2016 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Integrating Different Types of Models into a Complete Virtual System

The Simics SystemC* Library

Jakob Engblom, Andreas Hedström, Xiuliang Wang, Håkan Zeffer, Intel, Stockholm, Sweden

***Abstract*—Virtual platforms construction will typically involve combining component models from many different sources, with models written using different tools, languages, and runtime systems. To enable the convenient and efficient integration of SystemC models into Simics, we have extended the Wind River Simics virtual platform framework with support for SystemC models. The new SystemC Library exposes SystemC objects in Simics in the same way as native Simics models, and allows for uniform inspection and control across models. We create a single system model from the heterogeneous pieces. In this paper, we describe the structure and semantics of the integration, problems encountered and their solution, and some examples of applications.**

Keywords—SystemC, integration, virtual platform, simulation

I. INTRODUCTION

Fast, transaction-level, virtual platforms (VPs) are absolutely necessary in all SoC and platform development projects today. VPs enables pre-silicon development of software at all levels of the stack, including low-level validation software, firmware, boot code, UEFI BIOS, device drivers, and operating systems. VPs enable efficient hardware validation flows by integration with various forms of RTL simulation and emulation. VPs drive performance simulations and help architects estimate performance. VPs provide software developers with access to upcoming hardware and enable silicon customers to develop systems before silicon availability. VPs execute continuous integration and agile automated tests, both for pre-silicon and post-silicon hardware. In order to realize those benefits, the rapid construction of individual device models and overall virtual platform configurations is critical to keep projects on track and silicon product launches successful.

In practice, a virtual platform model is built from many different sources. Moreover, the actual models used typically change over the course of a project and will vary based on use cases. Different types of models and simulation frameworks have different strengths and weaknesses. In order to develop VPs in the most efficient way, we need to use the best tools available for each task. Our experience using Wind River Simics* [1] as a basis for VPs at Intel® [2] and other companies is that combined platforms work well. Typically, an overall platform model is created using the Simics simulation system, and subsystem models are created using a variety of languages and modeling frameworks and APIs [3][4][5][6][7][8].

SystemC [9] is a language and simulation kernel that is commonly used in hardware design tools and flows. Rudimentary support for SystemC has been available in Simics for quite a while, known as the *SystemC Bridge* [3][10]. However, it was limited in capabilities and complicated to use. Over the past few years, the SystemC support in Simics has been revamped and improved. SystemC models and subsystems now run in Simics as efficiently and natively as C, C++, or Python models. Simics inspection and control features apply to models in SystemC, providing a unified interface to all the models in a heterogeneous platform.

This paper describes the new *Simics SystemC Library*, its functionality and most important features, and some examples of how it has been used in practice to build integrated virtual platforms with capabilities beyond what could be achieved using just a single simulation system.

II. SIMICS

Since not everyone is familiar with Simics, we start with a short overview of the Simics platform. Simics is a fast transaction-level virtual platform framework that has been developed over the past 20 years, and which is currently developed by Intel and its subsidiary, Wind River [1][2][11].

Simics is based on an event-driven transaction-level concept. It is designed for high simulation performance and scalability from single processor configurations to large configurations with many hardware units and many processors. In SystemC terms, the current Simics model of transactions — just like that of Mambo [12] and Qemu, — is very similar to SystemC TLM-2.0 LT with zero timing annotations.

As illustrated in Figure 1, Simics is a modular simulation system that provides a small simulation core, an API, and the ability to dynamically load *Simics modules*. A module can, for example, contain a single device or processor model, or all the models of a SoC. A module can also include a set of simulator features relevant for a specific use case. A SystemC model used with Simics is built into a Simics module is dynamically loaded into a Simics simulation, just like any Simics hardware model.

