Fast Virtual Prototyping for Embedded Computing Systems Design and Exploration

Amir Charif, Gabriel Busnot, Rania Mameesh, Tanguy Sassolas, Nicolas Ventroux

Fast Virtual Prototyping for Embedded Computing Systems Design and Exploration

Amir Charif, Gabriel Busnot, Rania Mameesh, Tanguy Sassolas and Nicolas Ventroux
Computing and Design Environment Laboratory
CEA, LIST
Gif-sur-Yvette CEDEX, France
firstname.surname@cea.fr

ABSTRACT

Virtual Prototyping has been widely adopted as a cost-effective solution for early hardware and software co-validation. However, as systems grow in complexity and scale, both the time required to get to a correct virtual prototype, and the time required to run real software on it can quickly become unmanageable. This paper introduces a feature-rich integrated virtual prototyping solution, designed to meet industrial needs not only in terms of performance, but also in terms of ease, rapidity and automation of modelling and exploration. It introduces novel methods to leverage the QEMU dynamic binary translator and the abstraction levels offered by SystemC/TLM 2.0 to provide the best possible trade-offs between accuracy and performance at all steps of the design. The solution also ships with a dynamic platform composition infrastructure that makes it possible to model and explore a myriad of architectures using a compact high-level description. Results obtained simulating a RISC-V SMP architecture running the PARSEC benchmark suite reveal that simulation speed can range from 30 MIPS in accurate simulation mode to 220 MIPS in fast functional validation mode.

1 INTRODUCTION

To keep up with a fast-evolving and highly competitive embedded system industry, designers are compelled to deliver complete working solutions under tight delay and budget constraints. To make this possible, hardware architectures, as well as the full software stacks that drive them should be validated and optimized as early as possible in the design process. In this context, Virtual Prototyping has been widely adopted as a cost-effective solution for early hardware and software co-validation.

The rapid adoption of virtual prototyping solutions was greatly facilitated by the emergence of the SystemC/TLM 2.0 standard [1], which, in addition to offering interoperability and reusability of SystemC models, provides several abstraction levels to cope with varying needs in accuracy and speed. More recently, in response to an increasing demand for simulation speed, the use Dynamic Binary Translation (DBT) for CPU modelling has gained in relevance [17], [20], [10], and has effectively set a new standard for simulation performance in early prototypes.

However, as systems grow in complexity and scale, it is now more vital than ever that virtual prototyping solutions be able to wisely exploit these technologies to offer proper balance between simulation representativeness and execution speed throughout the design process. Moreover, at this level of complexity, the time required to model and explore new architectures can quickly become unmanageable, especially at the earliest stages, where it is often necessary to make heavy alterations before reaching a stable prototype.

To cope with these new difficulties, virtual prototyping solutions need to meet new requirements not only in terms of performance, but also in terms of ease, rapidity and automation of system modelling and design space exploration. This paper introduces VPSim, a feature-rich integrated virtual prototyping solution that addresses the aforementioned challenges based on three major contributions:

- A new way of integrating the QEMU emulator, making its rich CPU and peripheral model portfolio available as a collection of SystemC modules. Our method can leverage the technologies used in QEMU, such as Dynamic Binary Translation (DBT) and paravirtualization to bring outstanding simulation speeds into the more deterministic and standardized SystemC domain.
- An Accuracy Control framework, that can be used to define regions of interest in both the software and the simulated hardware dynamically. A system designer can therefore enable accurate simulation only on parts of the design and portions of code that are of interest, guaranteeing the best possible trade-off between performance and accuracy given specific needs.
- A generic and powerful infrastructure for dynamic platform composition and design space exploration (DSE). It allows a single compiled executable to be used to model an infinity of architectures. Combined with a Python scripting front-end, it makes it possible to simulate, explore and optimize highly complex systems in a single compact script.

RAPIDO '19, January 21–23, 2019, Valencia, Spain

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/3300189.3300192

The remainder of this paper is organized as follows: In Section 2, we review some related tools and methods. Section 3 provides a complete high-level view of the features and capabilities of the VPSim tool. In Section 4, details on the underlying implementation challenges are presented. The performance of our tool is evaluated in Section 5, before concluding in Section 6.

2 RELATED WORKS

The first generation of Virtual Prorotyping (VP) solutions used functional Instruction Set Simulators (ISSs) with more or less internal low-level details such as pipeline stages. Among such solutions, GEM5 [7, 16] is a discrete-event simulator able to dynamically switch between different abstraction levels. Detailed CPU models with full pipelining description can be simulated at 0.1 Million Instructions Per Second (MIPS), whereas instruction-accurate CPU models can reach 1 MIPS. Other MPSoC modeling environments, such as SESAM [24] or Unisim [3] used instruction-based ISS generation librairies to support the modeling of various CPUs, reaching approximately 10 MIPS. However, this accuracy and ISA flexibility come at the cost of limited simulation speed, which hampers their capacity to address complex systems embedding several CPUs and running full-fledged OSes.

