virtutech

TRANSACTION-LEVEL MODELING IN SIMICS

AUGUST 2009

JAKOB ENGBLOM

WWW.VIRTUTECH.COM

INTRODUCTION

Virtualized Systems Development™ (VSD) is a development methodology where the actual hardware of a system is replaced with a virtual platform running on a workstation or PC. The virtual platform can run the same binary software as the physical hardware and is fast enough to be used as an alternative to physical hardware for software development. The virtual platform should be *full-system*, i.e., contain the whole target system, including multiple chips, multiple boards, and multiple networks.

The virtual platform provides additional benefits to the user compared to physical hardware. For example, the virtual platform offers superior convenience and stability, full insight into the system execution, and better debugging facilities. It provides access to the target system long before prototype hardware is available. This enables software development to start early. The virtual platform supports fault injection and testing with multiple configurations. It also provides checkpoint and restart facilities, which is the focus of this white paper.

Transaction-level modeling (TLM) is an enabling technology for virtual systems development and virtual platforms. Transaction-level modeling raises the level of abstraction of hardware modeling for design, verification, and software development. Transaction-level models run faster and take less time to create than cycle-driven or RTL-level models.

TLM offers the best way to create models which are complete enough to run real software and still abstract enough to run fast. The addition of powerful TLM support to the SystemC™ design language in the form of the TLM-2.0 standard in 2008 was a milestone in the adoption of TLM in the EDA industry. Transaction-level modeling at core is a design principle that can be applied in any language and almost any simulation environment. For example, TLM models in SystemVerilog™ are used in hardware verification, and TLM programmed using C and C++ has been used to build virtual platforms in both commercial and open-source settings. TLM in networks is used in the ns2 simulation tool.

To enable full-system virtual platforms and the VSD methodology, transaction-based modeling needs to be applied everywhere in the system model, with careful design to retain speed and ease of model programming. This white paper will explain how transactions are used throughout Simics to provide fast simulation of full systems, with an easy and efficient programming model.

TRANSACTION-LEVEL MODELING

The key idea behind TLM is to communicate as seldom as possible in as large units as possible. Compared to how digital hardware actually works, it means abstracting from clocks, signal levels, pins, and signal lines. Rather than simulating the clocking and signals on buses, we just complete each logical *transaction* as a single unit, typically implemented as a *single function call* from one module of a simulator to another.

The key properties of TLM are that:

- Communication is accomplished by function calls directly between simulation modules, without involving any simulation kernel.
- Communication is performed in as large units as possible, to reduce the number of communications calls.
- Communication is performed as seldom as possible, typically only when something interesting happens that requires action on the part of the receiver of the transaction.

Figure 1. Principle of TLM

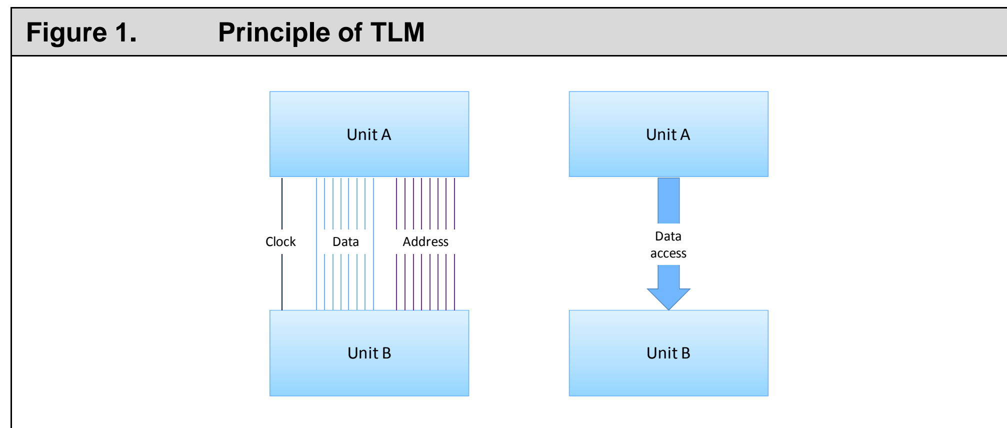


Figure 1 shows an example of a memory bus between two units. In a non-TLM model (left side), you have a clock and separate address and data lines. To do a read from unit B, unit A will set the address lines to represent some address, toggle the clock for the appropriate number of cycles, and then expect that unit B has deposited the data on the data lines and read the data. This is how hardware works. However, in TLM, we focus on the essential operation, and that in this case is a single "read", which we can implement as a single function call from unit A to unit B. The model of unit B immediately returns the data requested to unit A, with no clock pulsing or modeling of

data and address lines. This simplification makes the simulation run many orders of magnitude faster, at the cost of some accuracy in the modeling of the physical hardware timing.

