System Simulation with gem5 and SystemC

The Keystone for Full Interoperability

Christian Menard, Jeronimo Castrillon
Technische Universität Dresden
Dresden, Germany
Email: christian.menard@tu-dresden.de
jeronimo.castrillon@tu-dresden.de

Matthias Jung
Fraunhofer IESE
Kaiserslautern, Germany
Email: matthias.jung@iese.fraunhofer.de

Norbert Wehn
University of Kaiserslautern
Kaiserslautern, Germany
Email: wehn@eit.uni-kl.de

***Abstract*—SystemC TLM based virtual prototypes have become the main tool in industry and research for concurrent hardware and software development, as well as hardware design space exploration. However, there exists a lack of accurate, free, changeable and realistic SystemC models of modern CPUs. Therefore, many researchers use the cycle accurate open source system simulator gem5, which has been developed in parallel to the SystemC standard. In this paper we present a coupling of gem5 with SystemC that offers full interoperability between both simulation frameworks, and therefore enables a huge set of possibilities for system level design space exploration. Furthermore, we show that the coupling itself only induces a relatively small overhead to the total execution time of the simulation.**

I. Introduction

Today's companies have to deal with complex hardware architectures such as multi-cores, sophisticated interconnects and memory systems. *Virtual Prototypes* (VPs) are widely used to allow for early design space exploration and to decrease the *Time-to-Market* (TTM), costs, and efforts by developing software and hardware concurrently. They are high-speed, fully functional software models of physical hardware systems that can simulate the exact behavior of real hardware. With their help, complete *Multi-Processor System-on-Chips* (MPSoCs) can be simulated with reasonable simulation speed and visibility and controllability over the entire system. Case studies have shown that by employing VPs it is possible to deliver more competitive products up to six months earlier [1].

In recent years, the SystemC TLM2.0 IEEE1666 standard [2] has become the main developing tool for VPs in industry and research. It allows to quickly simulate HW and SW systems on different levels of abstraction in order to estimate and optimize the performance and power for different applications. In contrast to pin accurate models, *Transaction Level Modeling* (TLM) abstracts the communication mechanisms from the actual hardware. It encapsulates communication between interacting components in so-called transactions, which are transferred by function calls.

The industry offers several SystemC based CPU core models that provide a trade-off between simulation speed and accuracy. For instance, the *FastModels*, distributed by ARM Ldt. or the OVP [3] models use *just-in-time-compilation* for code execution and model communication using the TLM *loosely-timed* coding style. However, loosely-timed models do not reflect a realistic timing behavior and thus can mainly be used for software development and high level explorations. Cycle

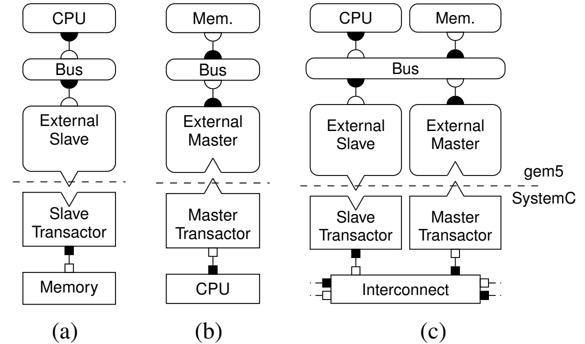


Figure 1: Possible scenarios for binding gem5 and SystemC:

accurate simulations can be performed with the commercially available Carbon models from ARM or the Aurix [4] TLM models from Infineon. However, these models are shipped as binary libraries, which makes them useless for micro-architectural research due to their inflexibility (i.e. they cannot be modified). Furthermore, they are slow and can, therefore, not be employed for fast design space exploration.

In contrast to industry, the academia lacks free, accurate and realistic SystemC models of modern CPUs. The most mature cycle accurate open source system simulator is the *gem5* framework, which is a modular platform for computer-system architecture research [5]. It is not only widely used in academia, but also the industry employs gem5 for research. For instance, ARM and AMD use gem5 internally for design space exploration and actively contribute to the open source project. However, gem5 is not implemented in SystemC as its development started before the IEEE ratified the official SystemC and TLM standard in 2005 [2]. Since this time, both frameworks, gem5 and SystemC have evolved extensively and independent in parallel. Therefore, gem5 is incompatible to TLM models that exist in industry and academia.