To address this complexity, a recent trend is the use of Dynamic Binary Translation (DBT). DBT consists in dynamically translating instructions of the modeled ISA (guest instructions) to host ones (usually x86), whenever needed during guest code execution, yielding very high execution speeds.

Today, all high-performance simulation environments use DBT. Industrial solutions are led by solutions like Virtualizer [25], Vista [19], VPS [18], VLAB [26], FastModels [20] or, for instance, Simics [22]. However, these come at a significant cost and do not offer the degree of customization that is required when designing new architectures. In addition, they do not provide fast design space exploration capabilities.

Amongst open-source virtual platforms, OVP [17] uses processor models simulated through a closed-source DBT engine named OVPSim reaching hundreds of MIPS. An OVP model can be wrapped for inclusion in a SystemC model using TLM 2.0 interfaces. However, OVPSim cannot provide performance evaluation as the models are timed relying on instruction count and neglecting memory access timings. In addition, OVP claims to support parallel multiprocessor simulation but this is at the cost of deterministic execution loss.

QEMU [4] is an open source DBT based emulator supporting many CPU models but does not provide performance estimation. Several works have tried to embed QEMU within a SystemC simulation environment to provide both determinism and timing evaluation [15]. In [15], the authors wrap QEMU processors within SystemC threads and investigate several QEMU instrumentation options to take into account instruction count and data access latency. Depending on whether synchronization shall occur on every data access or on a periodic basis, the simulation speed varies from 3 to 60 MIPS. However, the cumbersome annotation is to be performed for every ISA and possible accuracy settings are limited to a few predefined configurations. GreenSocs [10, 12, 13] propose a QEMU-based framework that exploits QEMU MMIO callback mechanisms to access external SystemC peripherals. This allows for very fast simulation for applications with few IO communications, as the execution mostly takes place in the context of QEMU. Unfortunately, this solution runs QEMU and SystemC in separate kernel threads, requiring frequent synchronization and precluding determinism.

The solution we propose integrates QEMU by executing its CPU and peripheral models in the context of SystemC threads, thereby preserving the predictability of Single-Threaded SystemC simulation. The accuracy of memory accesses, including instruction fetches, can be configured at a very fine granularity, and may change dynamically during simulation.

3 A USER-LEVEL VIEW OF VPSIM

VPSim is a key asset in charge of virtual prototyping and DSE within SESAM, an integrated EDA framework for complex electronic systems ranging from Cyber-Physical Systems to Microservers. SESAM provides a holistic environment to address HW/SW co-design, exploration and validation through Virtual Prototyping, HW prototyping and emulation while taking into account power, temperature and reliability factors. This section provides a high-level view of the major features and capabilities of VPSim as perceived by the end-user.

VPSim is a tool that was designed specifically to accelerate software/hardware co-validation at the earliest design stages of all kinds of computer architectures. It can be used to easily compose, simulate and explore new hardware architectures, but also to run, profile and debug full software stacks on the simulated platforms.

Figure 1 gives a global overview of how VPSim can be used. The user specifies a platform using a *platform composition front-end*, which forwards a *high-level platform description* to a central VPSim object named the *Platform Builder*. The Platform Builder uses this description, and a library of registered *components* to construct and run a SystemC simulation. Components can be either internal hardware models, or *Proxy Components* for interfacing with external subsystems. VPSim loads software binaries through the `ElfLoader` and `BlobLoader` special components. Once the SystemC simulation is launched, the user can use standard debugging tools or VPSim's built-in debug and profiling utilities, to evaluate and optimize their software. VPSim collects *fine-grained per-components statistics* and pushes them back to the high-level front-end, which may use them to present performance profiles to the user or to perform design space exploration.

VPSim is also capable of operating as merely one component of a wider Cyber-Physical System (CPS) simulation, in compliance with the Functional Mockup Interface (FMI) co-simulation standard [8]. In this mode, VPSim is packaged

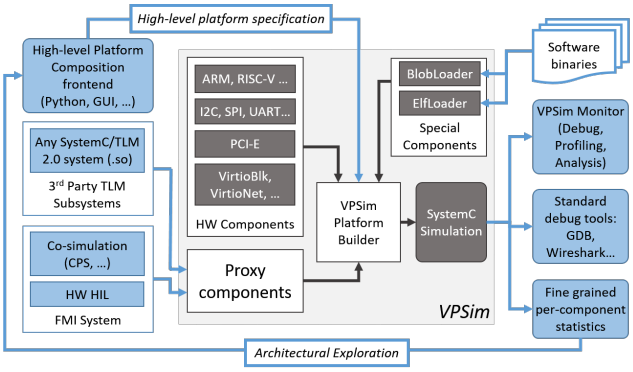

Figure 1: An Overview of the VPSim Platform.

as a Functional Mockup Unit (FMU), that can be loaded by an FMI simulation master. This makes it possible to evaluate the cyber part of a cyber-physical system (modelled in VPSim) along with its surrounding physical environment.

