# VP Performance Optimization - How to analyze and optimize the speed of SystemC TLM models?


## Authors:

Rocco Jonack; rocco.jonack@intel.com; +49 89998 8532 5061
Juan Lara Ambel; juan.lara.ambel@intel.com; +49 89998 8532 4732
Intel; Am Campeon, Neubiberg; Germany



## Abstract

With the dramatic increase of complexity in modern SoC (System on Chip) design and the increased time to market pressure, a critical need arises to make software architecture decisions early in the stages of a SOC project in parallel with the hardware development process. These architectural decisions can impact the market introduction and success of a product significantly. Software is, more and more, the driving element of what defines the success of a product. This paper discusses briefly approaches to enable early software development using virtual prototypes, and how SystemC TLM2 models can be used for this purpose. Such an approach implies several requirements for the development process and for the underlying modelling infrastructure. This paper will focus on discussing the simulation speed requirements. This paper explains how and why speed is an integral and fundamental requirement for Virtual Prototype (VP) modelling.

This paper includes several key topics, motivation and general approach for meeting a VP´s speed requirements and how a VP must fit into the development flow. As in most areas of SoC development, infrastructure, tools and IP play an important role for an efficient development process of VPs and for speed analysis and optimization. This paper contains an overview over such tools and strategies, as well as examples on how such tools can be used efficiently to ensure that the speed requirements can be achieved.

The included examples show how speed analysis and optimization techniques are applied for SystemC TLM-based VP models. VP models including a processor running an operating system and different subsystems with and without their own processing units are used for performance measurements. The optimization methods consist of several different strategies. The usage of Quantum in SystemC TLM2 models stands out because it proved to achieve speed improvements in orders of magnitude. The paper also discusses how different IPs and tools impact the VP performance.

Finally the results and benefits are summarized and an outlook is provided for future work and improvements.


## 1 Introduction

With software development becoming the fastest growing component of non-recurring engineering costs for both SoC and final product development, the challenges of developing, integrating, validating, and optimizing software dominates the embedded design process. Thus it has become a necessity to make a fast, accurate, low cost simulation model of the hardware available to the embedded software team very early in the design process. At the heart of the Virtual Prototyping solution are two distinctive components: the creation of the transaction-level modelling (TLM) platform and the usage of the virtual prototype. These typically correspond to two types of users: the TLM platform creator (typically a SoC/system architect or a hardware designer/modeller) and the virtual

prototype end user (typically a software or firmware engineer). The work that leads to this paper was done for a project where a TLM2-based subsystem was modelled and integrated into a platform model. The user of the VP model often has a very different view of the system. Virtual prototypes are a great vehicle to bridge the gap between hardware and software views of a SoC, but it can also become the boundary where issues between the two views become apparent.

Providing a good VP model implies a number of different things. For instance

- Good debuggability of software contents
- Good feedback of important events and values (interrupts, register values)
- Functional correctness of the necessary features
- Appropriate speed of execution

While all of those features are important for a VP model and would deserve a dedicated discussion, this paper will focus on the subject of speed of execution. In our experience the lack of execution speed can lead to reduced acceptance and in many cases even to rejection of a VP model by users. This is not to say speed is the only important feature of a VP model, but acceptable speed is typically a necessary requirement. What an "acceptable" speed is can vary depending on the usage model.

# 2 VP modelling approaches

## 2.1 System Components

A VP is a model of a hardware platform (or at least parts) simulated on a different hardware. Different techniques are being currently used to achieve this. Very often at the heart of a VP model is one or several instruction set simulators (ISS) supplied by the processor vendor or some other party which guarantees the accurate software execution on this model. One alternative can be native execution of SW/FW (software/firmware) on the host. A third option is to develop models for a system oneself. Very likely a complex system will contain a mix of these approaches. When mixing different models it is important to consider how to connect the different components. Last but not least tools and a good infrastructure help to analyse and to improve simulation speed.

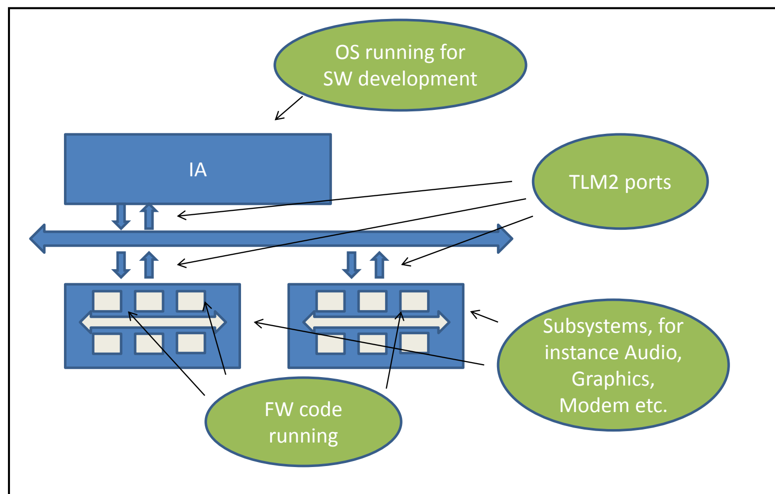


Figure 1: Principle structure for VP model

Figure 1 shows the principle structure of our VP models. There might be several components running firmware (FW) and the structure of the main processor might heavily vary, but the shown structure is representative.

### 2.1.1 Using 3rd party models

If the hardware platform which should be modelled as VP contains a 3rd party processor, the most intuitive approach is to use an ISS model provided by the 3rd party provider. Most vendors provide such models for various purposes like architectural analysis, verification etc. It is important to verify that such a model provides a reasonable performance and usability for an interaction into an overall VP model. This means setting up performance tests in order to check the achievable simulation speed (will be described in later chapters).