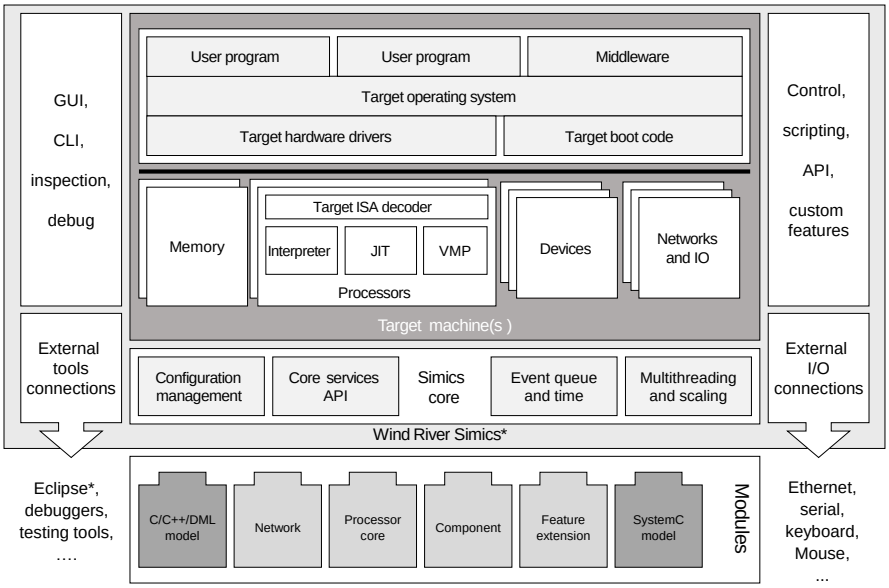

Figure 1. Overview of the Simics architecture.

Simics configurations are *dynamic*, and can be changed during a simulation run. Unlike SystemC, there is no setup phase (elaboration) and execution phase. New pieces of the system can be added at any point in time, and connections between existing objects can change dynamically. When a SystemC model is instantiated in a Simics configuration, we run a local elaboration phase to create the local SystemC hierarchy on a local kernel. This retains standard SystemC semantics for the models, while allowing standard Simics simulation setups to be used as-is.

To make system setup more efficient and to manage large configurations, Simics has a *hierarchical component* system. The component system provides a hierarchical name space to mirror the physical structure of a system, and the component classes themselves usually encapsulate pieces of hardware. When setting up a Simics configuration, you create a few Simics components at the top level, and those components then create configuration objects and subcomponents, and connect them together in order to create a functioning platform. Inside each component is Python code, providing a very flexible system for system setup. Components typically have configuration parameters to create variants of the platform. The SystemC object hierarchy is mapped into the Simics hierarchy, and the SystemC objects currently appear as Simics components in the hierarchy (as seen in Figure 5). SystemC models can also be created from Simics components and be integrated in the standard Simics platform setup system.

Simics supports multithreading to take advantage of multicore hosts, both running separate target machines in parallel and running individual cores in a tightly-coupled shared-memory system in parallel. Simics can do record and replay debugging, as well as reverse debugging [13]. Simics checkpointing allows a session to be saved and restored on any machine, at any location, at any point in time [14]. The SystemC Library makes it possible to leverage Simics features for the benefit of models written in SystemC.

Simics features an *Eclipse GUI* as well a *command-line interface* and *script* support for common tasks. It also contains a full *Python interpreter*. A running simulation configuration can be inspected, modified, and controlled.

The UIs allow users to inspect device registers, memory maps, and other hardware details that are hard to observe on physical hardware. The SystemC library makes those tools applicable to SystemC models. A Simics user does not need to know the source language of a model to use it as part of a Simics configuration. However, to help modelers and advanced users, there are SystemC-specific tools provided within the Simics UIs.

III. USING SYSTEMC IN SIMICS

For the purpose of this paper, we define a "SystemC model" as a model that uses the SystemC [9] API to run. It may or may not use SystemC additional libraries like the TLM-2.0 or AMS. Note that a model does not have to be actually coded in SystemC to be a "SystemC model" - it is common to have models that are written using other languages and runtime systems to be wrapped in SystemC for integration purposes. Such a model, including those delivered in binary form, can be run in Simics using the Simics SystemC library, as long as they can run on the standard Accellera SystemC kernel.

