QBox: an industrial solution for virtual platform simulation using QEMU and SystemC TLM-2.0

Guillaume Delbergue, Mark Burton, Frederic Konrad, Bertrand Le Gal, Christophe Jego

QBox: an industrial solution for virtual platform simulation using QEMU and SystemC TLM-2.0

Guillaume Delbergue
GreenSocs -
Bordeaux INP, CNRS IMS, UMR 5218
guillaume.delbergue@greensocs.com

Mark Burton and Frederic Konrad
GreenSocs
mark.burton@greensocs.com
fred.konrad@greensocs.com

Bertrand Le Gal and Christophe Jego
Bordeaux INP, CNRS IMS, UMR 5218
bertrand.legal@ims-bordeaux.fr
christophe.jego@ims-bordeaux.fr

Abstract

As industrial demands grows for modeling, simulation and exploration tools during SoC design, there is a need for simulated CPU models that work with the other tools in the space design. There have been many attempts to build libraries of CPU design models, most notably the Open Virtual Platform project [1]. This work focuses on defining a complete open source SystemC compliant approach with a QEMU based model, which supports almost all current and past CPUs. QEMU [2] has many advantages not least a very large and dynamic open source community with hundreds of developers active all over the world. Our work makes QEMU available in a standard SystemC [3] (IEEE 1666) based tool environment. This article addresses the current state and upcoming features of two different integration techniques that allow the QEMU virtualizer and emulator to be used in a SystemC simulation context : QEMU-SC [4] and QBox (QEMU in a Box). This article also examines how these current implementation work. The limitations they have with respect to SystemC are also given. It will go on to look at recent developments both within QEMU itself and in the integration between QEMU and a SystemC that are aimed at improving simulation performance and usability of these solutions.

I. Introduction

The current embedded market is increasingly taking advantage of standard *Components Off The Shelf* (COTS), while adding value in software and complex Systems on Chip (SoC) IP. Within that context, driven by cost reductions and time to market, virtual platforms are essentials (to enable software teams) to become productive earlier in the design cycle. It also provides a common link environment between hardware, integration, verification and software development. In essence, virtual platforms reproduce system behavior, execution of target software, debug and development in the absence of "real" hardware platform. The virtual platforms can and should be used as a means for exploration between hardware and software engineers during development cycles.

The SystemC language allows hardware descriptions to be constructed in a *C++* based language. However, as the complexity of the IPs increases, the SystemC simulation environment is not necessarily suitable to provide suitably fast models. It is theoretically possible to simulate complex IP's such as CPU's within SystemC simulation kernel. But as we can see in SoCLib[5], performance can be an issue, especially when the processor is modelled at RTL level that is computationally intensive. A better solution for complex IPs like CPUs is to model it in a virtualizer or emulator and then to integrate the model into a SystemC simulation environment. Moreover, the TLM-2.0 (Transaction-Level Modeling) standard, which is an extension of SystemC, improves interoperability between memory mapped bus models. It also includes the notion of time quantum which was explicitly intended to assist with this sort of integration.

One such external virtualizer is named QEMU (Quick EMUlator). It is a generic and open source machine emulator and virtualizer which allows engineers and developers to execute their software binaries (operating systems and applications) made for one machine (processor and its peripherals) on another one. For example, the execution of ARM or PowerPC binaries on a x86 processor based computer. QEMU enables simulation of multiple kinds of processor cores and is based on a JIT (Just In Time compiler) code generation technology. JIT technology enables code recompilation / translation at run time. It can achieve emulation of the simulated processor core at near real time speed.

To interface QEMU (the CPU virtualizer or emulator) with a SystemC simulation environment containing SystemC models, multiple solutions have been proposed such as QEMU-SC, an open source solution. QEMU-SC aims to embed SystemC models in QEMU through a wrapper (as detailed in Section II-A). However, this purposed solution has drawbacks. The authors in [6] implement a solution to interface QEMU and SystemC simulation environments by separating SystemC and QEMU on two threads. The communication is made through a shared memory and a FIFO mechanism. However, the document doesn't detail time synchronization aspects between QEMU and SystemC simulation kernel based time. The solution in [7] integrates SystemC with both QEMU and OVP using a SystemC bridge. The QEMU/OVP simulations run on a own thread and the bridge is a set of C functions included in a library which is statically linked to the SystemC runtime library. The time synchronization between QEMU/OVP and SystemC is performed on a call to a SystemC model. It enables the update of SystemC time to be synchronized with the QEMU time. This solution breaks dynamic quantum (See Section III-A). A similar implementation [8]

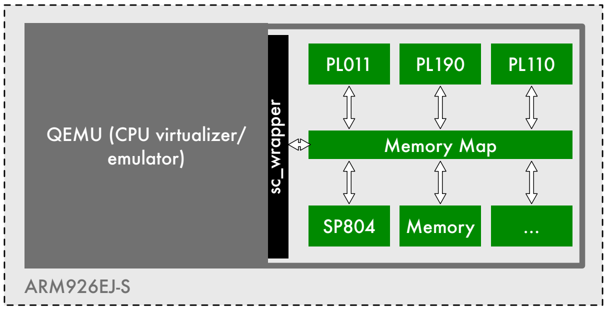


Fig. 1: ARM926EJ-S architecture with QEMU-SC

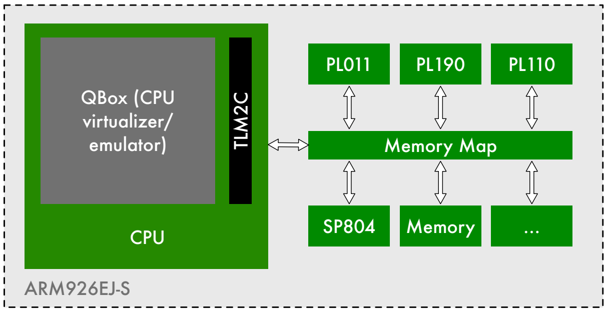


Fig. 2: ARM926EJ-S architecture with QBox

to QEMU-SC adds the capability to estimate the performance of a target system from system perspective. It is also able to trace all the information that are necessary for the estimation. In [9], BSD sockets enable communication between two virtual CPUs (two instances of QEMU) and SystemC models. A thread controller manages transactions from different cores to SystemC models. To synchronize the time, the authors send the simulation time in their transactions. However, there are no details about the algorithm used to synchronize all bases of time.