To recap, TLM means using direct *function calls* between simulated units as the communications mechanism in a computer simulation. In general, there is no need to construct special transaction data structures to do transaction-level modeling. If the information to be communicated is simple, such as raising an interrupt, all that is needed is a simple function call parameter. This reduces the memory and time overhead of TLM significantly.

Another subtle by important aspect of TLM is that transactions only communicate *interesting changes* in the simulated system. In a clock-driven system, simulation units often report their state on every cycle, regardless of whether it has changed or not. Another common mechanism in detailed modeling is the signal line that only reports activity to its recipients when it changes value (for example, the `sc_signal` functionality in SystemC). In this way, TLM reduces the amount of activity in the simulation to a minimum.

Discussions about TLM tend to focus on the memory-mapped interface of a model and memory transactions. In particular, TLM is often associated with the interface between a processor and the devices and buses in the system. To create a truly transaction-based system simulation, you need to apply TLM to *all* interfaces between simulation models (and not just the memory bus). As illustrated in Figure 2, in addition to the memory bus facing the processor, that also includes direct interfaces to other devices, interrupts, and backends like user interfaces and network simulations. Note that the interface to the simulation kernel is not affected by the TLM discussion, but it is included in Figure 2 for completeness.

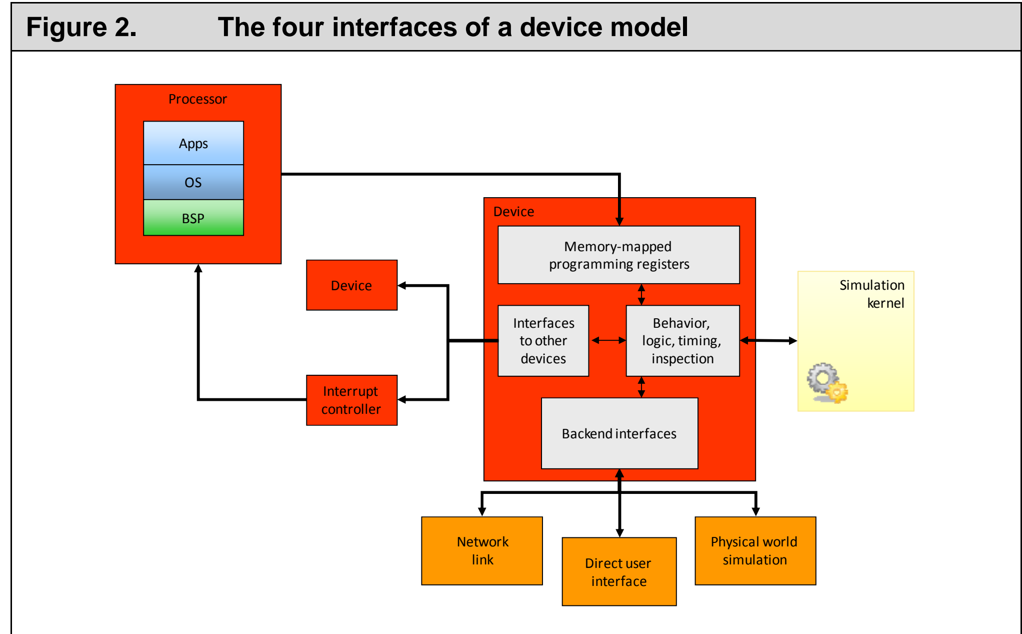


Figure 2. The four interfaces of a device model

Virtutech Simics® has been built on the assumption that all communications between units is performed using TLM, and that nothing is communicated using a clock-driven system or via the Simics kernel.