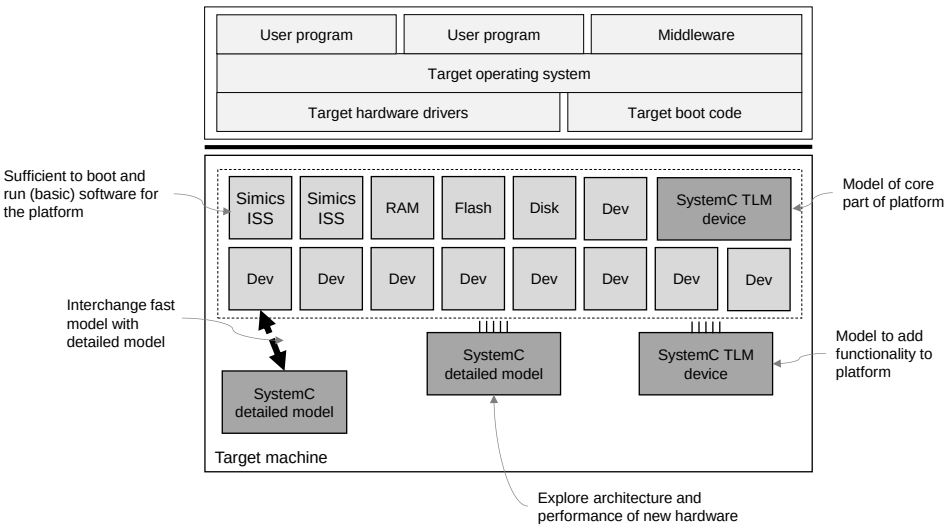

Figure 2. Simics-SystemC common use cases

Figure 2 shows the most common uses of SystemC models in Simics. The Simics base platform provides the ability to run firmware and software and to do so efficiently. The SystemC subsystems provide additional functionality or more detailed models.

For example, architects and validation teams have clock-cycle-level and detailed timing models of their hardware that they want to run in a system context. Such runs can be done by exchanging a simple functional model (often called a black box) for a detailed subsystem model including embedded processors and memories, or by adding on a new device not found in the base platform. Hardware or software teams can have TLM models of hardware that are added to the base platform. Third-party intellectual property (IP) vendors can provide SystemC models of their IP blocks, which are used to build a complete system model. In the end, the language used to create a model and the source of the model does not matter. The setup and composition of the virtual platform is up to the platform developer. Simics does not impose any particular constraints, and there are cases where SystemC is used in Simics without any Simics target models at all – Simics can be used as a SystemC tool, providing control and inspection facilities not available in the standard Accellera kernel.

IV. SYSTEMC LIBRARY DETAILS

As shown in Figure 3, SystemC is integrated into Simics by putting a complete SystemC simulation system including the SystemC kernel into a Simics module. The SystemC model is not modified and runs exactly like it would run in a stand-alone simulation or inside other SystemC tools. Its external interfaces are connected to the interfaces on the Simics side using *gaskets*. To set up the connections, a small piece of *adapter code* needs to be written.