In all cases, SystemC is used to simulate complex SoCs including digital peripherals and one or multiple processor cores. The rest of the article is organized as follows. Section II briefly introduces and compares QEMU-SC and QBox solutions by presenting typical use cases. Then, detailed features, advantages and limitations of QBox for industrial virtual platforms are discussed. Finally, different solutions have been benchmarked and compared from virtual platforms to real hardware.

II. QEMU-SC simulator and QBox CPU virtualizer/emulator

A. *QEMU-SC simulator*

Combining QEMU's ability to efficiency simulate processors with the flexibility of SystemC meets the needs of much of the embedded software industry, to provide a model environment based on standards with high performance. This environment captures the power of QEMUs CPU simulation environment along with the standard approach for writing models. QEMU-SC [10], based on works of the authors in [11], provides a solution to connect SystemC simulation environment with QEMU. QEMU-SC is an open source hardware and software emulation solution for SoC development, specifically targeted development of add-on cards or peripherals that have to be connected to an existing (off the shelf) platform.

In this context, QEMU is the "master" during simulation execution. It contains the off the shelf platform. It instantiates and controls the SystemC part. This approach enables models written in SystemC to be used from within QEMU. Models can be integrated using specific memory mapped addresses such that QEMU communicates through MMIO (Memory-Mapped I/O) or PCI interfaces. In this context, SystemC is used as peripheral models to a QEMU based software platform.

As show in Figure 1, QEMU-SC provides a QEMU hardware model named `sc_wrapper` to communicate between QEMU and a SystemC model. This wrapper handles reads and writes to the memory map by building TLM transactions and redirecting them to the SystemC side. Transactions are cached, but nonetheless, this incurs a pointer and several integer copies. While in general performance is an issue for TLM systems, this overhead is minimal as these devices are not typically accessed frequently. QEMU-SC also provides a wrapper for interrupts in both directions. `sc_platform` registers SystemC models within QEMU and binds them to TLM router and IRQ vector. Depending on the simulation requirements, two different implementations are available:

- MMIO which is a generic implementation in TLM-2.0 standard,
- PCI which is a PCIExpress bus implementation in TLM-2.0 standard.

The current implementation of QEMU-SC has some limitations in that IRQs are not handled as cleanly as they could be. It means that SystemC models need to have a 'handle' to the QEMU model, and pass that back to QEMU when they are sending an IRQ. This breaks the TLM standard and means that some alterations are required to a SystemC model to be used with QEMU-SC.

B. *QBox CPU virtualizer/emulator*

QBox is an integration of QEMU virtualizer and emulator in a SystemC model. Contrary to the QEMU-SC solution, QBox or QEMU in a (SystemC) Box, treats QEMU as a standard SystemC module within a larger SystemC simulation context. SystemC simulation kernel remains the "master" of the simulation, while QEMU has to fulfill the SystemC API requirements. This solution is an open source QEMU implementation wrapped in a set of SystemC TLM-2.0 interfaces. QBox allows the powerful JIT based

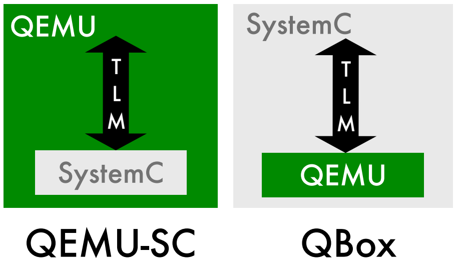


Fig. 3: QEMU-SC (QEMU is master, SystemC simulation kernel is slave) vs QBox (SystemC simulation kernel is master, QEMU is slave)

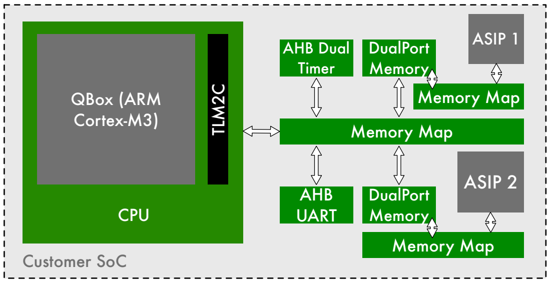


Fig. 4: Example of a virtual triple-core platform

CPU simulations to be totally exploited within a TLM-2.0 context. QBox is provided as shared libraries that contain QEMU based CPUs as shown in Figure 2. Each QBox library contains a specific CPU core and exports a set of TLM-2.0 like 'C' interfaces. A QBox library is instanced in a SystemC simulation context through the SystemC wrapper called TLM2C. TLM2C library provides the C++ TLM-2.0 standard interfaces. It exports TLM-2.0 'C' like interface to the standard TLM-2.0 C++ interface.

This solution integrates well into a virtual platform to facilitate the co-design of hardware models but also to validate the software which runs onto the platform. QBox is sufficiently flexible to support some basic system components from the QEMU library that have been included within the QBox component. So, only the models of interest can be specified in SystemC language. It means that QBox is a flexible platform simulation environment. Thus, novel components can be added and device drivers can be designed. QBox can be used within the context of a much wider system as mealy one of many components.

Figure 2 provides an overview of an existing virtual platform using the SoC ARM926EJ-S. It is the SoC used in the ARM Versatile PB board. Depending of the host machine, QBox emulates or virtualizes the core part of the SoC (an ARM9E-S) and connects its AMBA bus as a SystemC TLM socket, thanks to the TLM-2.0 standard generic protocol. As QEMU is written in C (as opposed to SystemC which is standard C++ class library), a wrapper called TLM2C is required to connect them. Each of the remaining Versatile PB models are modelled in SystemC language and connected to the bus using TLM-2.0 standard. To specify the bus routing, GreenRouter has been used. It is a part of the open source library GreenLib [12]. GreenRouter makes use of optional extensions in the TLM-2.0 protocol that are provided by GreenSocket. It allows routable addresses to be allocated in devices, rather than in the router. This is merely a convenience and ease the system description.