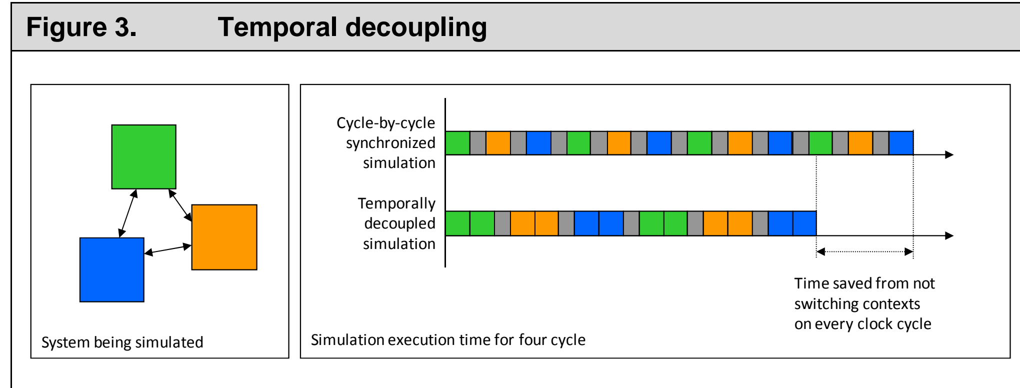
Related Performance Enhancements

There are some other important performance enhancements that are natural to use with TLM. The two most important techniques, used universally in fast virtual platforms, are:

- Direct-memory interfacing (DMI)
- Temporal decoupling

DMI is the optimization of accesses to *target* system memory by not actually doing a transaction for each memory operation. Devices and models of target memory in the virtual platform provide processors (and other active initiators of memory accesses) with pointers to the models' internal representations of target memory. Using such direct pointers makes a target memory access much cheaper, as it should reduce to a single host memory access rather than a full function call to perform a transaction.

Figure 3. Temporal decoupling



Temporal decoupling is a way to reduce the simulator kernel and context-switching overhead by not synchronizing the time in different simulation units on every target system clock cycle. As illustrated in Figure 3, this means letting each unit of the simulation run a time slice containing several cycles before switching to the next unit. The time in the simulation units will thus be synchronized after each time slice, rather than on each cycle, but this is usually not a problem in practice.

Note that in Simics, temporal decoupling is applied between processors. The devices that belong to the time domain of a specific processor communicate directly with each other and the processor, without any delay. This means that in Simics, interrupts from devices reach the processor immediately, and that devices need to not concern themselves with temporal decoupling.

TLM ABSTRACTION LEVELS

Abstraction levels is a contentious topic to discuss, as different industries and communities have very different ideas of what “abstraction” really means and how to sort it into levels (not to mention which abstraction levels make sense for which use cases). However, in the field of virtual platforms, there appears to be a few abstraction levels that recur across simulation kernels and user models.

From a virtual platform perspective, the main axis of abstraction is timing detail and timing fidelity to the eventual physical hardware. Once the ambition in timing detail is decided, many aspects of the simulation implementation follow naturally.

In the table in Figure 4, we summarize the main timing abstraction levels that we have seen applied in practice, and what they imply in terms of what can

be observed for the two most important interfaces: memory buses and networks.

Figure 4. Generalized Abstraction Levels

Level	Memory-mapped bus	Network
Zero local delay TLM	No delays on memory accesses, no arbitration of accesses, data immediately available from transaction target, memory accesses driven by initiator	Complete delivery of packets in one step, no model of contention, no model of bandwidth, no model of CAN priorities
Local delay TLM	Delays on memory accesses, no arbitration, data immediately available from target, memory accesses driven by initiator	Media busy times can be modelled by the delay.
Phase-by-phase TLM	Phase-by-phase bidirectional exchange of transactions, arbitration modelled, data can take time to become available, tight timing synchronization between simulation units	Bandwidth limitations, parallel vs serial buses
Clocks and pins	Waveforms on signal lines, driven clock by clock, no direct transactions, tight timing synchronization	CAN priority, Ethernet CSMA effects, data waveforms, individual bit errors, timing of each bit of information
Physical simulation	Analog effects of traces, impedance, and other electrical effects	Noise on the line, radio network interference, signal degradation