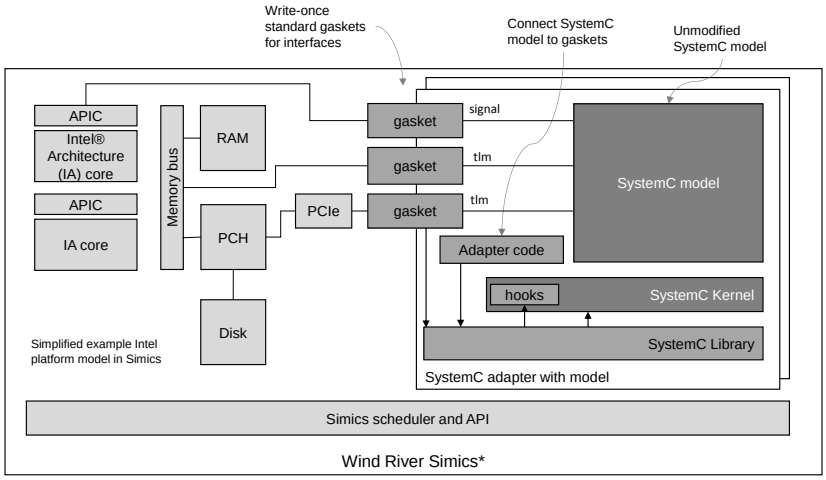

Figure 3. Details of the SystemC integration

A. *A. SystemC Kernel*

As a baseline, Simics provides an *unmodified Accellera 2.3.1 kernel* for use in Simics. This kernel can be used to validate that standard SystemC setups work inside of Simics. Using only the standard kernel features and API, you can inspect the hierarchy of the SystemC subsystem, control the running of the SystemC subsystem from Simics, view TLM socket connections, and get *sc_report* logging rerouted to Simics log messages.

The *default kernel* provided with Simics is *binary compatible* with the Accellera kernel, but adds additional *hooks* for Simics. Being binary compatible, it can run SystemC models provided as binaries as well as those built from source. Any model that runs with the unmodified kernel should work exactly the same with the default kernel.

The default kernel has some hooks patched into the SystemC kernel (invisible to models) to allow deeper inspection and control. It enables the Simics UI to provide insight into the SystemC event queue, to trace and break SystemC events, processes, signals, and sockets, and to profile SystemC models with regards to memory usage and execution time. There are also patches to the SystemC kernel to allow multiple instances of the same model in a single Simics session – the standard SystemC kernel has some global variables that prevents instantiating it multiple times from the same dynamically loaded module.

Another small change is the ability to run with unconnected SystemC ports. This helps when developing a new model: the developer might not be able to connect all the signals and sockets. With the standard Accellera kernel this triggers an assert which terminates the run. The Simics kernel has the ability to run with unconnected ports; only if one of the unconnected signals or sockets is accessed is an error message is printed. The feature is controlled by an attribute on the adapter, defaulting to the standard SystemC behavior.

B. *Time Synchronization*

Each SystemC subsystem integrated with Simics is either connected to the time of a Simics processor model, or runs off of its own Simics clock object. On the Simics level, the clocks follow Simics temporal decoupling semantics, and are never more than one time quantum apart. However, inside each SystemC subsystem, there is also the SystemC kernel time. As shown in Figure 4, the kernel time is decoupled from the Simics time, and can be both behind and ahead of Simics time. The SystemC model only runs when Simics is running – Simics provides run control for the SystemC model and subsystem.

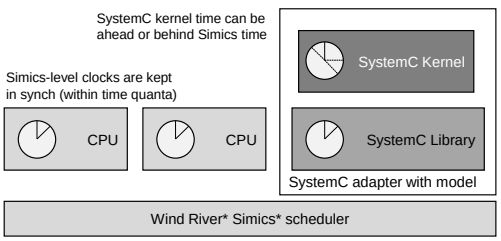

Figure 4. Time in the SystemC integration

For a purely event-driven SystemC model, SystemC kernel time is advanced only when an event happens in the SystemC system (lazy synchronization). Simics knows about the events scheduled inside the SystemC subsystem, and will run the SystemC kernel to bring it up to the current

Simics time when the Simics time in the associated Simics clock or processor has reached the time of the event. When no event is scheduled and no threads are running, the SystemC subsystem is not invoked at all, which essentially means zero overhead for an idle subsystem. These semantics do not change the behavior of the model, since the model-observed time and event order is not changed.

