

The implementation of PQEMU has been instrumental in identifying deficiencies in guest device emulation. For example, the ARM PL011 UART in PQEMU uses the same emulation callout as in QEMU, which has no Tx FIFO mechanism due to a simplified implementation. The simplified implementation works well in the original round-robin VCPU

emulation environment; but in PQEMU, Tx IRQ could overwhelm the guest interrupt handling routine if all VCPU cores are generating data simultaneously via UART. In such a case, the guest Linux will be forced to shutdown this spurious UART IRQ, and causedthe guest console to freeze.

4 Experimental Results and Conclusion

We have used PQEMU to emulate an ARM11 MPcore (with four cores) [1] on a quad-core Intel i7 based host machine. When running the parallelized SPLASH-2 [2] benchmark on the emulated MPcore, PQEMU is 3.8x, on average, faster than the original QEMU. To further understand the performance and the possible limitation to scalability of PQEMU, we have built multiple internal profiling tools to examine the synchronization overheads of the UCC and the SCC designs. For the research community, it is very useful to have such a parallelized full system emulator, such as PQEMU, so that the emulation for future multi-core architectures can run concurrently on the current multi-core computers. PQEMU opens up new opportunities to more efficiently study the issues of parallel applications and system software for future multi-core systems.

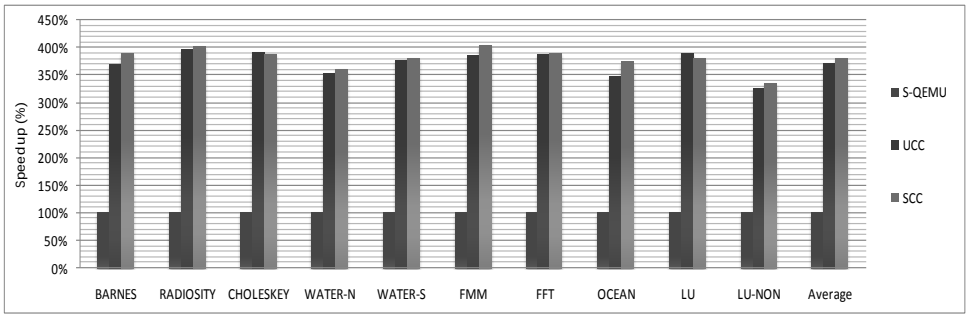


Fig. 2. Splash-2 benchmark speedup of PQEMU (UCC and SCC) compared to serial QEMU (S-QEMU)

References

1. K. Hirata and J. Goodacre. ARM MPCore; the streamlined and scalable ARM11 processor core. ASP- 748, Jan. 2007.
2. S. C. Woo, M. Ohara, E.Torrie, J.P. Singh and A. Gupta, The SPLASH-2 Characterization and Methodological Considerations, 22nd Annual Int. International Symposium on Computer Architecture, June 1995

QEmu TCG Enhancements for Speeding-up the Emulation of SIMD instructions

Luc Michel, Nicolas Fournel and Frédéric Pétrot
TIMA laboratory,
CNRS/Grenoble INP/UJF, Grenoble, France.

February 11, 2011

Abstract

This contribution presents a strategy to speed-up the simulation of processors having SIMD extensions using dynamic binary translation. The idea is simple: benefit from the SIMD instructions of the host processor that is running the simulation. The realization is unfortunately not easy, as the nature of all but the simplest SIMD instructions is very different from a manufacturer to an other. To solve this issue, we propose an approach based on a simple 3-addresses intermediate SIMD instruction set on which and from which mapping most existing instructions at translation time is easy. To still support complex instructions, we rely on the existing *helper* mechanism of QEmu. We present how this solution can be handled in QEmu's Tiny Code Generator for the ARM Neon extension running on a MMX/SSE x86, and the gains it provide.

1 Introduction

The purpose of this contribution is to propose a solution applicable to retargetable dynamic binary translation that can make use of the host computer SIMD capabilities. As the nature of the SIMD extensions of the different instruction set architectures (ISA) are quite different, what can seem to be a trivial problem ends up as a fairly complex one, at least when targeting the ability of doing a translation between many extensions.