3.1 Component library

VPSim includes a large library of CPU models, buses and peripherals to chose from. Common controllers such as UART, I2C and SPI, PCI-Express from various vendors (Xilinx, Renesas, Cadence, etc.) are modelled in VPSim. Network, block devices and GPUs are also made available through Paravirtualization [21]. All the components that are instantiated in VPSim are fully standard-compliant SystemC/TLM 2.0 modules. CPU and Virtio models are made available to the SystemC world through a new approach for integrating QEMU into SystemC, which will be described in Section 4.

To further broaden its range of supported models, VPSim provides *proxy* components. These components are able to load and connect SystemC/TLM 2.0 initiator and target subsystems from any third-party EDA vendor to the system described in VPSim. The external subsystem is separately compiled in a shared library that implements simple glue functions. Through this interface, VPSim can seamlessly integrate CPU models such as the ARM Fast Models [20], OVP (Open Virtual Platforms) CPUs [17] or QBox [10].

3.2 Composition and Exploration

To enable truly rapid prototyping of complex architectures, it is not enough for simulations to be fast. The time required to fully describe an architecture, configure it, modify it is also of critical importance to designers and software developpers.

Most related tools employ a common scheme, wherein the platform's top is written in SystemC, either manually or generated using a Graphical User Interface, and then compiled. Later on, the configuration of the simulation is usually described in a higher level language such as Lua [10].

VPSim adopts a different philosophy: Compilation shall take place only once, and the whole simulation, including the platform to be simulated, the applications to be run, and the configuration, shall be described using a dynamic front-end. The benefits of this approach are manyfold. For instance, changes in the simulated architecture do not require recompilation. This is a true game changer at early design phases, where it is very common to make adjustments in the architecture itself. Also, the ability to describe the system hierarchy along with the components' configuration in the same environment removes the need to create a custom configuration file format for every new platform.

VPSim also provides the front-end with fine-grained simulation statistics, making it possible to perform both platform specification and Design Space Exploration within the same environment. VPSim communicates with the front-end using the XML markup language, as described in Section 4.3.

By default, VPSim ships with a Python-based composition and exploration front-end, detailed in Section 4.4. Note that unlike other tools that use high-level description languages, e.g. GEM5 [7], VPSim is not tighly linked to its Python interface, nor does it know about its existence. By using XML, which is a standardized data exchange format, many new interfaces can be developped for VPSim according to specific needs. In addition, it makes it much easier to couple VPSim with existing XML-capable tools.

3.3 Debug facilities

A decisive factor in the usefulness of any virtual prototyping tool is its ability to inspect and help understand the behaviour of the system under evaluation. VPSim offers debug and control capabilities out of the box. These are described in what follows.

3.3.1 Standard tool support. VPSim makes it possible to debug individual CPU cores in the system using GDB. Each CPU core in the system has a `gdb_enable` attribute which, when set to `True`, enables a GDB session to control and debug the code it executes.

3.3.2 The VPSim Monitor. The system designer interacts with VPSim through a command line interface named the "VPSim Monitor". At any point during the simulation, the system can be frozen using a key stroke, bringing up a command prompt. Through the Monitor, the user can inspect/alter the memory and register contents, but also reconfigure the entire simulation. For instance, it is possible to change the debug level of any component in the system to enable more or less debug information for the rest of the simulation. It is also possible to create watches on specific address ranges to monitor memory accesses, and possibly manually override the read/written values.

3.3.3 Dynamic Checkpointing. When inside the VPSim Monitor, it is possible to create one or several named checkpoints. Then, at anytime during the rest of the simulation, it is possible to roll back to the named checkpoint to investigate the cause of bugs. When a checkpoint is created, the entire user process is copied using the `fork()` system call. The parent process then sleeps and waits for a wakup signal. Rolling back to the checkpoint simply consists in terminating the current process after waking up the correct parent. The

list of checkpoints and their associated process IDs is transported with each rollback to make sure that all checkpoints are accessible from any process instance. This non-intrusive approach has several advantages. Unlike [14], our method does not require implementing a checkpointing method for each individual component, nor does it introduce changes to the SystemC kernel. Since it operates on the entire process, it works even with closed-source SystemC modules. Compared to other methods that use snapshots of the entire process, such as [11], our method can operate fully in-memory, without having to save images to disk. This makes rolling back to checkpoints created within one same simulation much faster than [11], where several seconds are necessary to store and load the snapshots.

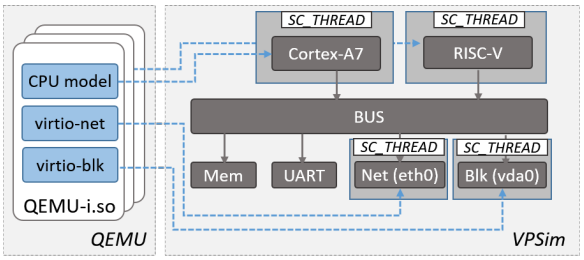

Figure 2: QEMU integration in VPSim.