Zero Local Delay

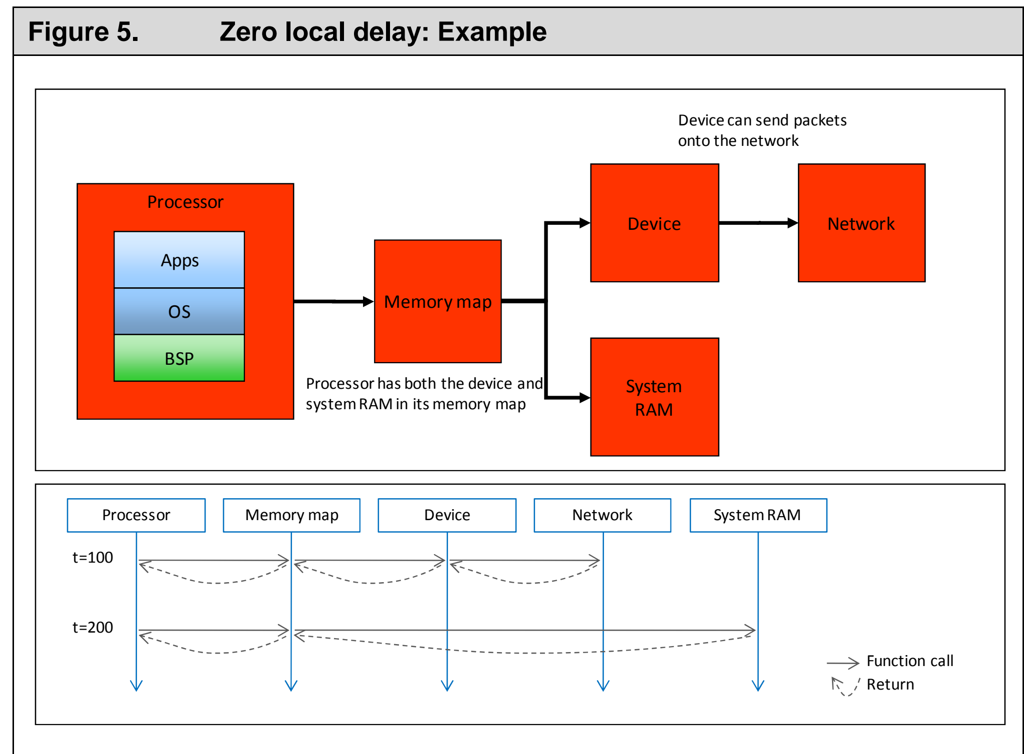
Zero local delay TLM is the dominant model for building fast virtual platforms for systems and software development. Such solutions do not try to provide details on timing properties and bandwidth usage, as that is not needed for most software work. DMI and temporal decoupling are always used, since performance on software workloads is the main target.

In zero local delay TLM, system timing tends to be driven by a small number of units, typically the processors running software. All other devices are passive and use event mechanisms and callbacks to perform periodic work. A transaction function call from an initiator results in a set of nested function calls that all take place at the same moment in virtual time, and time does not progress until all functions have returned to the initiator.

Figure 5 shows a simple example system of how zero local delay works. At time 100, the processor sends a write to the device to make it send a packet to the network. This is implemented as three function calls, all of which happen at time 100. At time 200, a memory access (not using DMI) is performed, which goes directly to memory and back.

Zero local delay makes eminent sense for functional network simulation: each network packet (Ethernet packet, ATM cell, serial character, SATA command, etc.) is delivered instantaneously to the network simulation.

Figure 5. Zero local delay: Example



The advantage of zero local time in practice is that it makes it possible to optimize both processor simulators and simulation infrastructure for the simplest possible case. The fact that time only progresses between transactions, not during transactions, makes it possible to reduce the amount of book-keeping in the simulation kernel. Device models are simple regular programming-language objects with methods and local variables, and do not need to be driven by a threading package.

Zero local time also facilitates checkpointing, since the system has many fewer active components that need special treatment to store their state in a checkpoint[1].

Zero local time does not mean that the system as a whole is untimed, though. Between each instruction (or other step in other initiators), time passes. All devices in the system have the option of posting timed events to the simulator kernel. This is used to implement timers, delays from operations to completion interrupts, and similar system-level timing effects. Without this, software would not work.

This level of abstraction is also known *Software Timing,* or *ST*. ST represents the most abstract system model that is able to run actual target software.

Pure-bred ST TLM is the design basis for solutions like VmWare, VirtualBox, Qemu, IBM CECSim, IBM Mambo, sdv, SimOS, and Simics.