In this paper, we present for the first time a comprehensive coupling between SystemC and gem5 that provides full interoperability. Through this coupling, any SystemC module that implements the TLM base protocol can be connected to any gem5 module, as shown in Figure 1. To the best of our knowledge, there exists no reference which describes syntax and semantics of both frameworks and how both simulation kernels can be coupled in order to enable full interoperability,

low overhead and therefore high simulation speed. Furthermore, we demonstrate the coupling for two complex MPSoC example platforms and evaluate the simulation performance. With our use-case, we show that only 7.6% of simulation speed is sacrificed for the coupling. The source code of the developed code has been committed to the official gem5 repository.

The coupling enables research and industry to enjoy the benefits of both frameworks and allows for the reuse of legacy models developed in the last decade. The development of and research on SystemC based memory, interconnect, and peripheral models benefits from the realistic simulation of workloads provided by gem5's detailed and freely available CPU models. The architectural and micro-architectural research in gem5 also benefits from the possibility to integrate modern and accurate third-party models of various system components. As it is not possible to simulate a system with heterogeneous *Instruction Set Architectures* (ISAs) in gem5, the coupling also provides a way to open gem5 for simulation of heterogeneous platforms.

The rest of this paper is structured as follows: Section II discusses related work. In Section III, we introduce TLM and gem5 along with their communication protocols. The co-simulation coupling and synchronization is explained in Section IV. We further evaluate the performance degradation of this coupling for two representative virtual platforms in Section V. Finally, Section VI concludes the paper.

II. Related Work

In the past, there where several attempts of coupling the gem5 and SystemC simulation kernels. Mingyan Yu et al. discussed a fast synchronization scheme [6]. Texas Instruments and GreenSocs proposed a coupling [7] and provided a patch that was not accepted in the gem5 main repository. Finally, Andrew Bardsley from ARM implemented a coupling that became part of the gem5 main repository in June 2014. However, while these couplings allow for co-simulation, they don't provide mechanisms for communication between gem5 and SystemC modules which limits their usefulness. Based on Andrew Bardsley's coupling, Matthias Jung presented a first interoperable coupling in a talk in the gem5 Workshop at ISCA 2015 [8] and laid the foundation for this work. At this time the coupling only supported scenario (a) from Figure 1.

Other approaches to simulator coupling include indirect couplings by means of traces as presented in [9] and [10]. Furthermore, simulation middelwares like SST [11] or FERAL [12] can be used to couple gem5 with SystemC. However, a middleware reduces the simulation performance compared to a fully customized solution.

III. Background

This section provides an overview of gem5 and SystemC, with focus on the concept behind transaction modeling.

A. SystemC & TLM

SystemC is a modeling library for systems containing hardware as well as software components. It is maintained by Accellera[1] and was ratified as IEEE 1666 standard in 2005 [2]. Although it is possible to model components on the *Register Transfer Level* (RTL), SystemC is mainly used for high–level system modeling, which leads to shorter simulation time compared to other *Hardware Description Languages* (HDLs) like VHDL or Verilog. SystemC extends the programming language C++ with classes and macros to provide an effective event-driven simulation kernel. These extensions provide the principles of parallelism and synchronization [13].

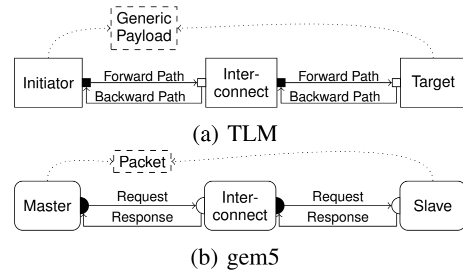


Figure 2: TLM and gem5 both simulate memory accesses by passing a reference to a transaction object between modules using a series of function calls.

Transaction Level Modelling (TLM) is used to model the communication between SystemC components by function calls. The emphasis is more on the functionality of the data transfers and less on their actual implementation. Instead of modeling data transfers between different modules pin- and cycle accurate, function calls referred to as transactions are used. Whole blocks of data are transferred as a reference, which leads to a significant simulation time speedup [14].