2 Dynamic Binary Translation of SIMD Instructions

2.1 Specificity and complexity of SIMD instruction sets

SIMD instructions perform parallel operations on multiple data (Single Instruction Multiple Data – SIMD). We can find today multiple ISA extensions providing SIMD instructions to general purpose CPUs. Among them, we specifically looked at MAX instructions for the PA-RISC, MIPS' MDMX and DSPASE, PPC's AltiVec, SPARC's VIS, Intel's MMX/SSE and ARM NEON. All these instructions set extensions share

the same characteristics, described below. For performing parallel operations on multiple data, SIMD instruction performs the operation (or sequence of instructions) on registers interpreted as array of values. This array of data can have a variable number of values of various size, for example a 128 bits wide register can be viewed as two 64 bits, four 32bits, eight 16 bits or sixteen 8 bits values. On top of that the variety of the operations applied to the data is huge. It ranges from the classical arithmetic operation (add, sub, shift, ...) to saturated or rounding arithmetic. Among this large range of instructions, each SIMD instruction set represents a unique subset choice made by the designers. On top of these classical instructions, SIMD extensions can even integrate some exotic instructions such as the `vmul.p8` Neon instruction performing polynomial multiplication which has no equivalent in other SIMD ISA.

We have then to make a careful choice of the micro-operations to add to the intermediate representation of the DBT. The main two constraints we respect for this extension are:

- limit the number of new IR micro-operations in order to limit the burden on the code generator and the overall performances of the binary translator.
- add enough micro-operations in the IR to allow a maximum coverage of the SIMD instruction sets.

Indeed, adding too much micro-operations will tend to add one micro-operation per SIMD instruction. This will not solve the problem since the code generator (the second phase of DBT) will have an heavy work to do to guaranty the emulation of each of the micro-operations. However if we do not add enough micro-operations all SIMD instructions cannot be expressed and the translator will have a huge task to guaranty this translation. We concentrate then on an extension of the IR with a set of instructions which is close to the intersections of the studied SIMD instruction sets. The IR micro-operations will be 3-addresses operations since it is the most general case and allows to represent the 2-addresses versions easily, whereas the reverse is not true.

As opposed to scalar DBT, finding instruction equivalence in SIMD DBT has to take care the SIMD specificities: parallelism and register interpretation.

3 Realization

The proposed IR extension has been ported in the QEMU binary translator. The translator for the ARM Neon instruction set and the code generator for the Intel MMX/SSE instruction set have been implemented. Experiments on synthetic benchmarks show that a gain up to $4\times$ can be obtained compared to the use of helpers (that are however much simpler to implement).

PRoot: a Step Forward for QEMU User-Mode

Cédric Vincent and Yves Janin

STMicroelectronics,
Compilation Expertise Center,
12 Rue Jules Horowitz, Grenoble, FRANCE
firstname.lastname@st.com

Abstract. This paper introduces PRoot, a new tool specifically designed to extend QEMU user-mode. We detail our motivations, compare with other similar solutions and give first results.

1 Extending QEMU User-Mode

QEMU [1] is an open-source hardware emulator with two main usages: in *system-mode* a whole target machine is emulated to run a full operating system. It can benefit from virtualization support like KVM to avoid the emulation of most CPU operations. By contrast, QEMU *user-mode* only emulates the CPU to run one target process at a time, and communications with the host kernel are converted by a thin layer.

Our initial motivation was to use QEMU to ease the development of embedded Linux applications by emulating their build system and test-suites directly on developers' workstations. During our experiments we found out that the system-mode was not fast enough and we decided to focus on the user-mode. Unfortunately this latter cannot execute a tree of target processes within a target Linux distribution for reasons we detail in the next section. To get the best of these two modes we developed PRoot, a tool fully compatible with QEMU that can be seen as an *extension* of the user-mode.