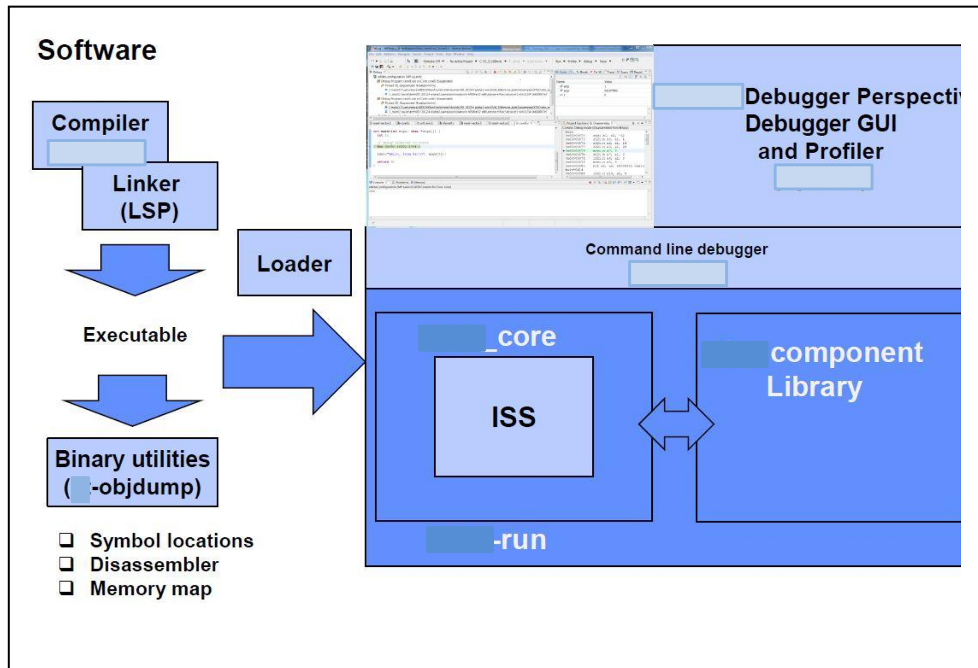

Figure 2: ISS model and tool chain

In our case the main processing units of the subsystem that has been modelled are using a standard ISS. There is a tool chain, which allows the connection of our own models and to provide hooks to the tool environment (debugger, profiler etc.) when needed.

Just like the ISS other components of a platform might also be used from 3rd parties. Examples would be DMA, memory, interconnect or peripheral models. Especially commodity models, which follow well known standards, should be used as predefined units. For those models the speed also has to be assessed carefully based on the requirements -ideally in individual test environments in order to pin point potential issues early on-.

### 2.1.2 Modelling the OS

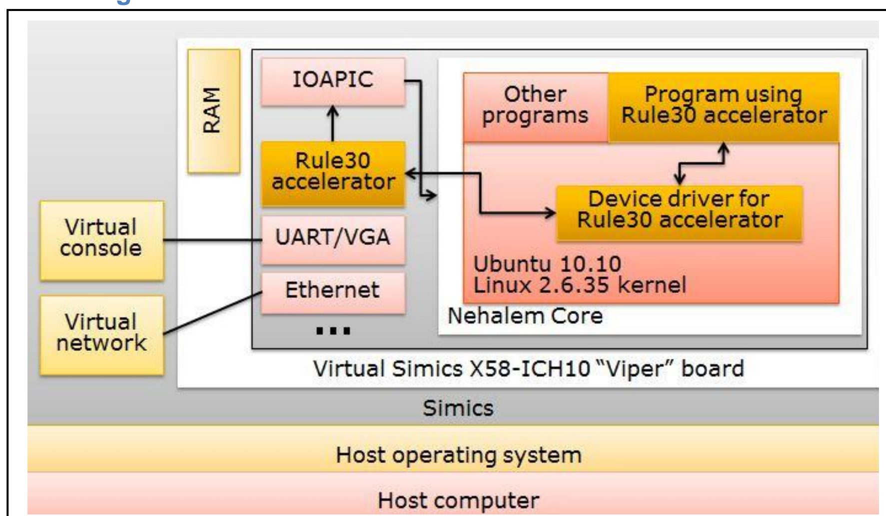

Figure 3: Structure of Simics platforms

In order to model an operating system running on the VP the main processor has to be modelled. Since the OS is requiring a lot of system resources it can be very challenging to also meet speed requirements. Consider that for many VP use cases it is important to provide good reactivity to OS input devices.

In order to satisfy those requirements we are using the Simics environment for the modelling of the main processor, connectivity and other system components. The Simics environment allows modelling of a processor efficiently and the resulting platform provides a good simulation performance which allows running a complex OS like Windows8, various Linux distributions or Android. For our project we use an existing Simics processor platform on top of which the guest OS (Linux) is running.

Simics is a simulation technology and framework built around a proprietary PV (programmers view) modelling language called DML. The host platform elements are hence modelled after PV, but Simics allows co-simulation with SystemC models. Simics has its own scheduler, with which it drives both the PV models and the SystemC sub-module. In this situation basically we try to execute two independent time simulation basis while keeping data consistency across domains.

### 2.1.3 Using Virtualization

Virtualization has become a mainstream technique to run an operating system on top of an already running host operating system with the help of a hypervisor. Several software products use this technique mainly to improve the efficiency with which users can access modern high performance servers. One interesting usage model for VP development is running the OS of a SoC as a guest OS in a hypervisor and to connect other parts of the VP to this guest OS.

This can be very interesting for a VP application where subsystems have to interact with the overall platform (and hence with the OS, too). Virtualization provides OS interaction in real time, which allows for very interactive usage scenarios. However interacting between such an OS running in a hypervisor and models which are not part of the existing host hardware can be tricky. This is because these models might fall short to model correctly all resources that the OS is using, as the OS is running on the host machine and not on a full-blown model of the target SoC.

Similarly to Virtualization, the technique of host-based simulation where the FW code would be executed directly on the host machine can be an alternative. Such a simulation would allow typically for better simulation performance and better integration with other models, since all parts of the system would be in this case compiled by the same compiler for one host system. But on the other side a host-based simulation requires a layer around the FW which translates calls to the VP platform from something that executes on the host to something that executes on the VP platform (HAL layer adaptation). That approach also doesn't allow for connections to the 3$^{rd}$ party debugging environment. We decided not to invest in this approach for our project, since the ISS based simulation did allow for sufficient simulation speed.