When Simics initiates a transaction into the SystemC subsystem, Simics requires that the result of the transaction is returned to the Simics side and made visible in zero virtual time according to standard Simics semantics [1][10][11]. As a result, SystemC time might be forced *ahead* of Simics time. In particular, we have seen this happen with TLM LT-style transactions that call into a SystemC thread that calls *wait()*, and when running through the protocol steps needed to get the actual result from an operation in a TLM-2 AT-style model. When such a transaction returns to Simics, SystemC time will be ahead of Simics time. Simics will then execute in Simics time, eventually catching up to the SystemC time. However, if new transactions are issued before the Simics time has caught up with the SystemC kernel time, they could push SystemC time even further ahead. With SystemC time ahead of Simics time, it does mean that observed time in the SystemC model and in the Simics model are different, but as long as models do not exchange time stamps, this should not affect simulation semantics and software-visible state and event orderings. Still, it is suboptimal. To solve this problem properly, we would need to extend Simics memory system semantics to allow asynchronous transactions.

C. *Adapters*

Each SystemC model that is integrated into Simics needs to have an adapter. The adapter sets up the connection between the SystemC model and the rest of the platform, creating the gaskets needed to connect interfaces. It provides Simics attributes to allow configuring the SystemC subsystem (pushing configuration values from Simics into the SystemC subsystems), and the attributes to set up the connections between the SystemC subsystem and the rest of Simics. Which other models the SystemC model is connected to is dynamic, just like any other Simics connection between Simics objects.

From a Simics perspective, the entire SystemC subsystem in the adapter is considered as a single unit by Simics. The set of SystemC objects and their connections is set up exactly as they would be in a stand-alone SystemC simulation. Once created, the internal structure of the SystemC model is fixed for the rest of the simulation run. Simics attributes can be added to the source code of the SystemC model to allow more extensive internal inspection and configuration of the model. SystemC models written using the Intel-internal modeling framework ISCTLM [7] automatically expose attributes for register inspection, state inspection, and back-door access.

Adapters are written *entirely in SystemC*, which was an important design goal for the SystemC Library. The old Simics SystemC Bridge [10] required the user to write code both on the Simics side and the SystemC side, which was rather cumbersome and forced users to understand two different APIs in order to integrate the models.

D. *Gaskets*

Simics models communicate with each other using interfaces (sets of C-language function pointers) [1][11]. Simics provides predefined interfaces for interconnects such as generic memory operations, interrupts, Ethernet, I2C, serial, SPI, MDIO, PCIe, SATA, and IDE. To connect a SystemC model into the Simics system, a conversion is needed, which is the purpose of the *gaskets*. Gasket classes provide a way to capture a conversion for reuse, which is much more convenient than having to code the same conversion over and over again or copy-paste code.

The Simics SystemC Library provides a set of standard gaskets. It supports the TLM-2.0 memory-mapped bus, *sc_signal* (which is commonly used to model interrupt and reset lines), PCIe (based on an Intel-internal implementation of PCIe in SystemC), and some Intel-internal interfaces. The set of interfaces is expanding over time as new use cases are discovered. The TLM-2 gaskets supports the Direct Memory Interface (DMI) for transactions between SystemC and Simics. Gaskets can be *composite*, combining multiple interfaces into a single gasket. This is used for PCIe, for example, where both memory-mapped accesses and interrupts are logically combined into a single interface.