In TLM, modules communicate with each other through *sockets*. *Initiator* sockets start new transactions and *target* sockets respond to transactions. Initiator modules (e.g., processors) have one or more initiator sockets and target modules (e.g., memories) have one or more target sockets. Interconnect modules (e.g., buses) use both socket types. TLM transactions are encapsulated in the so-called *Generic Payload* which contains address, command, status, and other information along with the actual data. In order to simulate a memory access, modules use a series of function calls and pass references to a generic payload object as well as the timing annotations to each other. Figure 2a shows a high-level overview of communication between modules using TLM.

The TLM standard defines four transport interfaces: *blocking*, *non-blocking*, *debug*, and *direct memory interface (DMI)* [15]. The blocking interface is associated with the so-called *loosely-timed* coding style. A complete transaction is performed through a single function call modeling only the timing points at the start and end of the transaction. The blocking interface allows fast execution of transactions but can only provide limited timing accuracy.

DMI is typically used for functional simulation as this interface significantly reduces the overhead for simulating transactions. With DMI, a target can grant an initiator direct

[1] www.accellera.org

access to a memory area by passing an access pointer. The initiator can perform memory access without initiating any transactions, which drastically increases the simulation speed. However, DMI completely disregards the timing of interconnects and memories.

The debug transport interface is similar to the blocking transport interface in the sense that it performs a complete transaction with one function call. However, debug accesses do not account for timing and don not have any side-effects. This makes the debug transport interface suitable for initialization and debugging during simulation.

The non-blocking transport interface is associated with the so-called *approximately-timed* coding style. It is used to model a sequence of interactions between the initiator and the target during the course of a single transaction. This provides more accuracy in modeling the timing of memory accesses. For simple memory-mapped bus models, the TLM standard defines a base protocol consisting of four phases. However, the base protocol can be extended to accurately model more complex bus protocols (e.g. as shown in [16]).

The TLM non-blocking base protocol consists of the following phases: `BEGIN_REQ`, `END_REQ`, `BEGIN_RESP` and `END_RESP`. Two functions handle the communication. On the forward path, `nb_transport_fw()` transfers a generic payload object from an initiator to a target. On the backward path, `nb_transport_bw()` transfers a generic payload object from a target to an initiator. Both functions receive a phase and a timing annotation as additional arguments. The return value indicates whether the function return corresponds to a transition in the protocol. Returning `TLM_ACCEPTED` indicates that there is no phase transition and the next transition will be performed by a new function call. `TLM_UPDATED` or `TLM_COMPLETED` indicate that there was a phase transition to any following phase or to the last phase, respectively.

Figure 3 depicts the sequence of function calls required for a complete transaction in the TLM non-blocking base protocol. Note that between each function call the modules may advance in simulation time to model delays. The initiator starts a transaction by calling `nb_transport_fw()` passing the `BEGIN_REQ` phase. The target acknowledges the request by calling `nb_transport_bw()` and advancing to the `END_REQ` phase. The initiator must not send another `BEGIN_REQ` signal until it receives an `END_REQ` signal for the previous request. This is referred to as the *request exclusion rule* and allows the target module to put back pressure on the initiator. The exclusion rule can be used to model busy buses or slow target modules.

After the target module acknowledged the request, it sends a response. The target module starts a response by calling `nb_transport_bw()` passing the `BEGIN_RESP` phase. The initiator acknowledges the response by calling `nb_transport_fw()` and passing the `END_RESP` phase. The target module must not send another response until the initiator acknowledged a previous response. This is referred to as the *response exclusion rule* and allows the initiator module to put back pressure on the target module.

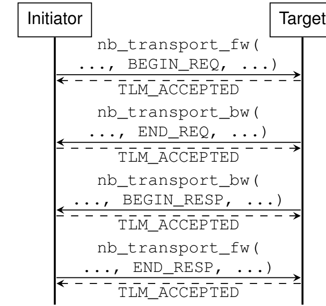


Figure 3: Sequence diagram illustrating the non-blocking base protocol of TLM.