### 2.1.4 Developing models

Typically there are a number of models which have to be modelled by development teams during a SoC project. Those are for instance new units or units which have significantly changed compared to previous platform versions. It is very important to consider speed implications when designing such models right from the start. In our experience using a standard framework like TLM2 and strictly following the TLM2 methodology allows the developer to design models which can be very fast. It is important to be aware of the modelling techniques and their impact on simulation speed, for instance the differences between AT and LT modelling and how to apply a quantum.

Traditionally, in VP modelling there is the ubiquitous trade-off between abstraction and computational effort. Time can be abstracted out in several ways giving as a result different timing granularities. In general, the higher the granularity the higher the computation effort while simulating. Several modelling styles have been evolved and some of them form part of industry standards as e.g. TLM2.

For our purpose we can classify some of these popular styles with respect to timing granularity as follows:

- PV (programmer´s view) or SV (SW´s view): typically, a master advances its local time line ahead of global simulation time by a given amount (known as “quantum“). An example would be a CPU model executing a number of instructions in a row. The transactions issued by the master towards HW peripherals are pure functional calls and have a non-blocking character. The possibility of synchronization among masters, or between the HW back to the master is constrained to happen at the endpoints of these “time-slices”.

- Loosely-timed (LT) as defined in the TLM2 standard: transactions issued by the master can be potentially blocking depending on slave implementation because delay is allowed to be consumed in the slaves. Simulation can be speeded-up by means of the temporal decoupling mode, wherein transactions become pure functional calls and slaves are not allowed to consume any delay. Instead, slaves annotate back to master in the transaction object how much time the master shall consume. Once consumed time by master reaches the quantum threshold, the master hands simulation control over to other masters. Hence, as in PV, synchronization among masters is only allowed at the end of a quantum. A speed-up add-on of TLM2-LT is also the Direct Memory Interface (DMI), whereby a master gets a pointer to a slave´s memory and thus can manipulate it directly by-passing e.g. complex interconnection-networks.

- Approximately-timed (AT) as defined in the TLM2 standard. This style interface is not used in our VPs even though many components would support such transfers, which would be important when a model is later refined for other applications, as design space exploration.

The usage of temporal decoupling in SystemC Audio, as defined in TLM2-LT, helps reducing the number of context switches between SystemC processes during the simulation. If a slave accessed by a master is not advancing simulation time but returns immediately the access as a pure functional call the efficiency is significantly improved. The mechanism of the slave returning the delay back-annotated to the master and the master advancing its local time only when the quantum has been exceeded minimizes the number of context switches at the master side to one per quantum. If this practice is enforced at all masters in the SystemC model this leads to a dramatic speed-up, to an extent that temporal decoupling becomes mandatory.

It is paramount to plan for test cases to verify the speed. Once a system is assembled, it becomes more and more difficult to find bottlenecks. Planning and testing components first individually can significantly reduce the effort. Also reusing existing components, for instance from infrastructure libraries, typically reduces the risk to introduce speed bottlenecks. Nevertheless even reused components should be checked for simulation performance.

### 2.1.5 Model connections

We are using TLM2 connections throughout our VP model development. Nevertheless some models or tools have different requirements and gaskets have to be introduced to gap differences. Those gaskets have to be carefully designed in order to provide sufficient functionality, but at the same time fast performance. The gaskets that have been used in our project are mainly around connecting the Simics platform (DML to TLM2) and the ISS model.

When putting a SoC model together it is best to have a single executable for simulation performance. This is only possible if all models in a system use a consistent compiler and library dependencies. It is unfortunately very common to have one or several models which do not allow such a linking approach. In this case an alternative can be socket based connections. Sockets are very flexible and easy to establish. The drawback however is that socket connections use operating system calls, which implies a pretty significant runtime overhead whenever data has be transferred over such a socket. It is very important to plan carefully where such connections make sense. If the amount of

data or better the frequency of communication through a socket is too high, the simulation speed will be defined by the socket connection alone. We were able to avoid such connections in our VP model.

## 2.2 Heterogeneous models

In a typical mobile platform-SoC several subsystems execute SW or FW, which thus lend themselves to be modelled as individual VPs. A typical example would be the OS or application SW running in a general purpose CPU and the FW for an integrated multimedia device incorporating one (or many) special purpose processors, as DSPs.

A VP adjusts its degree of abstraction to the target SW or FW, and hence also its modelling style. Generally, systems running SW are more complex and span much more peripherals than those running FW. Hence, for the former higher abstraction levels are necessary in order to meet a simulation speed requirement while for the later, lower abstraction -higher granularity- levels are affordable. Consequently, a platform-(SoC-wide) VP can be also heterogeneous in terms of modeling styles. This poses extra performance challenges related to the data synchronization between domains belonging to different styles, and extra care has to be taken at the realization of the interfaces at the boundary between domains.

## 2.3 Infrastructure and tools

In order to model a VP efficiently it is very important to use a strong infrastructure. Such an infrastructure must provide base classes for debugging and error logging, tracing and configuration. Also a number of common functional units like memories and interconnect structures should be provided in base libraries. Such base models can significantly improve the efficiency of modelling and provided those models are well tested can also help avoiding runtime issues. Examples for such libraries are SCML or ISCTLM (Intel SystemC TLM library).