Outside of the SystemC subsystem, all transactions are Simics transactions, and if one SystemC model communicates with another SystemC model through Simics, they would not be able to tell the difference from

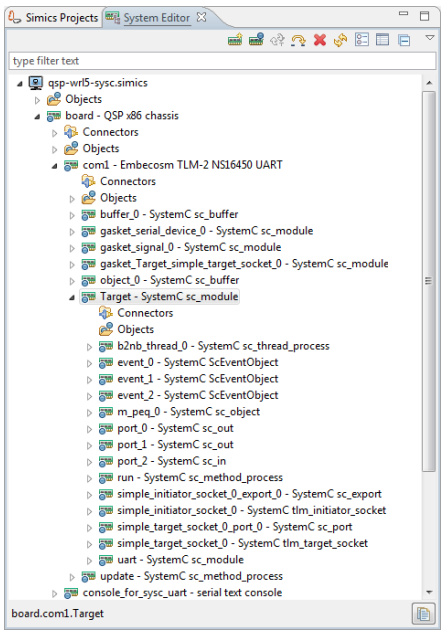

Figure 5. SystemC subsystem in the Simics hierarchy

communicating with a standard Simics device. This is by design, since the goal is to freely mix models of different types inside of Simics. This essentially makes Simics into an integration framework between separate SystemC models, as well as models written natively for Simics and created using other simulation frameworks and integrated with Simics. As long a model can run in Simics, it can run together with any other model with a Simics adapter.

E. Inspection Facilities

Figure 5 shows how an example SystemC subsystem looks inside a Simics target model. The "com1" object is the top-level adapter that provides the interface between Simics and the SystemC subsystem, and the objects underneath it are the SystemC modules that make up the model.

Inspection of the SystemC subsystem is integrated with the standard Simics inspection mechanisms, mapping concepts between the simulator APIs. SystemC *sc_report* maps to the Simics log mechanism, allowing Simics log settings to be used to control the level and verbosity of output. Output is collected together with logs from the rest of the virtual platform. Another example is the ability to show SystemC connections, such as port and export connections, in the Eclipse GUI.

Simics standard break and trace apply to the *SystemC subsystem as a whole*. They break and trace on the interface between Simics objects and the adapter. In addition, the Simics SystemC Library provides commands and tools to look inside the SystemC subsystem. This functionality is useful both to debug the SystemC model itself, as well as software running on any processors inside the SystemC subsystem. Standard commands provide the ability to break and trace on processes, sockets, events, ports, and signals, as well as to write VCD traces. There is also a TLM protocol checker. Users can add their own tools using the Simics instrumentation framework, for example custom statistics modules or pretty printers. The Simics SystemC Library also provides the ability to profile the memory and processor usage of a SystemC subsystem, in order to help understand bottlenecks and optimize models.

F. Register Metadata and Attributes with ISCTLM

The inspection and control features we discussed above only rely on the standard SystemC kernel and TLM features, and are limited by what the standard covers – essentially the model structure and connections between modules expressed using SystemC signals, ports, and TLM sockets. In a virtual platform, you typically also want to inspect device registers, and use back doors to access and modify the state of a model.

At the time of writing, there is no SystemC standard in place for this tool-to-model interface. Lacking a general standard, we have worked with groups inside of Intel to connect the features of the ISCTLM modeling library [7] to the Simics user interface. The features of ISCTLM have been mapped to the Simics UI, providing rich inspection abilities while still allowing the model to run outside of Simics. Other SystemC modeling libraries can be connected in a similar way. The Accellera SystemC CCI WG is working on standardizing the tool-to-model interface, and such a standard would make it possible for us to provide model inspection facilities out of the box for any model using the standard.

G. Performance Impact

A common concern for simulator integrations is that the integration itself will add overhead, reducing performance of the combined solution compared to each individual solution running on its own. In a number of internal projects at Intel, the Simics SystemC Library does not appear to introduce much friction, and temporal decoupling benefits are maintained. For example, when comparing the boot time of a complete platform with and without an integrated SystemC model, the boot times are at most 15% slower with the integrated SystemC model

in place. This indicates that there is no inherent overhead in the integration per se, and that slow-down can be explained with the different behavior and detail level of the SystemC model compared to a simple black-box model.

V. Application Examples

A. *Audio White-Box Model*

When developing Intel platform models, some components are modeled both as black-box and white-box models. A black-box model is a single functional model that models the visible behavior and interfaces of a complex subsystem such as secure boot, audio or board management controller (BMC).