The platform described in the Figure 2 boots with a Linux kernel. It is possible to interact with Linux through the UART (PL011 model) and also view the output of the graphical interface PL110, which is part of the SoC. All parts of the Figure 2 are available and open source[13].

C. *Use cases*

QBox and QEMU-SC both have their uses. QEMU-SC is useful in cases where cards or a small number of components are under development. It is also useful when components have to be added to existing off the shelf platforms. QBox is more suitable when the CPU element is considered to be one component of a wider system. The main difference between QBox and QEMU-SC is that:

- For QBox CPU virtualizer/emulator, SystemC simulation kernel is the "master" of the simulation, ie QBox, which embeds QEMU, acts as a slave, a SystemC model. SystemC simulation kernel controls the QEMU execution. SystemC models can be interfaced without modifications.
- For QEMU-SC simulator, QEMU is the "master", ie QEMU pause and resume SystemC simulation kernel if necessary, SystemC models are slave. SystemC models have to be adapted to the QEMU interface.

The Figure 3 illustrates differences.

Critically, both QBox and QEMU-SC solutions enable execution of software binaries (operating systems and application stacks) without any modification to real target code. In both cases, QEMU directly executes the code from memory. For QEMU-SC, the memory is held within QEMU. For QBox CPU virtualizer/emulator, the memory is held within SystemC and accessible via a TLM-2.0 interface. QBox makes use of the TLM-2.0 Direct Memory Interface (DMI) to access memory. Thus, the execution speed of a binary in QBox, which doesn't use excessive IO (as an access to a register through the memory map), is typically indistinguishable from native QEMU. QBox's performance is typically extremely fast:

- for non natives guests (e.g. ARM on x86 processor), it can be within an order of the host machine. It is typically “real time” for most embedded processors,
- for native guests (e.g. x86 on x86 processor) the KVM (Kernel Virtual Machine) can be used. This provides simulation speed close to host performance.

However, there exists run-time penalties when the software running in QEMU accesses IO. Currently, the time to perform an IO as a simple write to a register from Linux is about 40ms (host time) on an host machine with an Intel Xeon E3-1271. For example, Linux kernel executed in QEMU performs a write to a register modelled in a SystemC model like one of the registers in PL011 peripheral). Looping on IO constantly suffers from these penalties. However, this is typically not the behaviour of most systems. Actually, most systems do exhibit this behaviour, typically they are spinning waiting on IO, so the delay isn't important either. Nonetheless, this is an area where we intend to do further research. We expect to decrease this number to few nanoseconds if the IO is thread safe. This issue is addressed in more details below (See Section III-B).

QBox is designed to fulfill today's requirements for virtual platforms. As QBox provides a TLM-2.0 component instance, it is ideally suited to support the increasing demand for platforms that integrate multiple homogeneous and heterogeneous processors. As QBox is a TLM-2.0 component, it is clearly possible to mix heterogeneous CPUs within a platform to achieve, for example, Asymetric Multi-Processing (AMP). This is a feature that is missing in QEMU (and QEMU-SC). These sorts of AMP systems are extremely typical in everything such as smartphones, Internet of Things (IOT) or smart devices. A typical system would contain an 'off the shelf' virtualized with QBox and one or more ASIPs. All of this executes within SystemC simulation. However QEMU does support homogeneous cores working in a Symmetric Multiprocessing (SMP) manner. This is also possible within QBox. Hence both homogeneous (SMP) and heterogeneous (AMP) systems can be modeled and simulated by QBox.

Currently, QBox has one limitation for multi cores support. All cores of an SMP CPU need to run in the same instance of QBox. QBox doesn't export into SystemC the inter-CPU communication system necessary to enable SMP to be simulated within SystemC. This would be an interesting arraignment if more complex interconnect structures have to be investigated. For instance, when some of the CPUs have access to certain peripherals. This is probably a minority of use cases right now. But nonetheless we would like to enable it.

1) Example with a triple-core platform: QBox has been used with success in the definition of a virtual platform containing an ARM Cortex-M3 with two ASIP models for a future IOT chip. This virtual platform is composed of standard Cortex-M devices modelled with SystemC like AHB UART but also some custom IP models as described in Figure 4. The communication channel between the two subsystems (ASIP and Cortex-M3) in this design is typical of many such designs revolving around the use of interrupts and shared memory and this aspect has been modelled. However, care must be taken in a quantum based system that both sides are reactive enough to those interrupts to enable the smoothly communication. As it is typically the case with quantum based TLM-2.0 models, some degrees of quantum 'tuning' are required to balance performance and communication accuracy.

This virtual platform use multiple threads to speed up simulation execution. It makes a better usage of resources available on the simulation host. The ARM Cortex-M3 is embedded in a QBox instance. QBox runs in 2 threads (one for IO and one for CPU, as QEMU). Another thread is used by SystemC itself. We recall that SystemC is the master of the simulation. As the ASIP model is provided by a third party it is outside our control.

This virtual platform has been developed in parallel with the real SoC. The platform has already shown benefit in terms of hardware and software teams that agree on their (complex) interface between the ARM and ASIP systems. They are now able to develop their softwares ahead of the hardware tape-out. This is a classic example of the Electronic System level (ESL) working, with a direct return on investment for a small device, based exclusively on open source infrastructure.

It should also be noted that, there is no agreement on how signals outside the memory mapped bus are handled. The third party devices in this case used a typical implementation of interrupt signals. Unfortunately, it does not play especially nicely in a quantum based TLM-2.0 simulation context. Of course, this only required wrapping of those interfaces. But nonetheless this is annoyance and confusion that nobody needs. We hope to address this issue in a future work.