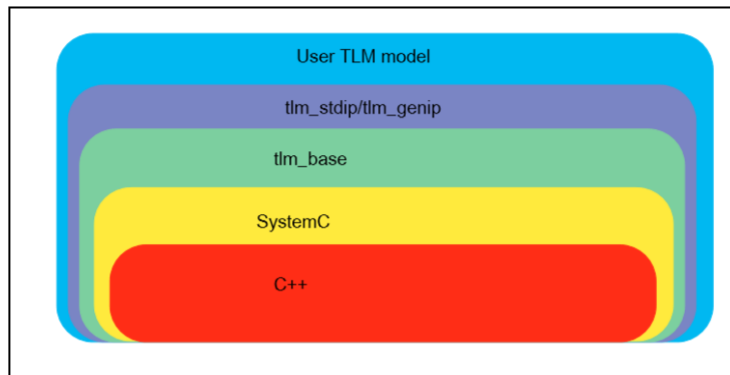


**Figure 4: ISCTLM Library hierarchy**

There are a number of tools which support VP modelling. There are many aspects which are addressed by such tools, but we will focus on the simulation speed in this paper. Most SystemC based tools do not significantly boost the simulation speed compared to the equivalent OSCI simulation. However there are analysis tools available which help analyse the simulation performance. The profiling capabilities have been used in our project and can significantly improve simulation speed especially when 3$^{rd}$ party components are used. We have used the SystemC profiling tool. Classical function profiling tools like vtune, valgrind or gperf are also useful. However the SystemC-aware profiling provided more significant information in terms of speed analysis.

# 3 Speed analysis and optimization

In this section we apply the techniques described in previous sections, first on the component level and later in a case study of a real-life, integrated VP. For the later a measurement setup is described and the obtained results are displayed.

## 3.1 Methodology for analysis

Before analyzing the speed of the entire VP system it is useful to break a model into subsystems and analyse their speed independently. This divide and conquer approach allows elimination of

bottlenecks early on. Such individual test cases can also be useful for other purposes, for instance testing certain specific features in a standalone environment or testing interoperability of 3rd party components in the target system.

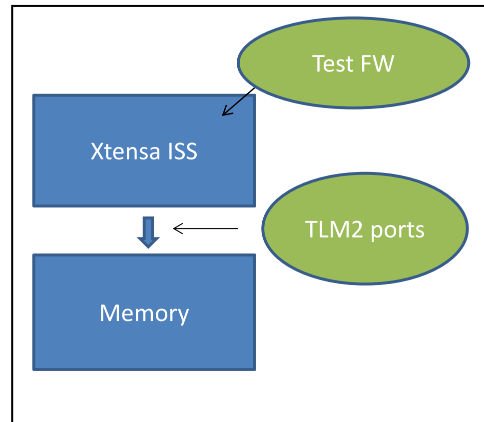


Figure 5: Setup for ISS speed tests

For the VP that has been the basis for this paper we tested 3 different types of subsystems:

- The part of the system which runs the OS, in our case a Simics model running Linux
- The part of the system which runs the FW, in our case a ISS (Figure 5)
- The rest of the system, in our case various SystemC TLM2 based models partially reused from earlier projects and partially developed specifically for this project (variations of figure 6)

Running tests for the first 2 subsystems requires the definition of reasonable and pertinent test cases. It is very important to consider carefully the kind of test cases, because in case the results show speed bottlenecks a discussion with the responsible groups/companies has to focus on finding a reasonable solution. Such discussions have to be based on reasonable requirements otherwise a lot of time can be wasted in this step. On the other side it is very important to understand the fundamental limitations of a model in order to avoid bad surprises when running the full system.

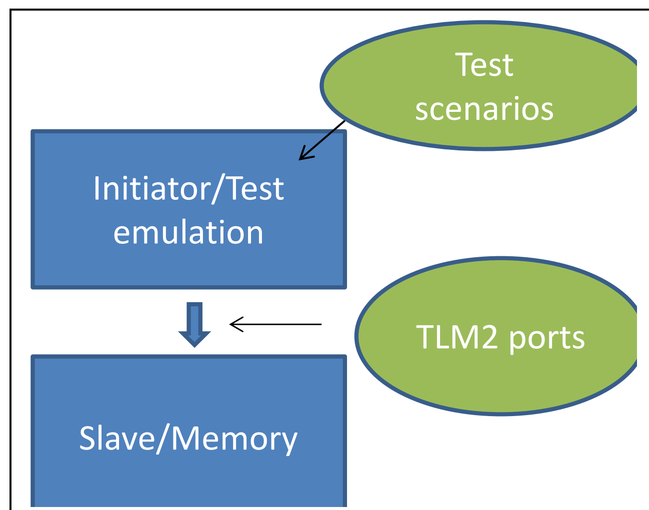


Figure 6: Generic setup for speed test of SystemC components with TLM2 ports

During our project we found, that both the Simics as well as the ISS model, were able to run the required test scenarios with sufficient speed, but in some cases it is necessary to use specific modes (e.g. internal host-based optimizations). Several test scenarios were developed in order to test such settings.

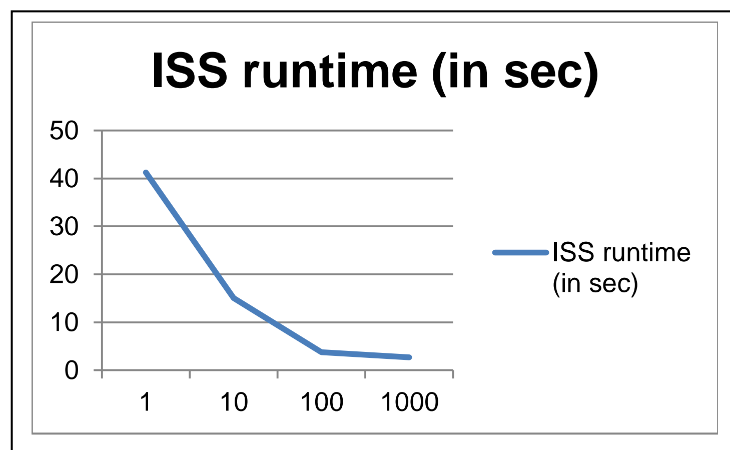


Figure 7: Results for runtime (y-axis) over quantum setting (x-axis)

For the models which were either reused or developed for the project, we also developed test scenarios. For those models we were able to optimize the speed ourselves since all sources are available. The TLM2 library already contains a lot of infrastructure and documentation about the subject, which was useful as reference and common development basis.