A white-box model, on the other hand, models the actual internal structure of the subsystem and runs the firmware found on embedded processors. White-box models are often written in SystemC, and Jonack and Ambel describe such case [7] where a SystemC-based model of an audio unit was integrated with a Simics model of an Intel platform. The paper reports on the performance optimizations performed to make the SystemC model run quickly on its own, and the effect of integrating the subsystem model in Simics. Overall, it was shown that with a reasonable temporal decoupling time of around 10000 cycles, integrating a Simics system with the audio system did not slow the execution down any more than you would expect from adding a second processor to a system. In addition, Simics has been used to run the audio model alongside other white-box models developed using a third separate simulation framework – basically using Simics to run two entirely different simulators together.

B. *SystemC Multithreaded Execution using Simics*

Simics has a well-established and solid system for running multithreaded simulations. Using the Simics multi-machine accelerator introduced with Simics 4.0, and the multicore accelerator introduced with Simics 5, Simics can take advantage of multiple host cores to accelerate the simulation of both tightly-coupled and loosely-coupled parallel systems, inside a single host process [1][11]. In contrast, the SystemC kernel is limited to running in a single thread, due to implementation issues and language semantics. There is research ongoing on parallelizing within a single SystemC model [15]. However, using Simics, Khan et al rather ran multiple SystemC models in parallel inside a single Simics process [16]. The setup is illustrated in Figure 6.

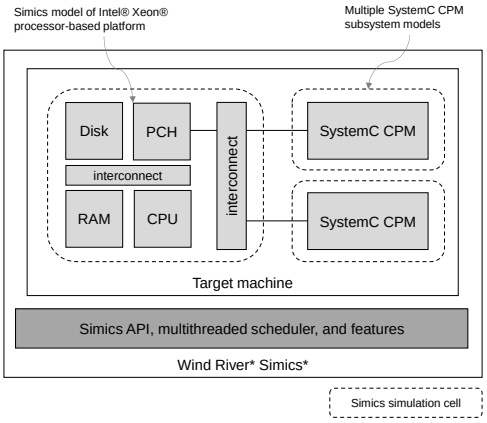

Figure 6. Multithreaded Simics-SystemC simulation

Their approach put the model of the basic Intel platform of the target system into one Simics cell (the Intel® Xeon® processor CPU, its Platform Controller Hub (PCH), and supporting objects like RAM and disk), while running one or three SystemC-based Content Processing Module (CPM) models in their own Simics cells. By using Simics cells and multithreading to do the parallelization, they avoided the need to add threading to the SystemC models themselves. Instead, multithreading was added on the outside, using the existing Simics facilities (plus an experimental interconnect). This means that the same SystemC source code was used as when the models were used in a single-threaded Simics run and used stand-alone outside of Simics; parallelization did not actually require changes the CPM model.

The models contained embedded processors and many detailed model, and thus they were costly in terms of execution. In the end, the benchmarks used ran from three times to 10 times faster than the single-threaded time-synchronized baseline, on a four-way host. The speed up was more than linear due both to some timing tweaks, and the greatly improved locality of the resulting model. This case demonstrates that Simics can serve as a general simulation framework, providing powerful capabilities to integrated SystemC models in a way that cannot be easily achieved using just SystemC itself. It shows the benefit of combining simulators and mixing simulation kernels to get the best of both worlds. In the case of the CPM, SystemC was used to enable models with more microarchitectural details and fine-grained timing and structure than what is typically the case when building Simics TLM models using the Simics native API and modeling tools. . At the system level, the existing ready-to-use Simics

Xeon processor-based server model provided a way to quickly run a real software stack, using the standard very fast Simics level of abstraction and the Simics tool features.