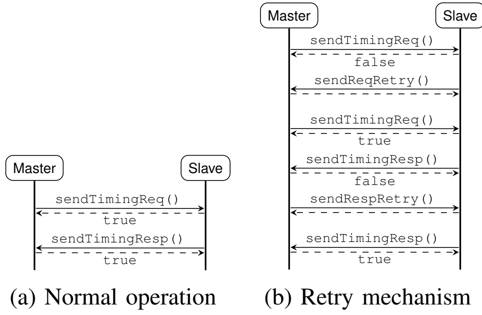
B. The gem5 Simulator

In contrast to SystemC, which is a generic standard, gem5 is a complete simulation framework that provides specific models for a variety of system components and means for easy configuration [5]. gem5 includes CPU models in various levels of detail and complexity (atomic, pipelined, out-of-order), caches, buses, peripheral devices, and DRAM controllers [17]. Each CPU model supports multiple ISAs including Alpha, x86, ARM, and RISCV. The CPU models can be integrated with two different memory backends. While the *Classic* memory model provides a fast and easy way to configure the memory system, the *Ruby* model focuses on the simulation of cache coherent memory systems. Furthermore, gem5 features the replay of *Elastic Traces* [18], which not only enables a faster full system simulation but also maintains a reasonable accuracy. Overall, these exciting features makes gem5 a powerful tool for analyzing and evaluating system-level architectures as well as CPU micro-architectures. In this paper, we use the classic memory backend of gem5 which we describe in the following.

In gem5, all objects that communicate via the memory system are called *memory objects*. Memory objects communicate with each other through *ports*. Ports are always used in pairs consisting of a *master port* and a *slave port*. The master port sends requests and receives responses. The slave port receives requests and sends responses. Typically, a master module (e.g. a CPU) has one or more master ports and a salve module (e.g. a Memory) has one or more slave ports. Interconnect components (e.g. bus, cache, bridge) have both port types.

A connection between memory objects is established by binding master and slave ports to each other. Each connection binds exactly one master port to exactly one slave port. Packets encapsulate transfers between memory objects. They contain the actual payload data of memory accesses, as well as meta data including address, size, command and status. Memory objects communicate, by exchanging references to packets in a series of function calls. Figure 2b depicts an overview of the communication of memory objects in gem5. The similarities of the communication mechanisms in gem5 and TLM are evident from Figures 2a and 2b.

Ports support three different access types: *timing*, *atomic*, and *functional*. Atomic and functional accesses are synchronous. To send an atomic or functional request, the master module calls `sendAtomic()` or `sendFunctional()` on



(a) Normal operation (b) Retry mechanism

Figure 4: Series of function calls required to simulate one timing memory access in gem5. In case one module is busy (returns `false`), the retry mechanism is used.

one of its master ports. The response is provided immediately when the function returns. Atomic accesses are employed for simulations (e.g. fast forwarding) for which accurate timing is not a requirement. Functional accesses are mostly used for initialization, debugging, and to load binaries to memories. Atomic accesses directly correspond to blocking transactions in TLM. Similarly, functional accesses correspond to debug transactions.

Timing accesses provide a model for realistic timings of real memory systems. They use an asynchronous protocol where responses are not instantaneous. Figure 4 shows a possible sequence diagram for a timing access in gem5. The master module sends a timing request by calling `sendTimingReq()` on one of its master ports. The corresponding slave port may accept or reject the packet. The boolean return value of `sendTimingReq()` indicates the accept/reject status of a request. If a packet is rejected, the master module must not send any further packets using this port. When the slave port is ready to receive a request, it calls `sendRetryReq()` to notify the master port. The master port can then resend the request by repeating the call to `sendTimingReq()`. However, the request may be rejected again. Once the slave module accepts the request, it may forward the request to another module (e.g., if it is a bus) or process the request (e.g., if it is a memory). On the response path, a similar protocol is used. The slave module calls `sendTimingResp()` and the master port may accept or reject the response. After a rejection, the master port calls `sendRetryResp()` to indicate that it is ready to receive a response.

Timing accesses in gem5 are similar to non-blocking transactions in TLM in the sense that they split a single memory access in request/response phases and use forward/backward paths for communication. However, gem5 and TLM employ a different mechanism for enforcing back pressur. While TLM defines exclusion rules, gem5 uses the retry mechanism described above. Table I summarizes the correspondence of gem5 access types to TLM transport interfaces. As depicted in the table, gem5 does not have any equivalent access type to DMI.