2 Overcoming QEMU user-mode limitations with PRoot

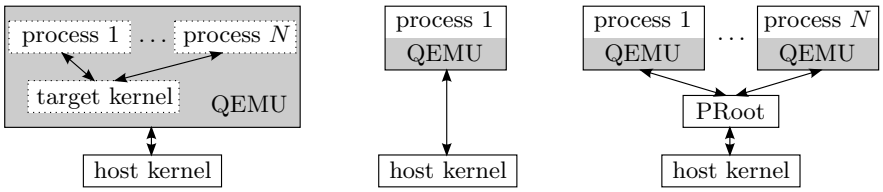
As illustrated by Figure 1 *(a)* and *(b)*, QEMU should run faster for a given process in user-mode than in system-mode since it does not emulate the peripherals nor the target kernel. However the boundary between the target environment and the host becomes *thinner*, and the host kernel now has to support some of the operations that were devoted to the target kernel. We can define three requirements that will guide us throughout the rest of this paper:

– *R1*: an efficient and correct path translation mechanism between the target and the host. This feature is typically required to emulate programs that perform absolute accesses to architecture specific files such as the dynamic linker `/lib/ld-linux.so`. Note that a correct translation scheme must support relative paths and symbolic links too.

– *R2*: the ability to spawn a new process and keep emulation running for this new process.
– *R3*: the ability to run a host process program. Strictly speaking *R3* is not mandatory, but using *un-emulated* cross-compilers and interpreters is particularly useful to speed-up build process and test-suites.

QEMU user-mode status for *R1* and *R2* is currently not satisfying. For *R1*, QEMU user-mode has the ability to redirect file operations made by the target process but it can take several minutes at each process creation to scan the Linux target distribution. Moreover there is no support for relative paths and symbolic links.

Let us now consider *R2*. From the point-of-view of the host kernel, an emulated process image $P1$ is actually an image of QEMU $Q1_{P1}$. When $Q1_{P1}$ executes a new program, the resultant process $P2$ is totally out of the control of QEMU. As a conclusion *R2* is not supported either.



(a) system-mode (b) user-mode (c) user-mode + PRoot

Fig. 1: Target process[es] running under QEMU, arrows are system calls flow

The ambition of PRoot is to fulfill *R1*, *R2* and later on *R3*, while keeping design, implementation and use as simple as possible. Since PRoot is also used for validation purpose, it has to reproduce kernel results and errors as if the emulated applications were run in a native environment.

PRoot is based on `ptrace`, a Linux system call [2] that is mainly used by debuggers or system call tracers such as *strace* and the GNU Debugger *gdb*. However since it provides unique facility to monitor and control processes, it can also be applied to a much wider range of applications bringing kernel-like features in user-space [3] [4]. With `ptrace`, PRoot is able to catch every system call emitted by a QEMUlated process and dynamically redirect them as shown in Figure 2. One of the main task in PRoot is the path canonicalization that translates paths from the target world to the host world and vice versa (*R1*). Additionally, when PRoot detects a new process about to be created, it can change the initial request and get a new instance of QEMU running the new process (*R2*) as shown in Figure 1 *(c)*.

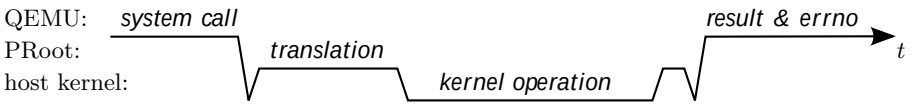


Fig. 2: Control flow of a syscall done by a target process under QEMU+PRoot

3 Results

Experiments presented in this section were performed on a Pentium-4 2.4GHz workstation with 1GiB of RAM running Slackware v13.1. Concerning the target system, we used ARMedSlack v12.2 with QEMU v0.12.5 configured to provide 256MiB of RAM when used in system-mode.

3.1 Functional Comparison With Similar Tools

Table 1 exemplifies two Linux packages that are representative of the complexity of an embedded Linux distribution; Perl 5 requires a target environment for its bootstrap whereas the Coreutils test-suite exhibits numerous limit configurations that can help to find unexpected behaviors or bugs.