III. Features of QBox

A. Time synchronization

QBox works within a SystemC simulation. QEMU's notion of time is typically based on the guest clock, also called `vm_clock` in QEMU. On the other hand, SystemC is purely event driven and its clock moves as fast as events can be processed. Due to different time domains, it is necessary to ensure time synchronization. To further complicate matters, QBox and SystemC run in a separate threads to improve efficiency and simulation speed. QBox takes advantage of the quantum mechanism built into TLM-2.0. This enables a model to be at most one quantum ahead of SystemC's current time. QBox's time increases in parallel to SystemC simulation time in a different host thread. Quantum level synchronization is maintained between threads by ensuring that Equation (1) is always respected.

$$|time(QBox) - time(SystemC)| \leq Quantum \qquad (1)$$

Time synchronization is one of the bottlenecks of a virtual platform. Indeed, due to the static length of a quantum, it is necessary to add checkpoint quantums even if there are no events pending in SystemC. In some systems, to handle tightly coupled IO, the quantum has to be reduced. But, this has a negative impact on performance because it increases simulation time. It can also be wasteful if IO is not always used. To ensure this, when QBox or SystemC executions are at a quantum boundary, they use a thread *wait* to synchronize. It enables to wait for the other parties to finish their quantum.

Currently, QBox only supports static quantums. However, in order to avoid redundant and unnecessary checkpoints, we speculate that a dynamic quantum mechanism may be beneficial. At this point, this is left as a future work. A straight forward dynamic quantum approach, in which a 'synchronization' - or quantum - would be placed into both SystemC and QEMU. At each point, their potential interaction (but no un-necessary points) would likely have a positive effect on a single CPU system. Our concern is that on a multi-core system, IO events on one core would cause un-necessary synchronization points on other cores. Therefore, our conclusion is that a 'simplistic' approach is not going to have universally positive results. Therefore we conclude that this is a subject for much more detailed research.

B. *Performing IO*

QBox accesses memory by using standard TLM-2.0 transactions. When the guest software running in QBox wants to perform an IO to a SystemC model, it stops its execution. Then, one of the following two cases will occur:

- If SystemC is running, a thread safe event is posted to the SystemC simulation kernel. Then the simulation scheduler has to run the TLM-2.0 IO request
- If SystemC is sleeping (as it has already finished its quantum), SystemC will be woken up. The same thread safe event is posted.

Once the processing of the SystemC event is completed, QBox is notified. Then it continues its execution with the completed transaction. The Figure 5 illustrates both cases. Transactions from SystemC to QBox are used for interruptions generated from devices (SystemC models). QEMU is able to receive these interrupts in a thread safe way. Hence, no special care has to be taken when they are generated by devices and sent by using standard TLM-2.0 based sockets.

When QBox wants to access an IO managed by a SystemC model, it does so by issuing a thread safe event which is executed by the SystemC simulator, within the SystemC thread. This guarantees that only the SystemC thread is used to execute SystemC code. This solution also maintains the single threaded nature of SystemC. However, this adds complexity and decreases performance of simulation when different simulators have to be synchronized. It may be possible to use some of the constructs in work on parallel SystemC [14]. But this falls outside of the existing SystemC standard.

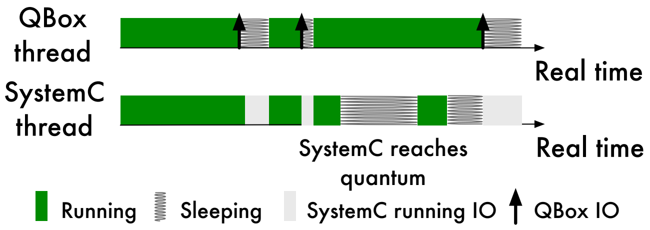


Fig. 5: QBox synchronization illustration

In general, it is possible to write thread safe SystemC models. The models own data structures that can be protected in the normal way. As the SystemC language is not guaranteed to be thread safe, models that require accesses to the kernel need to use an asynchronous notification. If we could guarantee that the models connected to QBox were thread safe, it is not necessary to switch threads. In this case, QBox would be able to directly access IO without the synchronization mechanisms. To add this feature we would like to add a new way to inform QBox that this part of SystemC simulation is thread safe. It means that it can be directly accessed without interruption. We do this while complying with the TLM-2.0 standard. This will speed up simulation and improve usage of parallel threads.

C. *From mono-thread to multithread*

Currently, QBox (and QEMU) run within two threads : one exclusively for IO operations and another one for CPU execution. The CPU thread runs the TCG (Tiny Code Generator) which performs the translation from CPU guest code to CPU host code. This is the JIT engine (or dynamic binary translator), which is the core of QEMU. Initially, QEMU was proposed to emulate one CPU on a host with a single CPU. When multiple CPUs are emulated, a "round robin" approach is used within the single CPU thread. However, with the proliferation of cores in both the PC hosts and the platforms being modelled, this approach is not optimal.

Previous works like [15] forked QEMU to run each virtual CPU on its own thread. To do that, it was necessary to parallelize the TCG [16] part of QEMU. The TCG performs two steps. On one hand, it transforms any supported guest CPU target instructions to TCG operations. On the other hand, it transforms TCG operations into CPU host instructions. This intermediate step between guest and host translation decreases CPU dependencies by adding a new level of indirection. It makes the TCG flexible and maintainable across a large number of hosts and guests. Unfortunately, parallelization is difficult to implement. The authors limited themselves to only one small case. Therefore, this approach is not safe for all cases.

In all cases, there are a number of issues that have to be addressed. One of the most obvious is the issue of 'atomic instructions'. For a single threaded model, the model can assume an ordered memory model. It can implement atomic (compare and exchange, load/store exclusive etc - used to synchronize multiple cores) with no special restrictions. However, this is not the case for multi-threaded models.

The authors of COREMU[17] emulate multiple cores by creating multiple instances of existing sequential emulators. A thin library layer is used to handle the inter-core, the device communication and the synchronization. The objective is to maintain a consistent view of system resources. However, they did no more than adding a mutex around the load and store exclusive instructions for ARM. Unfortunately, this is not sufficient. Indeed, it does not fulfill the ARM requirements, and fails to support anything but the smallest of guest code sequences. The authors of HQEMU[17] combines LLVM with TCG to speed up translation. This solution not only speed up single threaded applications that runs in the guest but also multi-threaded applications.