Table I: gem5 access types and their corresponding transport interfaces in TLM

gem5 Access Type	TLM Transport Interface
Atomic	Blocking
Timing	Non-Blocking
Functional	Debug
—	DMI

Table II: Comparison of the fields used to encapsulate transfers in gem5 and transactions in TLM

gem5 Packet	TLM Generic Payload
`flags`	-
`cmd`	`command`
`data`	`data_ptr`
`addr`	`address`
`size`	`data_length`
---	`byte_enable_ptr`
---	`streaming_width`

IV. Combining gem5 and TLM

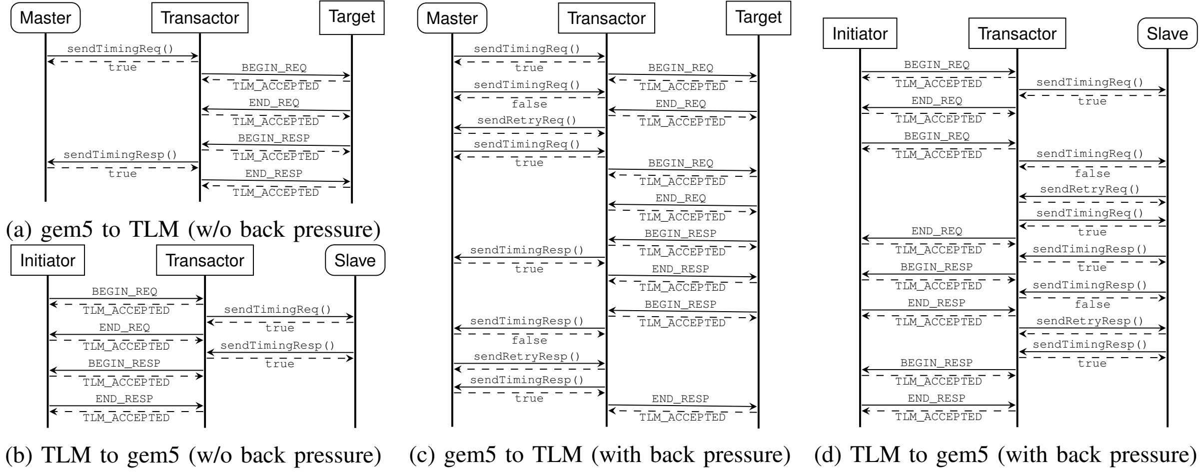
Since June 2014, gem5 supports co-simulation in a SystemC environment. The coupling is achieved by hooking the event loop of gem5 to the SystemC kernel. A special SystemC module hosts the gem5 simulation. It is responsible for assembling and initializing all gem5 modules according to a configuration file and implements a process that the SystemC kernel invokes for each gem5 event.

This simple coupling is limited in its usefulness as it does not allow for communication between gem5 and SystemC modules. To overcome this limitation, we present a mechanism that translates gem5 memory accesses to TLM transactions and TLM transactions back to gem5 memory accesses. Our transactors allow to connect arbitrary gem5 and SystemC modules creating a wide range of possibilities for research on system architectures.

In the following, we describe our *slave transactor* that translates gem5 memory accesses to TLM transactions as well as our *master transactor* that translates TLM transactions to gem5 memory accesses. We also discuss a special mechanism that ensures correct translation when both transactors are used. Finally, we describe how our transactor mechanisms can be used to connect gem5 and SystemC modules.

A. Slave Transactor

The slave transactor translates memory accesses in gem5 to TLM transactions. For each access, the transactor first converts the corresponding gem5 packet to a TLM generic payload object. The transactor acquires a generic payload object and initializes it according to the information provided by the packet. Table II lists the relevant fields of the gem5 packet and its equivalent fields in the TLM generic payload object. The `cmd`, `data`, `addr`, and `size` fields of the packet are directly converted to their equivalent in the generic payload. Since the `flags` are gem5-specific, they are only checked but not converted. The `byte_enable_ptr` and `streaming_width` fields are simply initialized to their default values, as gem5 does not support features that are equivalent to the byte-enable and streaming features of TLM. In order to remember the



(a) gem5 to TLM (w/o back pressure)

(b) TLM to gem5 (w/o back pressure) (c) gem5 to TLM (with back pressure) (d) TLM to gem5 (with back pressure)

Figure 5: Sequence charts illustrating the mechanism for transacting between the gem5 domain and the SystemC domain.

original packet, the transactor attaches a reference to the packet to the generic payload using its extension mechanism [15].

Transacting atomic and functional accesses from gem5 to TLM is almost trivial. A functional access simply results in a corresponding call to the debug transport interface of the transactor's master socket. Similarly, an atomic access leads to a call to the blocking interface. However, in order to correctly model the timings, we need to convert the TLM timing annotation (referenced to time object) to the gem5 annotation (return value denotes number of ticks).