3.4 Profiling and Performance Evaluation

Unless otherwise specified in global simulation parameters, VPSim attempts to run the target software as fast as possible. That is, most accesses to main memory will be emulated in a fast, untimed fashion based on the Direct Memory Interface (DMI) of TLM components. This mode of operations greatly eases and accelerates the functional validation and debugging phases. However, when optimizing the target software, it would also be helpful to get detailed information about its execution, such as cache usage and network latency.

One specificity of VPSim is its ability to dynamically define regions of interest, in both the software and the hardware. These regions of interest are simulated more precisely, i.e. bus accesses, caches, and other transactions are fully modelled. This helps provide a performance profile for a specific portion of the executed code.

The user can describe the beginning and end of the software region of interest dynamically (during execution), or statically (in global parameters). The hardware components that need to be precisely simulated can also be designated by the user. One common usage is when profiling a Linux user-space application. In such cases, the Linux boot phase is of little interest and can be quickly skipped through in untimed mode. VPSim then switches to a more accurate simulation mode once the user application is entered.

At the end of the software region of interest, statistics from all the simulated components are registered and displayed to the user (e.g. cache miss rate, network latency, executed instructions, etc.). This fine-grained control over the simulation's accuracy allows the user to make the best trade-offs between simulation speed and accuracy given their specific needs, which may evolve from one design phase to another.

4 KEY TECHNICAL CONTRIBUTIONS

In this section, we describe the key technical contributions that we made to enable all the features presented so far.

4.1 QEMU in VPSim

QEMU is one of the leading Dynamic Binary Translation (DBT) -based emulation solutions to date [4]. In addition to unmatched execution speeds, QEMU supports a wide range of virtualized and paravirtualized hardware models, including most of the existing CPU architectures, and many peripherals.

Methods for leveraging QEMU in a SystemC simulation environment have been extensively explored [12], [13], [10], [15], [23], [9]. Some of these methods view QEMU as merely an Instruction Set Simulator (ISS), while others make it possible, in addition to CPU clusters, to model complete subsystems within QEMU, and some peripherals in SystemC [10]. In the latter approach, accesses to the peripherals modelled within QEMU are not visible to the outter SystemC world.

VPSim adopts a different approach. In VPSim, all CPUs and peripherals must implement a common consistent interface and be executed in the SystemC context. This is key to guaranteeing a certain level of predictability, inspectability and compatibility with VPSim's debugging and profiling tools. Therefore, in VPSim, models that are backed by QEMU can be instantiated, configured, and controlled in the same way as native components. For instance, the `VirtioNet` and `VirtioBlk` components, which are backed by virtio-net and virtio-blk devices in QEMU, can be instantiated, mapped to any address, connected to any interrupt line in VPSim. Accesses to these peripherals, as well as the interrupts they generate, are all visible and debuggable through the VPSim Monitor, for instance.

In our approach, no IO accesses are served within QEMU. Instead, all accesses are visible to the SystemC world and completed by SystemC models. Accesses to RAM (fetches and data) can be completed in QEMU for efficiency, but only if the target `Memory` module has a Direct Memory Interface (DMI) as per the SystemC/TLM 2.0 standard. QEMU's CPU execution thread and IO thread are both executed in the context of `SC_THREAD`s and `SC_METHOD`s, making simulations more predictable and controllable. In the absence of external input (Network, Keyboard, etc.), simulations in VPSim are deterministic and guaranteed to be repeatable.

While transparent to the user, internally, VPSim maintains a number of QEMU instances, as shown in Figure 2. Each QEMU instance may be used to instantiate CPUs of the same model, plus any number of peripherals. These models are then associated to proxy VPSim `components`, which are exposed to the user like any other components.

4.2 Dynamic Accuracy Control

In the SystemC/TLM 2.0 standard, some memory-mapped modules may have a DMI (Direct Memory Interface). That is, it is possible to get a pointer to their internal memory space, and access it directly, without simulating the entire TLM transaction. QEMU in VPSim makes good use of this feature by declaring all DMI regions as RAM regions [2], thereby enabling ultra-fast inline accesses to main memory.

However, to enable the accurate simulation mode presented in Section 3.4, it is necessary to perform the full TLM transactions. `Components` in VPSim possess a `DMI_OK` flag, which is active by default. When set to `false`, the component will issue a DMI invalidation request for its address range. Optionally, only a subset of the entire a accessible range can be invalidated. This propagates the invalidation request upstream and will force the upstream component to initiate full TLM transactions for the invalidated range. The upstream component then sets its own `DMI_OK` flag to false, which will provoke the invalidation of its own accessible address space by its upstream initiator, and so on until the root of the device tree is reached.

VPSim can take TLM transactions only as far as necessary to get the desired level of accuracy. For instance, if the user disables the `DMI_OK` flag only on a Cache component, VPSim only makes sure that TLM transactions get to the designated Cache component, which then completes memory accesses using DMI pointers to the final targets. VPSim's `Component` interface automatically manages these pointers internally and is able to use them to complete a transaction when eligible.