D. MTTCG project

The MTTCG project (a QEMU project) aims to allocate one host thread to each simulated CPU to significantly improve performance. It also enables the power of the host machine within the heart of QEMU itself. This project also aims to upstream changes so that the techniques are re-donated to the community such that others can build upon it. For this first iteration, MTTCG focused on the ARM architecture. But it can also be extended to all targets within QEMU. The initial limitation is only due to a single patch serie that addresses atomic instructions within the ARM architecture. Other architectures are being addressed in parallel.

1) Global TCG State: In the current implementation of QEMU, there is no protection against two threads attempting to generate code at the same time, placing the results into the translation buffer (a part of TCG). In a multi-threaded system, it leads to corrupted code generation from time to time. In order to better handle the translation cache, the key question is whether the translated code cache should be per-guest-CPU (per-thread) or global (shared between multiple guest CPUs). Per-guest-CPU means less possibility of locking contention. But it means more overhead generating code. Every time the guest reschedules a process to another guest CPU, we'll have to translate all its code all over again for the new CPU. A strictly global cache is not a great idea either. Indeed, it won't work if we eventually moved to supporting heterogeneous systems (for example, one ARM CPU and one SH4). One possibility is a hybrid system. Each guest CPU have a pointer to its TCG cache (which could then be shared between several other CPUs).

Sharing a cache would allow us to take advantage of code that is translated by one core and then used by another. On the other hand with one cache per core, updates on the caches with a lot less locking can be performed. Each CPU could generate translations simultaneously for its individual cache. However, invalidates can be done across all the caches if any core writes to program memory. This would be expensive as all CPU caches would have to evict TBs, rather than a single 'central' entry being evicted.

Interestingly, the structures existing in QEMU are similar to a tiered cache system in hardware. While each CPU holds a local list of pointers to translated code, there is also a wider 'level 2 cache' shared by all CPUs. The local 'caches' are simple indirections to the main cache, so 'evictions' can happen centrally. This provides the benefits of tiered caching. It permits to implement a highly optimal solution. In its current form, only one CPU can generate a TB entry. Then, it is posted both locally and to the global TCG cache. This already provides a highly powerful solution. It would be possible to enable more than one CPU to generate different TB entries concurrently.

2) Dirty tracking: Currently, QEMU handles guest writes to memory by using a set of bitmaps for tracking dirty memory of various kinds setting the internal QEMU TLB entries up to force subsequent reads (e.g. for code execution) to consult the real memory (this is termed the slow-path).

This is a fairly long sequence of operations (guest write; read bitmaps; update TB cache structures; update bitmaps) which is currently effectively atomic because of the single thread being used. In order to enable multiple threads, this chain has to be dealt with carefully. Specifically, when a CPU marks an area as 'dirty' in the bitmaps, all CPU's need to see this message. QEMU works by assuring that the effect of the JIT TB cache can not be seen. Therefore, dirty information has to be transmitted and acted upon 'immediately'. In order to achieve this, a new service was necessary to be added to QEMU to allow a core to request that all CPU's stop. For instance, a TB entry was flushed atomically from the point of view of all simulated cores. This new mechanism is introduced in our work.

3) Memory consistency: Host and guests might implement different memory consistency models. For instance, the ARM memory model does not guarantee that other cores will necessarily see writes in order (or at all), until a memory barrier is executed. An x86 based machine has a stronger memory model. It normally guarantee that all threads see all writes. While supporting a weak ordering model (eg. ARM) on a strong ordering backend (e.g. x86) isn't a problem. For now, the only realistic solution is to place memory barriers after writes. The performance impact of this very much varies in terms of exact architecture and implementation. We do not present benchmarks here. Indeed, they are being worked on by other members of the QEMU community. As our work initially focused on ARM running on x86, we were able to ignore this issue for the moment.

4) Instruction atomicity: Atomic instructions allow guest code to run on several guest CPU's for the synchronization. They are the critical means by which e.g. SMP is achieved. They rely on a few basic instructions being 'atomic' from the perspective of all guest cores. One example is a store exclusive instruction in ARM where the store should succeed if no other thread is active (either read or written) to a memory location since a preceding load exclusive operation. One obvious implementation of these instructions is to check that the value remains the same between the load and store exclusive operations. Unfortunately, it is not architecturally correct, since the architecture specifies that no load or store should have occurred. Potentially a non exclusive load or store could have occurred. It may not effect the value in the memory. A subsequent store exclusive would then erroneously succeed.

Experimentally, we have replaced the store exclusive code that was written in this manner by host atomic instructions (e.g. using the `cmpxchg` primitive). We have shown it seems to be stable across many boots and application cycles. However, theoretically even using host atomic instructions is not necessarily architecturally correct in all cases. A more 'complete' solution would be to mark memory locations in the dirty bitmaps and force access to them to be taken through the safe-work mechanism described above. While this is conceptually much cleaner, it may have performance impacts on simulation speed. At this time, other ones in the QEMU community are building such an implementation based on our mechanisms to test out the potential speed reduction impact. Thus, host atomic instructions has not caused any issues in terms of atomic locks not working correctly. Test cases to show how locks could be broken have been attempted, but we have yet to produce the effect. Nonetheless, it makes sense to model the architecture more accurately if the performance penalty is acceptable.

E. Scientific contributions

Our approach is to provide usable virtual platforms for multi-core systems. Such systems have to to be simulated across a number of host threads. We remain at the "CPU block" level by mapping virtual cores onto physical host cores. This is the major difference with other approaches that focus on SystemC threads. It is driven by synchronizing the number of CPU cores.