Timing accesses, however, are more difficult to translate. The transactor needs to correctly implement both, the gem5 timing protocol and the TLM base protocol. Most notably, the transactor needs to enforce back pressure in both directions.

Figure 5a illustrates the normal operation of the master transactor without any back pressure. In the figure, calls to the non-blocking transport interface are shortened and only denoted by the phase argument. When the master module sends a request by calling `sendTimingReq()`, the transactor acknowledges this request by returning `true`. It then calls `nb_transport_fw()` passing `BEGIN_REQ` to initiate a new transaction and to forward the request to the SystemC target module. The transactor then needs to wait for the target module to advance to the `END_REQ` phase.

During the `BEGIN_REQ` phase, the transactor rejects all further requests that it may receive from the master module. In other words, each call to `sendTimingReq()` of the master module will return `false`. This ensures that the transactor complies to the request exclusion rule of TLM and that the back pressure is forwarded to the gem5 domain. If the transactor rejects a request, it will notify the master module by calling `sendRetryReq()` when the target module advances in the TLM base protocol to the `END_REQ` phase. This mechanism is illustrated in Figure 5c.

On the backward path, the target module initiates a response by calling `nb_transport_bw()` and passing the `BEGIN_RESP` phase. To handle the response, the transactor first retrieves the original packet that it attached to the generic payload object at the beginning of the transaction. Then it transforms the packet to a response and sends the response to the master module by calling `sendTimingResp()`. If the master module returns `true` (Figure 5a), the transactor completes the transaction by calling `nb_transport_fw()` and passing `END_RESP`. Otherwise, the transactor needs to wait for notification to retry (Figure 5c). The transactor only completes the TLM transaction, when the master module accepts the response. This ensures that the back pressure is forwarded to the target module, as the response exclusion rule ensures that the target module does not send another response.

B. *Master Transactor*

The master transactor translates TLM transactions to gem5 memory accesses. For each transactions, it first converts the corresponding generic payload to a gem5 packet. The transactor allocates a new packet and initializes the common fields according to Table II. The `flags` field is set to a default value and the `byte_enable_ptr` and `streaming_width` fields of the generic payload are ignored. However, if an initiator module uses these features, the transactor generates an error.

The transactor attaches a reference to the original generic payload object to the newly created packet. In gem5, this is achieved by pushing a sender state object to a stack of states that is maintained by the packet. The state can be retrieved later through a pop operation in order to restore the original generic payload.

Debug and blocking transactions are implemented in accordance to the previous section. A call to the debug transport interface of TLM will lead to a call to `sendFunctional()` in gem5 and a call to the blocking transport interface will lead to a call to `sendAtomic()`. Thereby, the transactor ensures that the timing annotation for blocking accesses is converted correctly.

Figure 5b depicts the conversion of the non-blocking transport interface to timing accesses in gem5 without back pressure. When the initiator module initiates a new transaction, the transactor calls `sendTimingReq()` to send the translated

packet to the slave module. If the slave module accepts the request (returns `true`), the transactor immediately advances to the `END_REQ` phase. Otherwise, the transactor waits for notification to retry by the slave module. Only after the slave module accepts the request, the transactor advances to the `END_REQ` phase. This ensures compliance to the request exclusion rule and forwards the back pressure to the TLM domain. This retry mechanism is shown in Figure 5d.

The slave module starts a response by calling `sendTimingResp()`. If the transactor is not processing a previous response, it accepts the response and returns `true` (Figure 5b). Then, the transactor retrieves the original generic payload object form the sender state stack of the packet and sends this generic payload to the initiator module by calling `nb_transport_bw()` and passing `BEGIN_RESP`. Once the initiator module acknowledges the response by sending `END_RESP`, the transaction is complete. While the transactor is waiting for the `END_RESP` signal, it rejects all other responses send by the slave module (Figure 5d). Once the transactor receives `END_RESP`, it notifies the slave module by calling `sendRetryResp()`. This mechanism forwards the back pressure on the return path from the TLM domain to the gem5 domain.

C. Combining both Directions

The transactor mechanisms described above translate gem5 timing accesses to TLM non-blocking transactions and vice versa. On the forward path, the transactors attach the original transaction object (packet or generic payload) to the newly created, translated transaction object. This allows the transactors to restore the original object on the backward path in order to create the correct response. This mechanism works well if requests are only send in one direction as depicted in Figures 1a and 1b.