Zero local time can be implemented to some extent within the SystemC TLM-2.0 LT (loosely timed) coding style. Since SystemC TLM-2.0 intentionally defines *coding styles* and not levels of *abstraction*, we do not use ST and LT in Figure 4. Implementing zero local time in a simulation system based on SystemC TLM-2.0 LT would only use a subset of the functionality available. It requires using a set of instruction-set simulators drive the simulation, and making sure these are the only active units in the system. All devices should be implemented using `SC_METHOD`. No b`_transport()` function in a device should add any delay to transaction calls, and no model code should ever call `wait()`. In essence, transaction transport calls are blocking calls that must not block, since SystemC lacks the concept of a synchronous call. All delayed actions in a device have to be implemented using `sc_event`, and not wait(). Conceivably, a SystemC kernel implementation could be optimized to only support zero local delay, but that would be breaking the SystemC language specification. Thus, simulators assuming ST models tend to be faster than those allowing SystemC LT, as LT allows a richer set of behaviors with more performance risks associated.

---

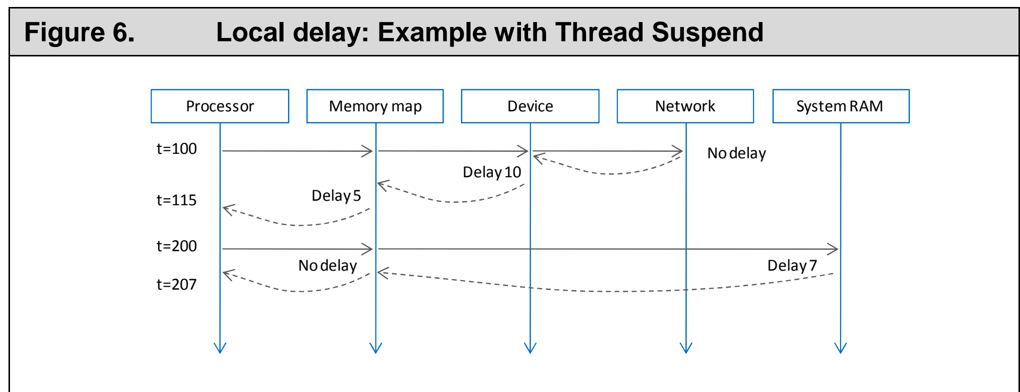
[1] For more on checkpointing, see the Virtutech whitepaper on Simics Checkpointing at http://www.virtutech.com/whitepapers/simics_checkpointing.html.

Local Delay

Local delay TLM is used when users want to account roughly for the time taken to access devices across a bus, model caches and their execution time penalties, and the difference between different memory sections in a system. Just like in zero local delay, the data from an operation is presented immediately when the transaction function call returns to the initiator. In network simulation, local delays can be used to account for busy periods on the network, providing rough contention and bandwidth modeling.

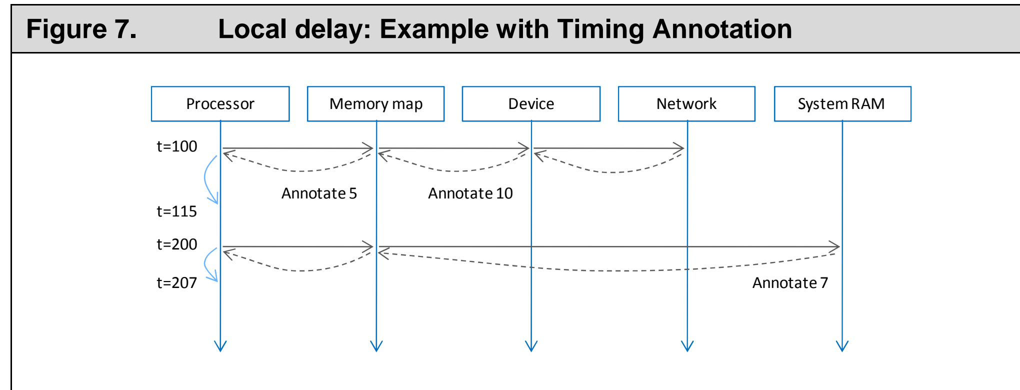
From the simulator implementation perspective, this means that the simulation infrastructure needs to handle the case that time passes as a device is processing a transaction. One solution to time handling is to let all units in a simulation be active threads, and use some form of thread suspend or wait operation to let time pass. Figure 6 shows how time passes in a threaded model.