During the investigations, we have identified and provided solutions for 2 JIT engines. Specifically a new approach proposed coordinate cache-coherency work within a JIT, between cores. We term this the safe-work mechanism. It allows cores to request work to be done locally, on specific remote cores, or globally. The mechanism itself is robust. We have identified when and how it should be deployed within the JIT for specific cache operations. One good finding is that the safe work mechanism, and the points during simulation when it should be deployed can be implemented in a architectural neutral way.

To solve the issue of the shared instruction cache, we discovered that the way in which the JIT cache structure operates actually mimics the effects of a real hardware cache. Because of this, we strongly favoured the approach of sharing a sort of level 2 cache between CPUs (much as is common in real hardware). Implementing this yields extremely impressive numbers (see Section IV).

Finally for atomic instruction handling, we re-used the hosts ability to perform atomic instructions, effectively translating guest instructions directly to host atomics. In our case, we restricted ourselves to the compare and swap (`cmpxchg`) atomics. This is clearly guest dependent. For our case, we focused on ARM on X86. However, we concluded that this solution has the drawback that some guest instruction semantics are not exactly implemented. Our solution is stable for ARM on X86, but is unlikely to solve all guests. Indeed even for ARM we were able to hypothesize a test case that would fail, though we have not been able to produce the true effect in reality. Nonetheless, we consider this as a negative result.

Overall, the result is that the speed improvement we get for multithread TCG is impressive. Our implementation is stable over Linux boot and all loads that have so far been run. We have not been able to show any issues from the atomic instruction implementation. For more details, see Section IV.

To sum up MTTCG and QBox, we have researched both approaches to multithreading simulation. Our conclusions so far are that multithreading QEMU itself is ideally suited to homogeneous (SMP) cores, while multiple instances of QEMU (QBox) are suitable for heterogeneous (AMP) cores.

F. Limitations

The SystemC scheduler is not preemptive, ie all processes run in cooperative mode. Each process handing control back the the kernel, under its own control. The SystemC scheduler will never interrupt a running process. It is necessary to introduce synchronization points to pass control back to the kernel. When there are no more processes to execute, then SystemC exits. In addition, the SystemC kernel is not thread safe. Since the scheduler is not preemptive, it is not really a requirement. This means that models written in SystemC do not have to consider thread safety issues. It simplifies the model and speeds up the development process. However from our point of view, this is problematic as we run several different threads for different CPUs. However, a thread safe mechanism has been added to SystemC to enable interaction between SystemC and other OS threads and processes. It is a specific thread safe event. It is the mechanism applied when QBox wants to post an event to SystemC simulation kernel.

If QBox wants to access an IO within a SystemC model, it can use a thread safe event which will then be executed by SystemC. It guarantees that the single SystemC thread is used to execute SystemC code. This solution also maintains the single threaded nature of SystemC. It adds complexity due to lock system between threads and decreases performance of simulation. In general, it is possible to write thread safe SystemC models. If the models connected to QBox were thread safe, it is not necessary to switch threads. In this case, QBox would be able to directly access IO without the synchronization mechanisms. To add this feature, we would like to add a new way to inform QBox that this part of SystemC simulation is thread safe and can be directly accessed without interruption. It has to be compliant with the TLM-2.0 standard. This will speed up simulation and improve usage of parallel threads.

G. *Open source license and business potential*

QBox, as a fork of QEMU, is a free and open source project released under GPLv2 license (see QEMU header files). QBox and the QBox libraries are also released under the GPLv2 license. TLM2C is our wrapper that can take any 'C model' based on structures like TLM-2.0 standard. It converts it into SystemC (C++) code. This is released on the GPLv2 with the additional right to use the code with SystemC. TLM2C is mealy a library that enables the construction of a SystemC model from a C based library, which could be QEMU or any other C based model.

Model constructors in an industrial setting can choose to use any TLM-2.0 standard compliant CPU models that suits their needs. We believe that QBox is an exceptionally good choice because of its quality, speed, license and features. When industrial model constructors ship their models to third parties, they do not have generally the rights to ship CPU models coming from other vendors. This is equally the case with our models. The customers must be able to select whichever CPU model suits their needs. This was the key motivating driver behind the TLM-2.0 standard. Again, whatever CPU was chosen during the initial construction of the platform, we believe that QBox is a good choice as a CPU TLM-2.0 model.

In addition to the quality and speed of a QEMU based solution, the open source license also guarantees the longevity of the solution. Indeed, as users are able to keep, modify and re-use the solution for as long as required. In the safety critical industry, this is a key advantage of open source as solutions need to be re-usable for decades into the future. Indeed, users can develop themselves of have any competent developer maintain and develop models based on this solution. This can encourage services based businesses, geographically placed in regions specializing in specific domains.

IV. Experimental results

A. *Introduction*

Experimentations are done with QBox, QEMU-SC, and ARM Versatile Express board containing a CoreTile Express A9 daughter board in addition to the motherboard. The daughter board contains four A9 cores. It is a good candidate to compare results for a multi-core SMP platform. QEMU officially supports this board.