In VPSim, components take the port from which these invalidation requests are received into account. Therefore, a `Component` may force TLM accesses only on specific ports, while allowing fast DMI simulation on the rest of the device tree. This is extremely useful when only part of the memory hierarchy is of interest. For instance, forcing TLM accesses on the instruction cache `component` will only force blocking TLM transactions from QEMU when fetching instructions, while data accesses remain unaffected.

4.3 Platform Builder

At the core of the dynamic platform composition presented in Section 3.2, is a central VPSim object named the *Platform Builder*. The Platform Builder interacts with a composition front-end, such as the Python frond-end presented in Section 4.4, and has three roles:

First, it communicates to the front-end the structure of the platform description document that it expects. Currently, VPSim supports platform descriptions in XML format. Therefore, the Platform Builder automatically generates, from the list of self-registered `Components`, an XML Schema Document (XSD) that describes the structure of the expected XML document. This document also includes information about all the available components in VPSim and their attributes. The front-end should use this Schema to present the components to the user. A GUI front-end might display them as boxes that can be connected to each other, for instance. The Python frond-end presented in Section 4.4 uses the Schema to dynamically generate Python classes corresponding to each component.

Second, upon receiving an XML document from the front-end, it elaborates and configures the specified platform. Platform elaboration takes place in three phases:

- **Make**: During this phase, all the components of the platform are instantiated and initialized. The Platform Builder checks whether all the required attributes were specified, sets the specified attributes, and assigns defaults when applicable.
- **Connect**: All connections between components that were specified in the platform description are realized. The availability and compatibility of the connected ports are checked during this phase.
- **Finalize**: The Platform Builder sets some Platform-wide parameters as specified in the input XML document, and, more importantly, invokes a `finalize` callback that `Component`s may implement to perform finalization actions after all the platform has been elaborated.

Finally, at the end of a simulation, the Platform Builder collects per-component statistics and forwards them back to the front-end in an XML document. The front-end may use these statistics to build performance profiles, or for architectural exploration.

4.4 The Python front-end

The default and preferred front-end in VPSim uses the Python language. With this front-end, VPSim strives to provide a dynamic platform composition environment that is compact, intuitive, but also expressive enough to allow detailed platform configuration and description.

To illustrate with an example, Listing 1 shows how a 4-core ARMv8 Platform capable of booting the Linux operating system is described and simulated in VPSim. A simulated `system` is represented by the `System` class. Each `component` is represented by a corresponding Python class, e.g. `Memory`, `Bus`, etc. The `attributes` of each component can be either specified in the constructor for compactness (see Listing 1, Line 9), or as regular class members (see Listing 1, Line 3). This makes it possible to set or modify some attributes programmatically. Of course, each of these components has many more attributes that can be configured. When omitted, VPSim sets them to sensible defaults. Some relationships between components can also be inferred automatically. For instance, since the GIC400 (Generic Interrupt Controller) is the only interrupt controller in the system, it is automatically connected to the interrupt lines of the UART and the architected timers of the CPUs. Calling the `simulate` method interprets the entire system and launches a simulation. At the end of the simulation, per-component statistics are returned in a Python dictionary object. This model allows for as much expressiveness and modelling freedom as one would get writing a Top in SystemC. In addition, we get all the dynamicity, compactness and power of the Python language.

```
from vpsim import *

class ExampleSystem(System):
  def __init__(self):
    System.__init__(self, 'MyExample')

    self.sysbus = Bus(latency=10*ns)

    ram = Memory(base=0x10000000, size=1*GB)
    self.sysbus >> ram

    rom = Memory(base=0x00000000, size=100*KB)
    rom.is_read_only = True
    self.sysbus >> rom

    self.sysbus >> GIC400(base=0xff000000)

    self.sysbus >>
        CadenceUART(base=0xfe000000,irq=0x70)

    for i in range(4):
      cpu = Arm64(id=i,model='cortex-a57')
      cpu.reset_pc = rom.base
      cpu('icache') >> self.sysbus
      cpu('dcache') >> self.sysbus

    BlobLoader(offset=rom.base,
        file='u-boot.bin')
    BlobLoader(offset=ram.base,
        file='example.dtb')
    BlobLoader(offset=ram.base+0x80000,
        file='Image')

if __name__ == '__main__':
  sys = ExampleSystem()
  sys.addParam(param="quantum", value=1000*ns)
  stats = sys.simulate()
```

Listing 1: Simulating a quadcore ARMv8 System running Linux in VPSim

The Python front-end of VPSim was also designed with rapid design space exploration in mind. Because several System instances can live within the same Python script, and because simulation results are made available as Python objects, complete design space exploration can be performed in one same script. To accelerate DSE, the Python frond-end makes it possible to perform the simulations concurrently. To do so, it is enough to call the `simulate` method with a `wait=False` argument, to run simulations asynchronously. This fully integrated approach saves the user from having to write ad-hoc automation scripts for each new study.