Figure 6. Local delay: Example with Thread Suspend



Needless to say, using frequent thread suspend and resume causes simulation overhead to increase dramatically, and reduces the effectiveness of temporal decoupling. Therefore, many simulators (including SystemC TLM-2.0) implement a timing annotation scheme as illustrated in Figure 7. Here, simulation units return immediately from transaction calls, and provide the time taken to complete an operation as an annotation to the call. The initiator is then expected to account for the time accumulated in some appropriate way.

Figure 7. Local delay: Example with Timing Annotation



Timing annotations are supported by many simulators built for the ST abstraction level. For example, Mambo and Simics both support local delays in order to account for caches and memory access times.

Models using local delays tend to be more complex than zero local delay models, as they have to account for the case that transactions enter while a device is nominally busy.

If local delay TLM is used to model contention points and bottlenecks in a system, temporal decoupling cannot be used very aggressively. To give reasonable results, the temporal decoupling time slice cannot be bigger than the shortest modeled latency in the system.

Local delay TLM (using either threads or annotations) is the natural abstraction level for the SystemC TLM-2.0 LT coding style. SystemC TLM-2.0 recommends using timing annotations with LT in order to increase simulation performance and allow temporal decoupling to be used. Still, supporting local delays presents some performance implications, as simulators cannot use all the techniques found in those assuming zero local delays.

Simulation solutions supporting local delay in SystemC include the OSCI SystemC kernel, CoWare, VaST, and Synopsys Virtio tools.

Note that the term “LT” is often used to denote local delay TLM, even though it strictly speaking is a coding style and not a level of abstraction.

Phase-by-phase TLM

Phase-by-phase TLM adds more detail to the models of interconnects by supporting the explicit modeling of bus phases and the bus state machine.

Typically, a memory bus request would go through the phases of bus arbitration, request, and response. As a result, simulation units need to contain protocol state machines, and the definition of the bus protocol in terms of phases and phase transitions is a very important part of the simulator design. Phase-by-phase TLM allows the modeling of cache coherence protocols and memory consistency models. Data is delivered using a backwards call rather than as part of the return value from a transaction function call.

Phase-by-phase TLM is often used along with cycle-level details in the simulation units (such as "cycle accurate" processor models). It is most appropriate for high-performance memory buses, as these tend to have the most complex bus protocols. For networks, it seems more natural to use cycle-based simulations to capture their detailed behavior. Short-distance interconnects like PCI Express, RapidIO, and HyperTransport are quite similar to memory buses in their design, and can be modeled in detail using phase-by-phase TLM.

In phase-by-phase TLM, simulation units need to be tightly synchronized, since all units need to be able to get onto a bus virtually simultaneously to correctly account for arbitration effects and overlapping requests and responses.

Phase-by-phase TLM is the target for the SystemC TLM-2.0 AT (approximately timed) coding style.

Clock-Cycle and Physical Simulations

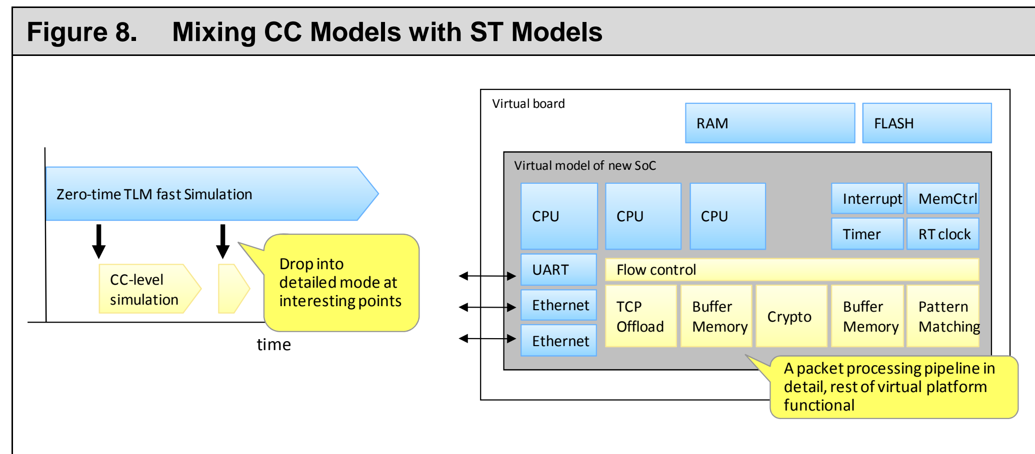
Figure 4 also lists clock-cycle (CC) and physical models. These are not transaction-level models, but rather the types of models that TLM was designed to run faster than. However, CC models are still very important when you really want to simulate exactly how a piece of hardware works. By driving models from a clock, it is possible to create very accurate models of the hardware.