Initial tests showed quickly that significant speed bottlenecks were introduced by context switching (*wait()* statements or switching between methods). By using LT function calls of TLM2 ports it was possible to minimize the number of context switches when transferring data. In order to further reduce context switching we strictly avoided *wait()* statements in the slave calls, which handle the *b_transport()* calls and rather used timing annotation. Even though this is not enforced by the standard, in practice it can be very important to adhere to such a style. Otherwise applying a quantum may not yield the expected speed up, because the context switch in the slave interrupts the quantum of the accessing master.

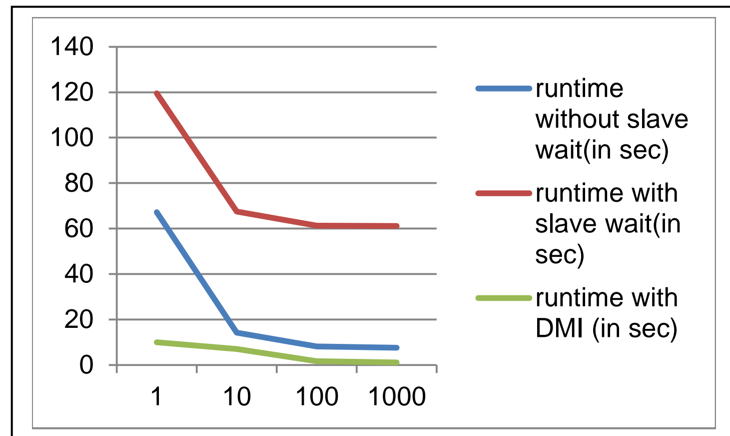

Figure 8: Results for simulation runtime (y-axis) over quantum (x-axis) comparing b_transport() and DMI access

Besides using known speed optimized coding techniques it is also useful to apply profiling in order to see where simulation time is spend. Classical profiling tools like VTune, valgrind, gprof or gperftools are able to analyse function calls. This type of profiling generated some useful information about bottlenecks. However if context switches between SystemC threads have the biggest impact on simulation speed, the above mentioned profiling tools are not very useful to analyse those effects. For the analysis of SystemC events in a simulation we found the profiling with commercially available event profiling tools very useful. This tool allows you to analyse and visualize when events and context switches happen on which thread. Very often even a quick analysis of event hot spots yielded very useful results. Especially in conjunction with predefined scenarios, where one can form a clear expectation of what events are expected, the profiling results turned out to be very useful.

Even though some aspects of simulation speed optimization are pretty obvious, it makes sense to review typical bottleneck introducing functions, for instance system calls (file IO, inefficient memory allocation, debug messages) and compile settings (all parts of the system shall be compiled with the proper compiler optimizations turned on). In a bigger project where different developers contribute it can be quite tricky to ensure that those optimization aspects are consistently applied. Also the usage of 3rd party IP often requires a careful review which debug messages are being produced as such mechanisms might be different from other models. Good infrastructure as a coherent build system and efficient base classes for logging can help tremendously.

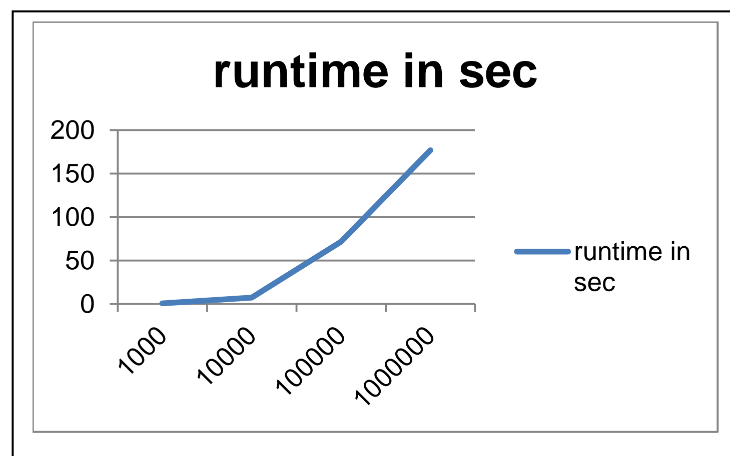

Figure 9: Results for simulation runtime (y-axis) over bandwidth transmitted (x-axis; in transfers per sec)

## 3.2 Simics and SystemC Integration

As in previous sections stated, a VP can be very heterogeneous in terms of building blocks, subsystems, and modelling styles. Indeed, at the level of a platform-SoC VP, this possibility is the

default situation. This heterogeneity in modelling styles is yet another dimension that has an influence in VP-execution-speed. When data transactions go over the boundaries of one domain using one modelling style into a different domain, speed-up techniques as the quantum are not so effective anymore and a performance penalty might occur. Therefore it is crucial to understand better this scenario.

As an example consider the system in Figure 10 below, where a SystemC model for Audio subsystem (LT modelling style) is co-executed with a mobile host platform model inside a Simics simulator. The host platform model includes a CPU model and several peripherals. The Audio subsystem includes one or more DSPs, and local peripherals for storage and transport of audio data.

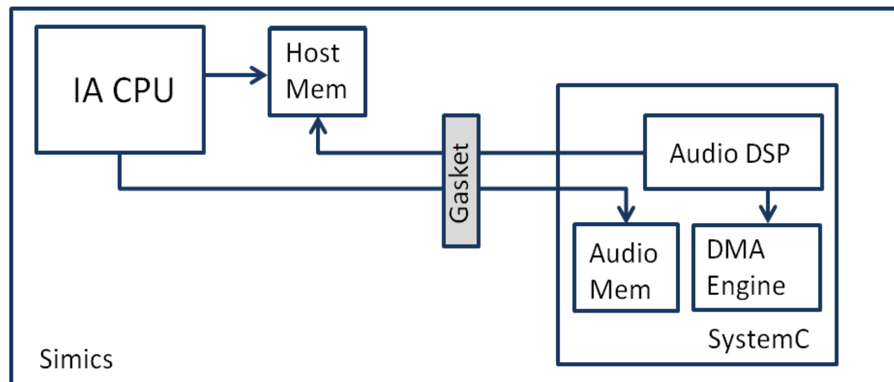


Figure 10: Audio-VP integrated with platform-VP in Simics Simulator

In the next sections we will discuss the system speed optimizations and quantify the above conjectured performance penalties for this case-study.