5 EXPERIMENTAL SETUP AND RESULTS

In this section, we describe the experimental setup and the results obtained in terms of simulation performance.

5.1 Simulated platform and experimental setup

All the experiments have been conducted on a RISC-V-based simulated platform [6]. It is an SMP platform whose number of cores ranges from 1 to 32, running Linux kernel 4.6.2 as guest. Each simulated CPU has 32KB instruction and data caches, and all cores share an L2 cache of 2 MB and 1 GB of RAM.

This platform was simulated under VPSim on a i7-4770 host machine with 16 GB of RAM running Ubuntu 18.04 LTS. We have checked that the CPU frequency remains stable at 3.9GHz under any load.

When driven by the Accellera SystemC simulation kernel, VPSim is single-threaded and relies exclusively on single-core performance. Simulation performance is never limited by RAM, as less than 200 MB are used in any simulation scenario. The simulations were executed one at a time inside Docker containers based on Ubuntu 18.04 LTS. We have verified that Docker has no significant impact on the runtime performance of VPSim.

The simulated platform runs Linux with various benchmarks from the PARSEC [5] suite: bodytrack, ferret, swaptions, blackscholes, fluidanimate and dedup. The standard *simmedium* datasets were used. The execution sequence starts with Linux boot, followed by the benchmark setup, warmup, the benchmark's Region of Interest (ROI), teardown and finally, the platform's shutdown. Simulation time starts when the `sc_start` function is called and ends when the kernel sends the poweroff signal. Platform elaboration time is negligible with respect to the duration of a simulation.

Depending on the experiment, four accuracy modes are used:

(1) `DMI`: this is the fastest mode where as many transactions as possible are completed using DMI.
(2) `L1-ROI`: DMI is used everywhere except in the ROI for transactions initiated by CPU 0. In that case, TLM transactions reach the L1-caches, which then complete the requests using DMI.
(3) `ACCURATE-ROI`: DMI is used everywhere except in the ROI, where blocking TLM is used.
(4) `ACCURATE`: this is the most accurate mode where all memory accesses are modelled and timed using blocking TLM transactions.

`ACCURATE-ROI` mode is as accurate as `ACCURATE` mode during the ROI, which is likely to be what the user is focusing on in a real world application. `L1-ROI` mode gives accurate information about the L1-caches of a single core during the ROI. In cases where the platform and the benchmarks are symmetric in terms of executed code, the behaviour of one core with respect to caches is likely to be representative of the rest.

The number of MIPS and the real simulation time are collected at the end of each simulation. A total of 3 experiments are conducted:

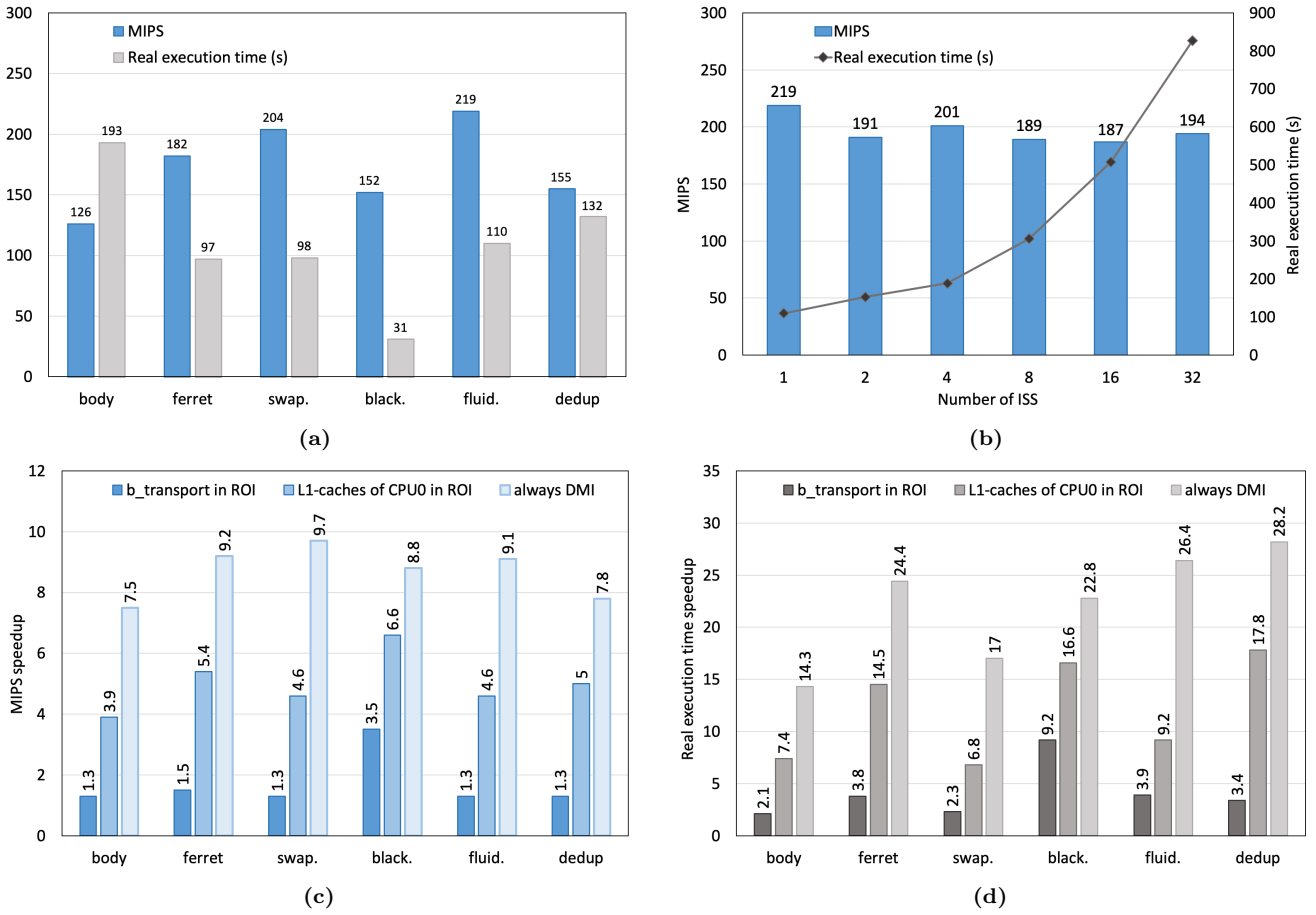

Figure 3: (a) MIPS and real simulation time for various benchmarks on a single-core platform running Linux. (b) MIPS and real simulation time for the fluidanimate benchmark with 1 to 32 cores. (c-d) Speedup with various simulation accuracy levels compared to `ACCURATE` mode on a quad-core platform.

(1) The first experiment only uses the single-core version of the platform and runs all the benchmarks to assess the performance of the simulator in `DMI` mode.
(2) The second experiment focuses on the `fluidanimate` benchmark and simulates it on platforms composed of 1, 2, 4, 8, 16 and 32 cores in `DMI` mode. It shows the effect of platform complexity on VPSim.
(3) The third experiment shows the effects of the various levels of accuracy on different PARSEC benchmarks.

5.2 Simulation results and performance

In Figure 3a, the raw performance of VPSim in fast `DMI` mode on a single-core simulated platform can be observed. The speed ranges from 121 to 219 MIPS and the real simulation time from 31 to 193 seconds. The Linux boot and shutdown processes take approximately 14 seconds. The difference in observed performance mainly comes from varying benchmark profiles and the way this impacts QEMU. Indeed, QEMU performance can be hampered by conditional jumps or memory accesses requiring the use of the software MMU. However, these factors are hard to predict and there is no clear correlation between, for instance, the frequency of load/store instructions of a benchmark and the speed at which it can be simulated.

In Figure 3b, the effect of platform complexity on MIPS and real simulation time is reported. While there is a slight drop in MIPS between the single and dual-core platforms, the speed remains in the noise margin up to 32 cores. This was expected as the overall simulation pattern remains the same: a single core doing the work outside of the ROI and all cores sharing the work inside the ROI. However, the real simulation time increases almost linearly with the number of cores. This can be explained by the sequential simulation of all cores, which introduces extra overhead in single-threaded phases (e.g. boot, setup, etc.). By contrast, the time required to simulate the ROI remains more-or-less constant, since a fixed amount of computation is shared between all cores. As a result, ROI simulation time becomes negligible compared to the other phases, hence the linear increase in simulation time.

Figures 3c and 3d show the speedups obtained when variable accuracy is applied. The reference speed for both MIPS and real simulation time is the `ACCURATE` mode. It can be observed that `ACCURATE-ROI` introduces a MIPS speedup comprised between 1.3x and 1.5x with no loss in simulation accuracy in the ROI. The only slight difference that occurs resides in the cold caches at the beginning of the ROI. However, if the ROI is big enough, this effect is negligible. Also, simulating the caches while not recording the statistics during the first steps of the ROI could mitigate this inaccuracy. The MIPS speedup obtained with `L1-ROI` ranges from 3.9x to 6.6x. In the context of cache performance evaluation on a multithreaded application, this mode may be sufficient. Indeed, if the application makes a comparable usage of all cores and exhibits negligible data sharing, then there is no need to simulate all caches.

The observation on real simulation time is comparable but speedup is larger. It ranges from 2.1x to 9.2x for `ACCURATE-ROI` and from 6.8x to 17.8x for `L1-ROI`. One particularly interesting thing to note is the difference between these speedups and the respective MIPS improvements. The reason resides in the timing accuracy variations. When simulating with maximum accuracy, all memory transactions are timed and participate in the update of the internal SystemC time, which passes faster than in `DMI` mode as a result. Consequently, all actions triggered by timed interruptions occur more often. This concerns several Linux routines such as thread scheduling, IO polling, etc. This variation in the simulated instructions is a side effect of the simulation accuracy changes. It plays in favor of variable accuracy as simulations get even shorter.