A system where requests may be send from the gem5 domain to the SystemC domain and vice versa needs special consideration. For example, the system shown in Figure 1c connects a subsystem consisting of a CPU and a scratchpad memory in the gem5 domain to an interconnect component in the SystemC domain. If we assume that multiple of these gem5 subsystems are connected to the SystemC interconnect, a memory request that originates in the gem5 domain may be translated to a TLM transaction and then be forwarded back to the gem5 domain. For instance, this happens if a CPU accesses data in the scratchpad memory of another CPU.

Using the mechanism described above, the slave transactor would create a new generic payload object, initialize it according to the gem5 packet and attach the original packet as a TLM extension. The interconnect handles the transaction and may forward it to a master transactor, which translates the transaction back to the gem5 domain. For this, the master transactor creates a new gem5 packet, initializes it according to the generic payload, and attaches the generic payload as a gem5 sender state.

The above mentioned naive approach is not appropriate due to the following two reasons. First, the conversion from a generic payload to a new packet incurs unnecessary overhead. Instead, the master transactor could directly use the original packet from the generic payload extension. Second, the slave module may have certain assumptions on packets that arrive at the module. For instance, it could expect that a certain sender state object is attached to the packet. However, the slave and master transactors do not translate the sender state. However, simply restoring the original packet would solve this issue.

The transactors implement a so-called *pipe-through* mode to resolve the issue discussed above. When the master transactor receives a generic payload, it first checks if the gem5 packet extension is present. If so, the transaction originated in the gem5 domain and was translated by a slave transactor. Otherwise, the transaction was initiated in the SystemC domain. In the latter case, the master transactor converts the packet as described in Section IV-B. In the former case, however, the transactor uses the original packet that is attached to the generic payload. It also sets a pipe-through flag in the generic payload extension in order to inform the slave transactor on the backward path.

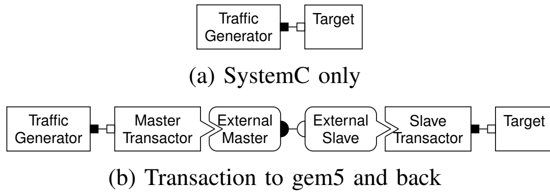
Similarly, the slave transactor needs to check if a response is originated in the gem5 domain. To implement this functionality, the slave transactor checks the pipe-through flag that is written by the master transactor. If the flag is not set, the slave transactor operates normally as described in Section IV-B. This includes modifications to the original packet to mark it as an error-free response. If the flag is set, this indicates that the packet was already transformed into a response by the slave module that answered the original request. In this scenario, the slave transactor simply forwards the packet without any modifications back to the gem5 domain.

The pipe-through mode is only implemented for memory accesses that originate in the gem5 domain. A similar mechanism is required for TLM transactions that are transacted to the gem5 domain and then back to the SystemC domain. However, this scenario is probably not a useful scenario for a coupling of gem5 and SystemC as normally gem5 would be used for simulating CPUs. Therefore, this scenario is not considered in our implementation.

D. Usage

The source code of our SystemC and gem5 coupling can be found in the `util/tlm` directory of the gem5 repository[2]. The coupling is managed mainly by five classes. `SCMasterPort` and `SCSlavePort` are implementations of the external port interfaces that gem5 provides for external couplings. In our implementation, these two classes manage the translation between TLM transactions and gem5 timing accesses. `Gem5MasterTransactor` and `Gem5SlaveTransactor` are SystemC modules that represent the coupling in the SystemC domain. They are bound to one `SCMasterPort` or `SCSlavePort` and provide a TLM target socket or initiator socket, respectively. This allows for easy connection to other SystemC modules.

[2]https://gem5.googlesource.com/public/gem5



(a) SystemC only

(b) Transaction to gem5 and back

Figure 6: Setup for measuring the overhead of the transaction mechanism.

`Gem5SimControl` is the central SystemC module that manages the coupling and encapsulates the gem5 components in the SystemC environment.