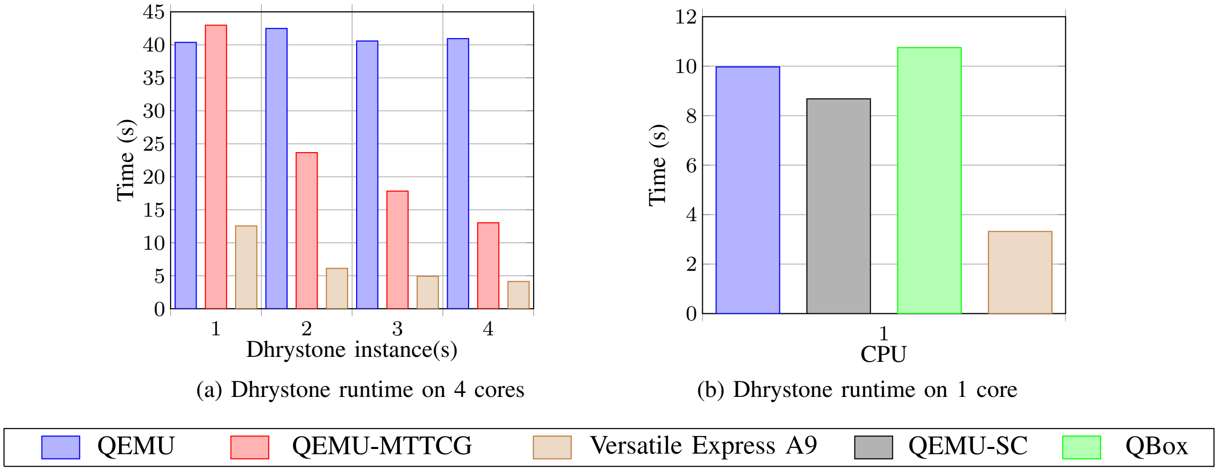
All benchmarks (see Section IV-B) have been executed on real hardware in addition to QEMU, QEMU with MTTCG, QEMU-SC and QBox on a computer running an Intel Xeon E3-1271 (at 3.6 GHz) and 32Gb of memory. We have modified the DTB (Device Binary Tree) to use only those devices that are available to all virtual and physical platforms. They are: the 4 x Cortex A9, System Controller, Generic Interrupt Controller (GIC), SP804 (Dual Timer) and PL011 (UART).

The objective of QEMU-SC is to model the majority of the platform inside QEMU by adding externally only a few elements. Specifically QEMU-SC expects that the memory will be held inside QEMU. In contract, the philosophy behind QBox is that the memory map is handled in SystemC. To keep the comparison as fair as possible, we set up the two environments as follows: QEMU-SC will run all Cortex A9 cores, System Controller, the full memory map and ARM GIC within QEMU. Both the SP804 and PL011 will be in the SystemC. QBox will run an A9 multi-core node with all 4 Cortex A9 processors, and the tightly coupled ARM GIC. The current version of QBox (1.3.0) is based on QEMU 2.3.0. ARM GIC is shared by the multi-core A9's on a private bus. The full memory map, the SP804 and PL011 will be specified in SystemC. After some experimentations, as we explain below, we have decided to fix a quantum value of 1ms for QBox and QEMU-SC.

B. *Benchmarks*

In order to provide a measure of performance focused on raw processing power, we have chose to use Dhrystone[18]. Indeed, it is readily available on all platforms. Dhrystone is an open source synthetic computing benchmark program used to measure the integer performance of processors. We used Dhrystone 2.1 that is built with the Buildroot toolchain. The program has been added to root file system to be run in the guest OS (Linux). 10^7 Dhrystone computations were benchmarked. As the time reported by a virtual machine may not be accurate, an external timer was used to measure execution time, averaging over 5 runs.

One of Dhrystone deficiencies in terms of benchmarking is a total absence of IO. In order to compare a more 'real world' scenario, we have also compared Linux boot performance which does use a mixture of IO's and processing. We built Linux kernel and a light root file system for our platform by using Buildroot 2015-08.01 with Linux kernel v4.1.4. We only enabled the required devices to boot the platform and run the benchmarks.



(a) Dhrystone runtime on 4 cores

(b) Dhrystone runtime on 1 core

Fig. 6: Dhrystone runtime on QEMU, QEMU-MTTCG, and Versatile Express A9 with different number of CPUs

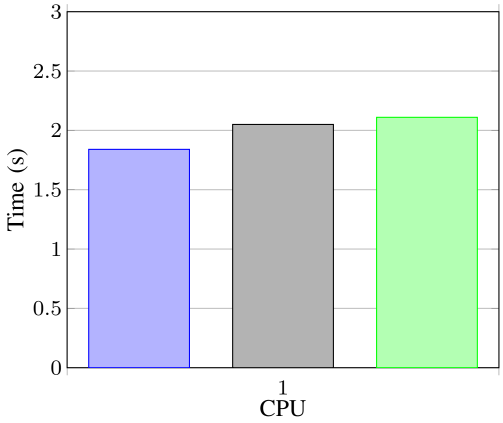


Fig. 7: Different solutions booting Linux

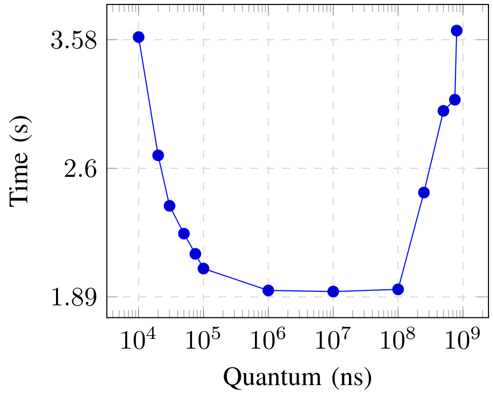


Fig. 8: Quantum impact on Linux boot using QBox

C. *Dhrystone*

As we can see on Figure 6, we benchmarked 4 Dhrystone's that are executed at the same time with different numbers of CPUs using real hardware, QEMU and the new QEMU-MTTCG. We suitably modified the DTB to limit the number of CPU's being used. The current QEMU does not take advantage of increasing the number of CPUs. Indeed, the time to compute the four Dhrystone is approximately the same for all configurations. However both QEMU-MTTCG and the real hardware are both capable of dividing the load over multiple CPUs. For of a single CPU, QEMU-MTTCG is slightly slower than standard QEMU. This is the overhead of ensuing thread safety in QEMU, measured as about 14%. For the case of a 4 cores CPU, QEMU-MTTCG is over 3 times faster than the existing QEMU. Overall, MultiThreaded TCG (MTTCG) has now been demonstrated with an impressive near linear speed improvement. We plan to implement MTTCG in QBox in a future work to speed up SMP CPUs simulations. Finally, we can see that QEMU emulator is around 3 to 4 times slower than real board on our host machine. Real board cores run at 1.3GHz so the host machine executes one virtual CPU instruction all around 10 host instructions.