Many path translation tools already exist but few pass these tests (*R1*). Fakechroot v2.14 and Fakeroot-Ng v0.17 fail. Plash is a modified GNU C library and was therefore not considered, and we did not consider `chroot` either since it requires administration privileges. Generally speaking, all these tools were not designed to support the execution of processes through an emulator (*R2*).

Only PRoot and ScratchBox 2 [5] proved complete enough to configure, build and validate the two complex Linux packages. Table 1 gives the results of their test-suites run with PRoot and ScratchBox 2, it also gives system-mode results as a reference.

Table 1: Results of the packages' test-suites

Package	ScratchBox v2.2	PRoot v0.5	system-mode
Perl v5.10.0	99.6%	99.6%	99.8%
GNU Coreutils v6.12	94.9%	97.3%	96.7%

ScratchBox 2 and PRoot have similar path canonicalization algorithms but differ in their design[1] and usage. Unlike ScratchBox 2, PRoot does not need any configuration and does not assume anything about the target and host environments. By contrast, ScratchBox 2 is a highly flexible tool with scripting facility. A complete technical comparison between ScratchBox 2 and PRoot is beyond the scope of this paper, but we can safely conclude that both are functional

[1] ScratchBox2 relies on the dynamic linker to override system call wrappers in the C library.

enough to be already used in an industrial environment. Choosing one instead another one depends on your expectations: simplicity with PRoot *vs* powerful customization with ScratchBox 2.

3.2 Performance Comparison With QEMU System-Mode

Table 2 compares performance of QEMU v0.12.5, first in user-mode with PRoot v0.5 and then in system-mode. Timings were measured with the command `time` and we checked that the system-mode results were coherent with the host clock.

Table 2: Performance of PRoot+QEMU user-mode *vs* QEMU system-mode

	Perl v5.10.0	Coreutils v6.12
archive extraction	3.6× faster	2.7× faster
configuration	2.0× faster	4.0× faster
build	2.9× faster	3.5× faster
validation	4.1× faster	3.6× faster

The speed-up with QEMU user-mode is quite impressive and comes from the fact that the target kernel and the hardware are not emulated. We think that all these results will be further improved when PRoot handles *un-emulated* cross-compilers and interpreters (*R3*). This feature is still under development and looks promising.

4 Conclusion

The simple fact that PRoot requires no privilege or setup is a great advantage that should ease the usage of this *extended* user-mode. It is worth mentioning that PRoot should work with any version of QEMU user-mode since no patch is needed, and in any Linux environment as well. Finally, the current version of PRoot is mature enough to be already used in an industrial environment.

References

1. Bellard, F.: QEMU, a Fast and Portable Dynamic Translator. In: USENIX Annual Technical Conference, pp. 41–46, FREENIX Track (2005)
2. Kerrisk, M.: The Linux Programming Interface. pp. 23, 43–46, 394, 545, 608, 801. No Starch Press (2010)
3. Spillane, R. P., Wright, C. P., Sivathanu, G., Zadok, E.: Rapid File System Development Using `ptrace`. In: Proceedings of the 2007 Workshop on Experimental Computer Science (ExpCS '07), 22. ACM (2007)
4. Dike, J.: User-mode Linux. In: Proceedings of the 5th Annual Linux Showcase & Conference - Volume 5 (ALS'01). USENIX Association (2001)
5. ScratchBox2, `http://www.freedesktop.org/wiki/Software/sbox2`

SysML based Codesign with Code Generation for QEMU-SystemC Cosimulation

D. He, F. Mischkalla, W. Mueller
Paderborn University/C-LAB
Germany

1 Introduction

Due to the increasing complexity of electronic systems UML profiles like SysML are getting very attractive for system documentation and verification aspects. However, there is still a big gap from UML models to executables for verification. In [1], we developed an efficient codesign approach based on SysML by covering the entire design flow from comodeling to cosimulation via retargetable code generation. For comodeling we defined additional profiles on top of SysML, which addresses two parts: (i) target SW integration and (ii) system architecture modeling.