### 3.2.1 Optimization for speed in Simics and SystemC Integration

The DSP model in the Audio subsystem plays a major role both in functionality and in simulation complexity. Therefore we will deal here in more details with how it handles temporal decoupling. The DSP model is basically an instruction set simulator (ISS) wrapped with SystemC interfaces. It can work either in a cycle-accurate mode or in “fast functional mode” (FFM). FFM is a sort of temporal decoupling mode where the DSP model executes as many instructions (and cycles) as a certain quantum encompasses, without returning simulation control until the end of such quantum, and without considering the delays back-annotated by slaves. Hence it differs from the temporal decoupling as defined in TLM2 standard.

### 3.2.2 Fast access modi

The proprietary DSP TLM defines mechanics for the DSP model to poll slaves about their ability to support some sort of fast access. A slave can support two flavors of fast-access recognized by the DSP model: pure functional calls that by-pass any delay (peek/poke) and Direct Memory Interface (DMI). The DSP model ports are augmented with adaptors converting the DSP specific TLM format accesses into fully compliant TLM2-format accesses, and vice-versa. The adaptor translates peek/pokes accesses into TLM2-*transport_dbg()* calls and translates direct memory accesses into its equivalents in TLM2 standard supported by the TLM2 DMI.

When set to work in FFM the DSP model can prematurely leave the quantum under a number of conditions, most commonly if it performs an access to any slave port where fast access has been denied. Leaving prematurely the quantum jeopardizes the benefits of FFM and therefore all slaves shall accept some form of fast access.

In the case of memory arrays the option-of-choice is using DMI. This provides the DSP model a pointer to the actual software object modelling the memory and enables the first to manipulate the

contents of the later without traversing the bus and routing infrastructure. This modality represents the absolute minimum in computational burden for moving data across the VP. In the case of accessing registers the possibility of generating side-effects as well as other functional artifacts (as the masking of incoming accesses) must be supported. This precludes the DMI option. As a solution, the peek/poke call (converted into TLM2-transport_dbg()) was used instead.

Hence a fraction of the memory space addressable by the DSP model is to be fast-accessed via DMI, others via *transport_dbg()*. Observe that the memory arrays modelled in DML within the host platform also have to support DMI fast-access. This memory space split can be accomplished by the combination of a router model and two differently configured adapters, one for each memory space. The Router contains a detailed description of the memory space that shall be addressed in each modality.

## 3.3 Measurements for Simics SystemC integration

### 3.3.1 Measurement scenarios definition

DML- and SystemC-based models are both co-scheduled within the Simics simulation. Because the timing assumptions of each side are broken when interacting with its counterpart, performance degradation can happen. In this section measurement scenarios are defined to capture and quantify that degradation, as well as the factors that contribute to it. Basically the scenarios are split in a first level into stand-alone and integration scenarios. On a second level there is a further split depending on the traffic patterns exercised within each simulated system constellation.

The comparison of performance for the integrated VP solution against that of stand-alone VPs is of outmost practical interest, since it can determine the weapon-of-choice for use-cases not requiring both Audio and the host platform. Hence, it shall be compared:

- Stand-alone Audio-VP operation against Audio-VP operation when integrated in Simics.
- Stand-alone operation of host platform-VP in Simics against same platform-VP with the Audio SystemC VP attached.

### 3.3.2 List of scenarios

**Scenario A):** Platform-VP stand-alone simulation in Simics where an Intel Architecture (IA) CPU is booting Linux (BusyBox). This scenario aims at providing a reference measurement for a later comparison with the integration scenario.

**Scenario B):** SystemC Audio-VP stand-alone simulation. Splits into B1, B2 and B3 depending on executed Audio DSP FW. B-scenarios also aim at providing a reference for comparison with the integration scenario.

- B.1: The DSP FW test-case (*"DSP Only"*) executes an infinite loop but there is no significant number of accesses to either internal or external memories or registers. This scenario aims at providing a *fastest-possible* performance reference.

- B.2: The DSP FW test-case (*"DSP RegAccess"*) executes an infinite loop accessing a series of Audio-internal control registers. The access rate can be modulated. This scenario aims at measuring performance when transaction traffic is circumscribed to Audio Subsystem.

- B.3: The DSP FW test-case (*"DSP Host"*) executes an infinite loop accessing a memory block external to Audio subsystem. The access rate can be modulated. This scenario aims at measuring performance when transaction traffic travels outside boundaries of Audio Subsystem.

**Scenario C):** Simics simulation with IA-CPU booting Linux (BusyBox) integrated with SystemC Audio. Audio DSP-FW runs *DSP Only.* The comparison point for this scenario is B.1.

**Scenario D):** Simics simulation with IA-CPU booting Linux (BusyBox) integrated with SystemC Audio. Audio DSP-FW runs *DSP RegAccess*, which generates heavy transaction traffic within the Audio Subsystem. The comparison point for this scenario is B.2.