Such models run between four and five orders of magnitude slower than the zero-time TLM models, and take one or two orders of magnitude more time to create (note that CC models can sometimes be automatically generated form RTL, which makes creation times fairly short).

Physical simulations sometimes move from the clocked domain of CC models into real continuous time, or they can be driven by some kind of

clock as well. In any case, they are even slower to execute and take more time to create and validate.

Figure 8. Mixing CC Models with ST Models



In Simics, CC models can be integrated with ST models using the Simics *Hybrid* system. Simulation speed will be dominated by the slow CC models, but "uninteresting" parts of the system can be modeled at the ST level. It is also possible to use the ST models to position a workload for the CC models, dropping into CC mode at interesting points in the workload mainly being simulated using an ST-level simulator. Both these variants are shown in Figure 8, using a hypothetical networking SoC as an example.

SIMICS TLM IN DETAIL

As discussed above, the Simics native level of abstraction is *software timing* (ST), or zero local delay TLM. ST TLM is applied consistently in all parts of a Simics simulation system, creating a simulation solution that is fast enough to truly enable full-system virtual platforms.

In Simics, processor cores are treated specially as drivers of the simulation. All other devices are considered passive devices that only use transactions and events to drive behavior. With this small set of active objects, Simics can apply very aggressive temporal decoupling for higher performance. Note that any simulation model can be a "processor" in Simics and drive the simulation, but generally it only makes sense to do so for actual processor cores. Processor simulation tends to consume 95% to 99% of the system simulation time, so it makes sense to spend significant effort in building processor simulators.

When device models in Simics communicate with each other, they use direct unidirectional function calls. The function calls are logically grouped into *interfaces*, which are named and registered with the Simics kernel. Any device can call any other device on any interface. The device only needs to know the name of the other device and the name of the interface. If an interface in a model requires two devices to exchange transactions, and not just one device calling another device, one unidirectional interface is used for each direction. Any model developer can define new interfaces in Simics, and use them just like the interfaces provided by Virtutech.

All calls over a Simics interface are synchronous and return immediately in virtual time. There is no option to block, wait, or pass time in any other way inside an interface call. If a model needs to perform some work at a later point in time, it posts an event to the Simics kernel and gets a callback at the time it requested. This is the essence of ST modeling, and crucially simplifies modeling and enables optimizations in the processor models.

Since Simics applies temporal decoupling between sets of devices attached to a common clock source (typically, a processor core), most activities in a device do not need to concern themselves about time. For most devices, the time of each transaction arriving will be greater than or equal to the previous transaction. This greatly simplifies most device modeling.

Most interfaces in Simics are logically unidirectional. For example, a memory transaction is a pure forward-only operation from a processor (or other device), through one or more memory spaces, to an end-point device. The end-point devices usually do not know or care about the path over which a transaction arrived. All it knows is that in some way, a memory access hit it in a particular offset in a particular register bank (to help resolve some cases where it is useful to know the source of the memory transaction, each transaction carries a pointer to the initiating object). The device model will compute the result of the access and return immediately. Another example is a keyboard interface: a keyboard model will call a keyboard controller model with a stream of characters being virtually typed, but it will never get anything back from the keyboard controller.

Memory operations are just one interface among many. There is nothing particularly special about them, even though they are the most heavily used interface for device modeling as all devices need to decode memory-mapped accesses.

Some interfaces are logically bidirectional. Networks are typical examples: an Ethernet controller can both send and receive frames. In Simics, this is implemented as two unidirectional interfaces going in opposite directions between two devices. The interfaces can be symmetric or asymmetric, depending on what the most practical design is for each case.