On the Figure 6b, we can see that overall QEMU is only 3-4 times slower than a board running at 1.3GHz. This is clearly highly dependent upon the host and guest. However, this falls well within the often assumed 'one order of magnitude', even for quite complex cores such as the A9. The various SystemC solutions (QEMU-SC and QBox) are both substantially similar to QEMU in terms of CPU computation performance. QEMU-SC runs very slightly faster than QEMU in this test, this is due to the version of QEMU used in QEMU-SC. The older version of QEMU used in QEMU-SC (close to version 1.4.50) runs Dhrystone about 20% faster than the version of QEMU used in QBox (and for the other QEMU tests), which is version 2.3.0. Applying a ratio between QEMU 2.3.0 and QEMU 1.4.50, we estimate that QEMU-SC overall has a performance cost of around 3% compared to the QEMU being used. QBox is slightly slower than QEMU-SC, specifically for IO accesses as discussed above. QBox has a performance cost of 8% compared to the QEMU being used.

D. Linux

During boot step, Linux performs a few thousands IO. As shown in Figure 7, we can see that boot time of all the solutions are very close to the time set by QEMU itself. QEMU-SC and QBox are a little bit slower than the original QEMU. Even though Linux boot performs significant amounts of IO, the boot times of all three options are substantially similar. We do not see a marked slow-down in any of the solutions. Clearly the way in which Linux boots, and how it times IO activity during the boot has a significant impact on these numbers. However, we argue that far from being atypical, Linux boot is one of the typical things that the users of virtual platforms will run, time and time again (especially if they are in the process of developing low level software, the very target of virtual platforms of this type). Hence the fact that all three solutions have a Linux boot time which is substantially similar is highly relevant.

E. Quantum

As shown in Figure 8, we can see the impact of the quantum duration on Linux boot by using the QBox virtual platform. We get the smallest boot time with a quantum around 1ms. For smaller quantum durations, QBox will process IO more frequently. It increases execution time and decreases simulation speed. However, as quantum durations increase, Linux will see fewer and fewer timer interrupts, which it uses to synchronize and schedule processes. As a result, processes which are spinning waiting for others, or waiting for a timer interrupt will potentially spin for longer, slowing the overall simulation to the point at the far right hand side of the graph where Linux is no longer capable of running. As the selection of an appropriate quantum is hard, it takes experimentation, and is system dependent. Typically, a 'bath' shaped graph can be expected, users may choose a lower quantum if higher timing fidelity is required.

V. Conclusion

This article provides a review of the existing solutions to interfacing QEMU and SystemC. We go on to look more closely at two solutions: QEMU-SC and QBox. Both solutions has been compared to get an objective point of view on the best solution. Some use cases have been studied. They show that QBox is more in line with current and future requirements by supporting homogeneous and heterogeneous simulations. Different approaches are proposed to simulate performance in SystemC and specifically multi-core simulations. Multiple solutions in the article are proposed for SMP and AMP platforms. For SMP, a new multithread solution is presented for QEMU called MTTCG. MTTCG results are good. Indeed, a linear speedup is achieved. All the works detailed here are available as open source on the GreenSocs.com web site. We actively seek feedback for our research activity.

References

[1] OVP. Open virtual platform. [Online]. Available: http://www.ovpworld.org

[2] QEMU. Qemu. [Online]. Available: http://www.qemu.org

[3] "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, Jan 2012.

[4] GreenSocs. Qemu-sc. [Online]. Available: http://git.greensocs.com/qemu/qemu-sc

[5] Lip6. Soclib. [Online]. Available: http://www.soclib.fr

[6] T.-C. Yeh and M.-C. Chiang, "On the interface between QEMU and SystemC for hardware modeling," in *Next-Generation Electronics (ISNE), 2010 International Symposium on*, Nov 2010, pp. 73–76.

[7] F. Cucchetto, A. Lonardi, and G. Pravadelli, "A common architecture for co-simulation of systemc models in qemu and ovp virtual platforms," in *Very Large Scale Integration (VLSI-SoC), 2014 22nd International Conference on*, Oct 2014, pp. 1–6.

[8] M.-C. Chiang, T.-C. Yeh, and G.-F. Tseng, "A qemu and systemc-based cycle-accurate iss for performance estimation on soc development," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, pp. 593–606, April 2011.

[9] C.-S. Peng, L.-C. Chang, C.-H. Kuo, and B.-D. Liu, "Dual-core virtual platform with qemu and systemc," in *Next-Generation Electronics (ISNE), 2010 International Symposium on*, Nov 2010, pp. 69–72.

[10] M. Monton, J. Engblom, and M. Burton, "Checkpointing for virtual platforms and systemc-tlm," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 21, no. 1, pp. 133–141, Jan 2013.

[11] M. Monton, A. Portero, M. Moreno, B. Martinez, and J. Carrabina, "Mixed SW/SystemC SoC Emulation Framework," in *Industrial Electronics, 2007. ISIE 2007. IEEE International Symposium on*, June 2007, pp. 2338–2341.

[12] GreenSocs. Greenlib. [Online]. Available: http://git.greensocs.com/greenlib/greenlib

[13] ——. Greensocs forge. [Online]. Available: http://git.greensocs.com/explore

[14] R. D. A. D. D. K. G. Liu, T. Schmidt, in *NASCUG Meeting, 6/2/2014.*

[15] J.-H. Ding, P.-C. Chang, W.-C. Hsu, and Y.-C. Chung, "Pqemu: A parallel system emulator based on qemu," in *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, Dec 2011, pp. 276–283.

[16] D.-Y. Hong, J.-J. Wu, P.-C. Yew, W.-C. Hsu, C.-C. Hsu, P. Liu, C.-M. Wang, and Y.-C. Chung, "Efficient and retargetable dynamic binary translation on multicores," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 3, pp. 622–632, March 2014.

[17] Z. Wang, R. Liu, Y. Chen, X. Wu, H. Chen, W. Zhang, and B. Zang, "COREMU: a scalable and portable parallel full-system emulator," in *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, 2011, pp. 213–222. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941583

[18] ARM. Dhrystone benchmarking for arm cortex processors. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.dai0273a/DAI0273A_dhrystone_benchmarking.pdf