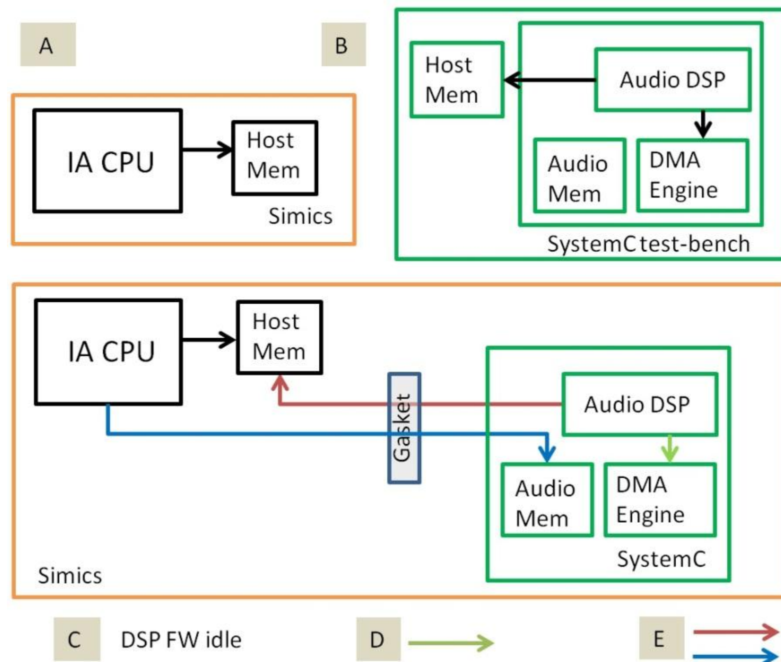


Figure 11: measurement scenarios for stand-alone and integration cases.

**Scenario E.1):** Simics simulation with IA-CPU booting Linux (BusyBox) integrated with SystemC Audio. Audio DSP-FW runs *DSP Host*, which generates heavy transaction traffic through the boundaries of Audio Subsystem.

**Scenario E.2):** Identical to E.1 but in this case IA-CPU additionally accesses Audio Subsystem internal memories.

A common factor in all scenarios involving IA-CPU is that IA-CPU boots Linux (BusyBox). This has the advantage of show-casing performance for a real setup. It has the disadvantage of masking any performance degradation due to Simics-Audio transactions, if it is little. However, since we would be concerned about and analyze the causes for this degradation only if it was large, this is not a concern.

The traffic patterns used for exercising the simulated system constellations span a range from typical to worst-case system utilization. The actual values for this range are derived from architecture specifications describing the communication needs and data transport mechanisms between the host platform and the Audio subsystem, yielding following rates for transactions:

- Incoming into Audio: rate spans $10^3$ to $3*10^3$ transactions per simulated second.
- Outgoing from Audio: rate span from $45*10^3$ to $4.8*10^6$ transactions per simulated second.

### 3.3.3 Results

This section shows the measured results that were achieved for the above described scenarios. The presented Real Time Factor (RTF) values are defined as Wall-clock Time (WT) over Virtual Time (VT). The initialization wall-clock time (construction of SW objects before simulation time 0.00) is discounted when doing the measurement. RTF is measured over a long enough span in order to ensure its statistical representativeness: over whole BusyBox boot (over 8.9 sec. VT), or over 1 sec. VT for Audio standalone.

Figure 12 shows the results for scenario B.3. The diagram plots the measured wall clock time for different simulation times. The different curves show the impact of the quantum setting (q stands for different quantum settings). The interesting effect is that at some point increasing the quantum does not yield any more speed improvement. Since the quantum also introduces other effects, using a value which is large enough to improve performance but also small enough to avoid introducing synchronization issues, is crucial.

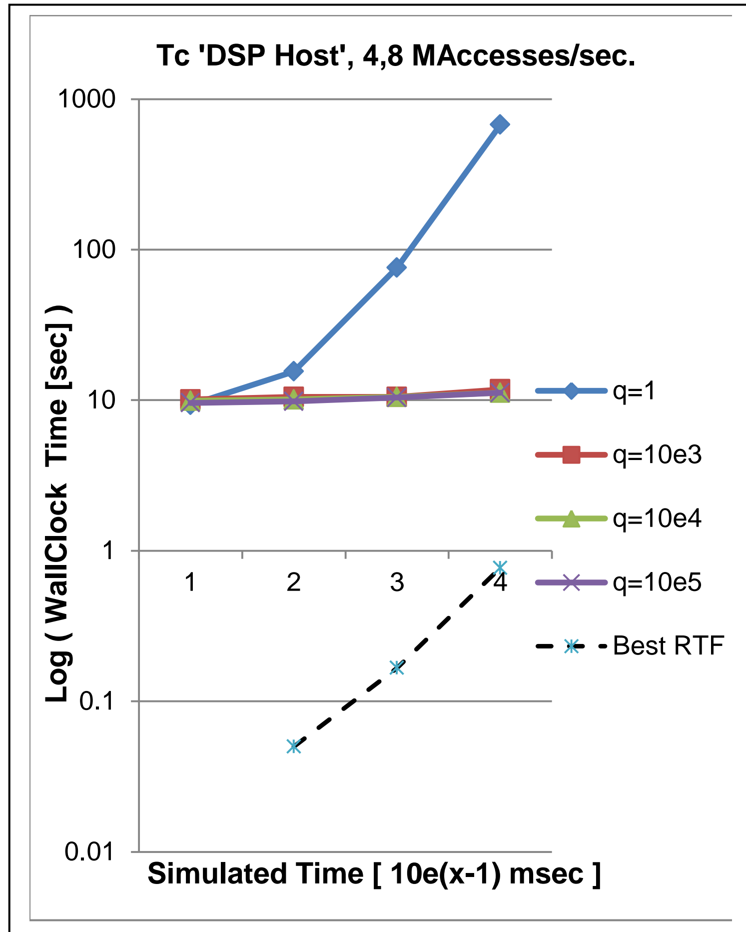


Figure 12: Wall-clock times and RTF values for scenario B.2

Figure 13 shows the results for scenario B.2. The test is focusing on accessing registers within the audio subsystem by its embedded DSP. Similarly to previous one, this figure plots the measured wall clock time of the target system over simulated time, which allows the calculation of the RTF. Results are plotted for quantum values of 1 and $10^5$ (as multiples of 10 ns cycles). The various curves show how the work load variants impact the simulation speed.