VI. SUMMARY

In this paper, we have described the new Simics SystemC Library. This Library offers the ability to run SystemC models inside of Simics, without requiring any Simics programming or knowledge. The integrated models run just like native Simics models, including taking advantage of multithreading and Simics run-control and inspection features. In addition, some features have been added to Simics to specifically support SystemC model debug and profiling, providing a good environment for SystemC developers. The development of the integration has been done in collaboration with a series of Intel projects. The result is a fast integrated system, with no discernible overhead to attribute to the integration, allowing system models to be written from a heterogeneous set of basic device and processor modelss.

VII. LEGAL NOTICES

REFERENCES

[1] Daniel Aarno and Jakob Engblom, Software and System Development using Virtual Platforms - Full-System Simulation with Wind River Simics, Morgan Kaufmann Publishers, 2014.

[2] Stuart Douglas (ed), Simics Unleashed – Applications of Virtual Platforms, Intel Technology Journal , volume 17, September 2013.

[3] Asad Khan and Chris Wolfe. “Simics-SystemC Integration”, Intel Technology Journal, Volume 17, Issue 2, September 2013.

[4] Matthew Liong. “Performance Analysis with Hybrid Simulation”, Presentation at the Freescale Technology Forum (FTF), November 2008.

[5] S. Koerner, et al.,. “Firmware verification and simulation in IBM zEnterprise 196”, IBM Journal of Research and Development, Volume 56, Issue 1/2, January-March 2012.

[6] Greg Wright, Phil McGachey, Erika Gunadi, and Mario Wolczko. Introspection of a Java™ Virtual Machine under Simulation. Sun Labs Technical Report SMLI TR-2006-159, September 2006.

[7] Rocco Jonack and Juan Lara Ambel, “VP Performance Optimization – How to Analyze and Optimize the Speed of SystemC Models”, Proceedings of the Design and Verification Conference Europe (DVCON Europe), München, 14-15 October 2014.

[8] Lee Liang, Design and Implementation of an Extendable SoC Virtual Platform in SystemC-TLM 2.0, Master’s thesis, School of Information and Communication Technology (ICT), Royal Institute of Technology (KTH), Stockholm, Sweden, 2012.

[9] IEEE Std 1666-2011, IEEE Standard for Standard SystemC® Language Reference Manual, IEEE, January 2012.

[10] Marius Monton, Jakob Engblom, and Mark Burton. “Checkpointing for Virtual Platforms and SystemC-TLM”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol 21, Issue 1, pp. 133-141, January 2013.

[11] Jakob Engblom, Daniel Aarno, and Bengt Werner. “Full-System Simulation from Embedded to High-Performance Systems”. Processor and System-on-Chip Simulation, Rainer Leupers and Olivier Temam (ed), Springer New York Dordrecht Heidelberg London, 2010.

[12] J. L. Peterson et al. "Application of full-system simulation in exploratory system design and development", IBM Journal of Research and Development, Vol 50, no 2/3, March/May 2006.

[13] Jakob Engblom. “A review of reverse debugging,” Proceedings of the System, Software, SoC and Silicon Debug Conference (S4D 2012), Wien, Austria, September 19-20, 2012.

[14] Jakob Engblom. “Transporting Bugs with Checkpoints,” Proceedings of the System, Software, SoC and Silicon Debug Conference (S4D 2010), Southampton, UK, September 15-16, 2010.

[15] Weiwei Chen, Xu Han, Che-Wei Chang, Guantao Li, and Rainer Dömer. “Out-of-Order Parallel Discrete Event Simulation for Transaction Level Models”, IEEE Transactions on Computer-Aided Design Of Integrated Circuits And Systems, vol. 33, no. 12, December 2014.

[16] Asad Khan, Weiqiang Ma, Chris Wolf, and Bengt Werner. “Multi-Threaded Simics SystemC Virtual Platform”, Proceedings of the 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Austin, Texas, November 2015.

[17] Rechistov, G. “Simics on the shared computing clusters: the practical experience of integration and scalability”, Intel Technology Journal, Volume 17, Issue 2, September 2013.