6 CONCLUSIONS

This paper introduced several contributions. First, we presented an alternative approach to integrating the QEMU emulator into SystemC. Our approach makes the CPU and peripheral models of QEMU available as TLM modules. One key benefit is the ability to leverage QEMU's high-performance emulation technologies such as Dynamic Binary Translation and Paravirtualization to boost simulation speed. Results show speeds approaching 220 MIPS when simulating workloads on top of Linux. Then, we presented a method for fine-tuning the simulation to get the best performance given specific accuracy requirements. Our framework is based on a clever use of the TLM 2.0 DMI specification. We experimentally demonstrated that by focusing simulation effort only on regions of interest, speedups of over 17x could be achieved. Finally, we showed how it was possible to abstract away all the complexity SystemC architecture modelling, by introducing a sophisticated dynamic platform elaboration infrastructure. By allowing designers to specify their modelling and exploration needs at a higher level, hours of development and compilation time can be saved. All of these promising building blocks are integrated into a unique virtual prototyping solution named VPSim.

REFERENCES

[1] TLM 2.0. 2018. Open SystemC Initiative (OSCI). https://www.accellera.org/images/downloads/standards/systemc/TLM_2_0_LRM.pdf

[2] QEMU Memory API. 2018. QEMU. https://github.com/qemu/qemu/blob/master/docs/devel/memory.txt.

[3] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. A. Penry, O. Temam, and N. Vachharajani. 2007. UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development. *IEEE Computer Architecture Letters* (2007), 45–48.

[4] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX Annual Technical Conference (ATEC)*. Anaheim, CA, 41–41.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. *The PARSEC Benchmark Suite: Characterization and Architectural Implications*. Technical Report TR-811-08. Princeton University.

[6] Andrew Waterman et al. 2014. The RISC-V Instruction Set Manual.

[7] Binkert et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[8] Blochwitz et al. 2012. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In *International MODELICA Conference (MODELICA)*. Munich, DE, 173–184.

[9] Chiang et al. 2011. A QEMU and SystemC-based cycle-accurate ISS for performance estimation on SoC development. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 4 (2011), 593–606.

[10] Guillaume Delbergue et al. 2016. QBox: an industrial solution for virtual platform simulation using QEMU and SystemC TLM-2.0. In *European Congress on Embedded Real Time Software and Systems (ERTS)*. Toulouse, FR, 315–324.

[11] Kraemer et al. 2009. A checkpoint/restore framework for SystemC-based virtual platforms. In *International Symposium on System-on-Chip (SOC)*. IEEE, Tampere, FI, 161–167.

[12] Monton et al. 2007. Mixed sw/systemc soc emulation framework. In *IEEE International Symposium on Industrial Electronics (ISIE)*. Vigo, ES, 2338–2341.

[13] Montón et al. 2009. Mixed simulation kernels for high performance virtual platforms. In *Forum on Specification & Design Languages (FDL)*. Munich, DE, 1–6.

[14] Màrius Montón et al. 2013. Checkpointing for virtual platforms and SystemC-TLM. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21, 1 (2013), 133–141.

[15] M Gligor, N Fournel, and F Pétrot. 2009. Using Binary Translation in Event Driven Simulation for Fast and Flexible MPSoC Simulation. In *International Conference on Hardware-Software Codesign and System Synthesis (CODES+ ISSS)*. Grenoble, FR, 71–80.

[16] C. Menard, J. Castrillon, M. Jung, and N. Wehn. 2017. System Simulation with gem5 and SystemC. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. Samos, GR, 62–69.

[17] Open Virtual Platforms (OVP). 2018. Imperas Ltd. http://www.ovpworld.org

[18] Virtual System Platform. 2018. Cadence. https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/Archive/virtual_system_platform_ds.pdf

[19] Vista Virtual Prototyping. 2018. Mentor, A Siemens Business. https://www.mentor.com/esl/vista/virtual-prototyping/

[20] N Rodman. 2008. ARM fast models-virtual platforms for embedded software development. *Information Quarterly Magazine* 7, 4 (2008), 33–36.

[21] Rusty Russell. 2008. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review* 42, 5 (2008), 95–103.

[22] Simics. 2018. Wind River. http://www.windriver.com/products/simics

[23] TLMu. 2018. Edgar E. Iglesias. https://edgarigl.github.io/tlmu/tlmu.pdf

[24] N. Ventroux, A. Guerre, T. Sassolas, L. Moutaoukil, G. Blanc, C. Bechara, and R. David. 2010. SESAM: An MPSoC Simulation Environment for Dynamic Application Processing. In *IEEE International Conference on Computer and Information Technology (CIT)*. Bradford, UK, 1880–1886.

[25] Virtualizer. 2018. Synopsys. https://www.synopsys.com/verification/virtual-prototyping/virtualizer.html

[26] VLAB. 2018. ASTC. http://vlabworks.com/index.php/products/