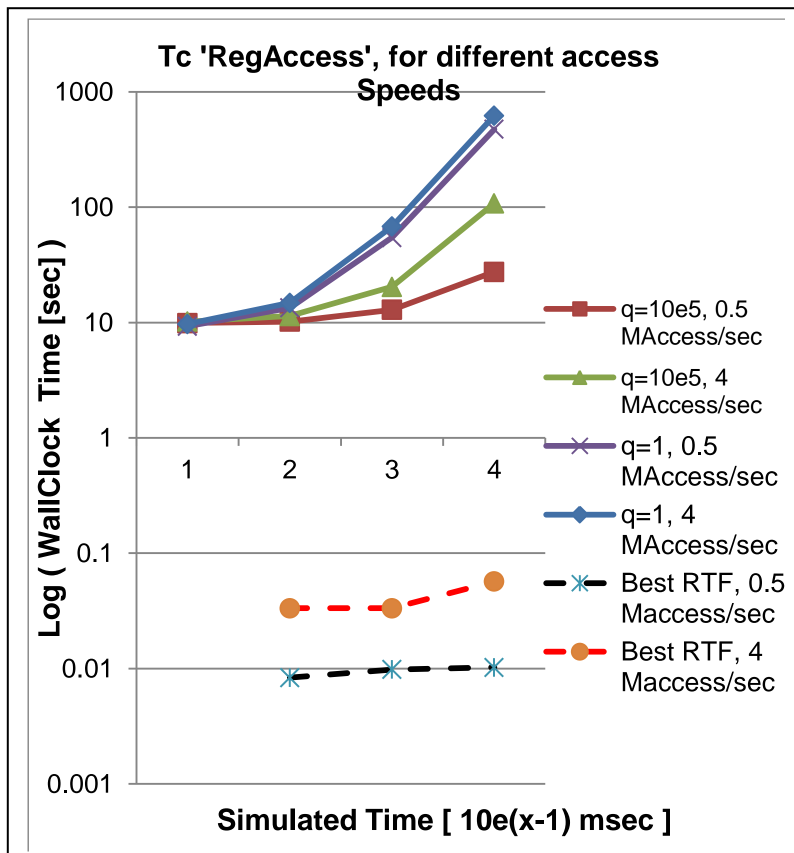


Figure 13: Wall-clock times and RTF values for scenario B.3

Despite the use of quantum, the increase in performance over cycle-accurate operation (q=1) is not as high in this case as it was for scenario B.3. This is due to the fact that in B.2 the accesses are done to registers, where DMI techniques cannot be applied and hence buses and interconnection networks cannot be by-passed, while for B.3 accesses took place over a memory model supporting DMI. This gives a quantitative measure on DMI´s importance.

Table in Figure 14 shows a summary of the average RTF values for the different scenarios (under column “RTF Optimized”). The measured RTFs previous to applying optimization techniques (i.e. for quantum equal to 1) are also exemplary displayed for standalone scenarios B.1 to B.3. Integration scenarios were not simulated for quantum equal to as it was already concluded from standalone scenarios that applying an aggressive quantum became indispensable for VP operation.

| | RTF *(Optimized)* | RTF |
|---|---|---|
| A, Simics standalone, booting BusyBox | 2,7 | - |
| B.1, Audio standalone, number crunching | 1,6 | 415 |
| B.2, Audio standalone, internal-reg. traffic | 17,5 | 613 |
| B.3, Audio standalone, Audio→host-memory traffic ( 4,8 MAccess/sec ) | 1,3 | 667 |
| C, integration, number crunching in Audio | 3 | - |
| D, integration, internal Audio traffic: | 12 | - |
| E.1, integration, Audio→host-memory traffic ( 4,8 MAccess/sec Uplink ) | 3,1 | - |
| E.2, integration, Audio ←→host traffic ( 4,8 MAccess/sec Uplink, 3 KAccess/sec Downlink) | 3,9 | - |

**Figure 14: RTFs summary for the different scenarios**

Finally, the Table in Figure 15 is comparing numbers before and after integrating the audio subsystem. RTF for Simics booting BusyBox worsens from 2,7 (A) to between 3 - 3,9 (C, E.1, E.2) by including the Audio model running traffic-representative FWs. RTF for Audio worsens from 1,3 (B.3) to

| | Audio idle | Traffic over Audio boundaries | |
|---|---|---|---|
| | | Outgoing | Out-/Ingoing |
| Audio SA | - | 1,3 (B.3) | - |
| Simics SA | 2,7 (A) | - | - |
| Integration (w/ Busybox Boot) | 3 (C) | 3,2 (E.1) | 3,9 (E.2) |

**Figure 15: Comparison of stand-alone and integration scenarios**

3,2 (E.1) by including Simics booting BusyBox. Hence it can be concluded that the addition of a complex SystemC model do not hinder significantly the overall simulation performance if this model has been optimized for speed.

### 3.3.4 Results conclusion

Results show how important is the quantum usage when optimizing the design for speed and the impact of supporting DMI-based techniques. This applies for both, standalone as well as integration scenarios in heterogeneous platform-VPs. Once optimized, RTFs in the integration scenario lay in the range of 3 to 12 and the booting of Linux (BusyBox) in integration scenario takes approximately 30-35 seconds wall-clock time for traffic-representative use-cases. These performance figures allow fast turn-around times for SW & driver development.

# 4 Summary

The project has shown that it is possible to integrate SystemC TLM2 based subsystems into a VP model, which models the SoC and runs an OS and FW on top. The resulting model can be used as a software development tool meeting performance and turn-around requirements. It has also been shown that it is necessary to apply different optimization techniques in order to achieve high performance. Specifically minimizing context switches by applying a quantum approach proved to be crucial. The quantum approach also allows performing a trade-off between accuracy and speed. Our project has not tried to apply the same SystemC subsystem in more simulation environments with more accurate timing, for instance for architectural analysis. But in principle this would be possible and can become a topic for future work.

# 5 References


- Wind River Simics; Model Builder User Guide, version 4.8
- IEEE Standard for Standard SystemC Language Reference Manual.
- IEEE Std 1666-2011. Jan. 2012, ISBN 978-0-7381-6802-9.