For example, a model of a broadcast network would have an interface from devices to the broadcast network where the devices register themselves with names and hardware addresses. The interface from the broadcast network to the devices would be simpler, just delivering packets. On the other hand, a model for passing traffic over a point-to-point link with the same type of device at each end is naturally symmetric, since both sides are equals in the communication.

Other examples of TLM interfaces in Simics are interrupts and resets, where a device just performs a single function call with no special transaction data structure to cause an interrupt to happen. DMA accesses are performed in a single step, rather than modeling the word-by-word movement of the hardware.

For more insight into how Simics uses TLM and examples of modeling patterns, please see the System Modeling with Simics white paper at `http://www.virtutech.com/whitepapers/modeling.html`.

Serial Device Example

Figure 9 shows the `serial_device` interface used in Simics between a serial device (such as a UART or console), and the link object representing the cable or connection between them. The same interface is used in both directions. Since Simics implements an object-orientated structure in C, all calls carry an “`obj`” parameter which is the identity of the receiving object. In DML and C++, this can be hidden. “`conf_object_t`” is the base type of all Simics simulation objects.

Figure 9. Simics Serial Link Interface

```
#define TTY_ABORT   0x100

typedef struct serial_device_interface {
    int (*write)(conf_object_t *obj, int value);
    void (*receive_ready)(conf_object_t *obj);
} serial_device_interface_t;
```

There is no special support for connecting devices in this interface. It assumes that some higher-level configuration mechanism (usually, the Simics hierarchical components system) takes care of ensuring that both ends of the connection know about each other. Thus, the interface becomes very simple. It is just a matter of sending a character to a recipient by calling `write()` on the receiving device. To support out-of-band values, the data is sent as an `int` rather than as a `char`. There is no special transaction structure needed. The `write()` call will return either one or zero, with zero indicating that the receiving device did not accept the character sent. In this case, the receiving device will at some later point call `receive_ready()` in the originating device to tell it is possible to send characters again.

Thus, these two simple function calls on each side of the connection suffice to implement a flow-controlled sequence of character sends between two serial devices.

PCI Example

PCI is a fairly complex interconnect, containing several different memory mappings as well as side-band data such as interrupts. In Simics, after a PCI mapping has been setup, processors and devices perform direct memory transactions into the mapped memory. This covers most PCI operations. For other PCI-related operations, PCI devices implement a second interface shown in Figure 10.

Figure 10. Simics PCI Device Interface

```
typedef struct pci_device_interface {
    void (*bus_reset)(conf_object_t *obj);
    int (*interrupt_acknowledge)(conf_object_t *obj);
    void (*special_cycle)(conf_object_t *obj, uint32 value);
    void (*system_error)(conf_object_t *obj);
    void (*interrupt_raised)(conf_object_t *obj, int pin);
    void (*interrupt_lowered)(conf_object_t *obj, int pin);
} pci_device_interface_t;
```

This interface lets the PCI controller send PCI-defined events such as bus resets and system errors to devices. It also handles the sending of interrupts between devices (`interrupt_raised()` and `interrupt_lowered()`), and the PCI bus interrupt acknowledge cycle. This interface and the memory-mapped interface are sufficient to implement PCI devices in Simics.

SUMMARY

Simics is a full-system virtual platform that can simulate very large target systems with high speed thanks to pervasive and aggressive use of transaction-level modeling (TLM). Simics provides TLM interfaces for all communications in a target system, including memory accesses, interrupts, and disk and network input and output. Within the design space offered by TLM for memory transactions, Simics uses a level of abstraction, *Software Timing* (ST), which is more efficient than loosely-timed (LT).

CONTACT INFORMATION

North and South America sales_americas@virtutech.com	**Europe, Middle-East, Africa** sales_emea@virtutech.com
Asia-Pacific sales_apac@virtutech.com	**Japan** sales_japan@virtutech.com
http://www.virtutech.com	

© Copyright 2009 Virtutech, Inc. All Rights Reserved.

TRADEMARKS. Virtutech, Simics, Hindsight, and the logos thereof, are trademarks or registered trademarks of Virtutech, Inc. and/or its subsidiaries, in the United States and/or other countries.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. VIRTUTECH MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

Virtutech, Inc., 2001 Gateway Place, Suite 201E, San Jose, CA 95110, USA