This paper focuses on the code generation for the automatic configuration of HW/SW cosimulation via QEMU and SystemC. We implemented the code generation scheme in ARTiSAN Studio® (an integrated development suite for model engineering) to generate C/C++ code for target SW binaries, synthesizable SystemC for platform components and other utilities such as makefiles and scripts for design flow automation. We apply QEMU in full system mode as a processor emulator running the target SW binaries under Linux as operating system.

2 SysML based Codesign Methodology

Figure 1 shows our overall code generation scheme for the automatic configuration of a QEMU-SystemC cosimulation environment. System engineers use SysML frontend editors like Artisan Studio to specify and model the entire system. By means of the additional stereotypes described in [1] they are able to give more system engineering and SystemC specific information to the model, which is necessary for automated code generation to transform the SysML model to final executables. Thus, a SysML model consists of three different kinds of components: (i) software executables, (ii) processors, (iii) interconnect and hardware components.

For each <<C Executable>> component referencing external C source code, the code generator generates hardware address parameter information, user-space driver functions and makefiles to compile the software to a target SW binary. A target platform based on a dedicated ISA (Instruction Set Architecture) and an OS (Operating System) image is specified by the processor that is allocated from these binaries. Finally, generated scripts compile this bundle to a QEMU executable.

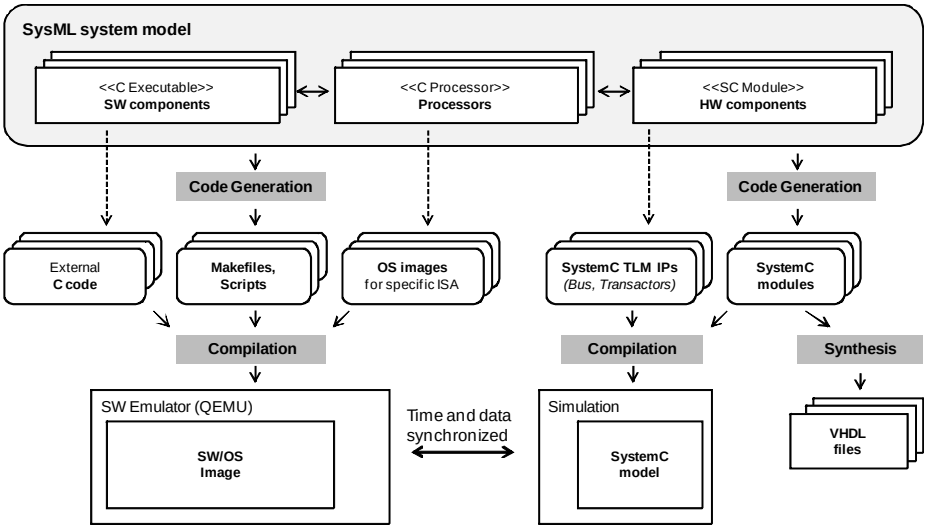


Figure 1: SysML based code generation for an automatically configured QEMU-SystemC cosimulation environment.

A <<C Processor>> component communicates over so-called interconnects with other system components. Such interconnects and other HW components are either predefined in a SystemC IP library or self-modeled as SystemC modules. Predefined components are linked into the generated SystemC code. The user-modeled SystemC components are translated to SystemC source code. Subsequently generated makefiles also run the compilation steps to build the entire SystemC executable.

After code generation, cosimulation is started by one simple button click. Due to the support of multiple target architectures by QEMU and our profiles, system engineers can easily choose between them in the SysML model. Currently, the PowerPC 405 and ARM926EJ-S based architecture with Linux are supported.

4 Conclusion

In this paper, we introduced a HW/SW system design methodology to close the gap between SysML based comodeling and QEMU-SystemC cosimulation via automatic code generation. The code generation includes C/C++ for target SW, synthesizable SystemC for HW components and other utilities for design flow automation.

References

1. Mischkalla, F., He, D., Mueller, W.: Closing the gap between UML-based modeling, simulation and synthesis of combined HW/SW systems. DATE 2009, Dresden