In order to use gem5 components within a SystemC simulation, the user needs to instantiate the `Gem5SimControl` class and provide a gem5 configuration file. The control module assembles the gem5 component according to this configuration and manages the simulation of all components it encloses. This includes all the external master and slave ports that are supposed to communicate with SystemC components. For each external master or slave port that is specified in the gem5 configuration, the user also needs to instantiate a corresponding master or slave transactor module in the SystemC domain. The external ports and transactor modules are associated with each other through their name. Based on the name, the `Gem5SimControl` module automatically binds external ports and transactor modules to each other.

V. Evaluation

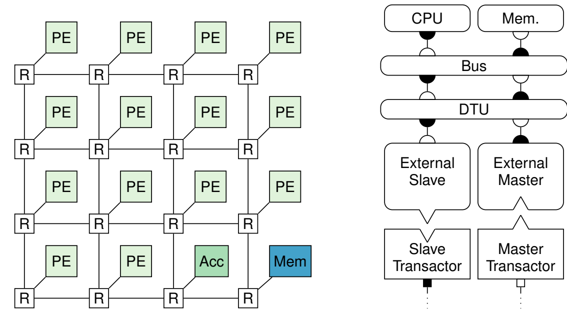
In the following we present three experiments with different gem5 and TLM coupling scenarios.

A. *Pure Runtime Overhead*

To estimate the runtime overhead for transaction to and from gem5, we used a simple benchmark consisting of two SystemC modules. The *Traffic Generator* is an initiator module that randomly generates new transactions. The *Target* module provides a simplistic, functional memory model. Both modules can be found in the example use-cases that are part of the gem5 source code. Figure 6 shows the setup for our measurement.

In a first measurement, we connected the modules directly in SystemC without any transactors (Figure 6a). We modified the Traffic Generator so that it measures the host CPU time required to complete each transaction. On an Intel i7-4790 host CPU, we measured an average of $818\,\mathrm{ns/transaction}$ for a total of 1.5 million transactions.

In a second measurement, we added transactors that translate the transactions to the gem5 domain and back to the SystemC domain (Figure 6b). The gem5 domain does not implement any functional behavior. It forwards all memory accesses from the external master directly to the external slave. In contrast to direct communication in the previous measurement, each transaction now needs to be translated to the gem5 domain, forwarded within the gem5 domain, and translated back to the SystemC domain. Using this setup we measured an average of $3\,\mu\mathrm{s/transaction}$. This is about $3.7\times$ slower than the direct TLM to TLM communication.



(a) 4×4 mesh NoC architecture (b) Processing Element (PE)

Figure 7: MPSoC platform used for evaluating the coupling between gem5 and SystemC. While the tiles are gem5 sub systems, the NoC is modelled in SystemC.

It is not trivial to measure where exactly in the chain of transactors which portion of the host CPU time is spent. Therefore, we estimate the worst case overhead for one transaction based on our measurement. Since we consider the worst case, we can neglect the overhead for forwarding memory accesses in the gem5 domain and assume that all time is spent for transacting the accesses. Further, we assume master and slave transactor induce about the same overhead. Based on these assumptions, we conclude that converting between timing accesses and TLM transactions increases the host CPU time required for handling one transaction at most by factor 2.

When we simulate the setup from Figure 6, most of the host CPU time is spent on modeling the communication. Since the initiator and target modules are very simplistic, the time required for simulating their behavior is negligible. In a more realistic scenario, a larger fraction of the host CPU time would be spend on simulating behavior. Furthermore, not every binding between two modules requires transaction. Normally there are groups of modules in gem5 and SystemC that are directly connected and form sub-systems. Only communication between these subsystems requires transaction. Therefore, the actual runtime overhead for entire MPSoC simulations will be much lower than factor 2.

B. *Case-Study: MPSoC Simulation*

In order to illustrate the usage of the SystemC coupling and to analyze the runtime overhead in a realistic scenario, this section discusses the simulation of an MPSoC architecture.

Modern MPSoC architectures typically use a Network-on-Chip (NoC) for on-chip communication. However, the classic memory backend of gem5 does not provide a model for Network-on-Chip (NoC) architectures. Therefore, we use gem5 to simulate all tiles of the MPSoC architecture and use our coupling to connect the tiles to a custom SystemC NoC model in Synopsys Platform Architect.

On this virtual platform, we run the M3 operating system [19]. M3 OS is a micro-kernel based operating system for heterogeneous many-core architectures. The M3 approach hides heterogeneity behind a common hardware interface and provides isolation on the NoC-level. For this, each tile requires

a special hardware module called *Data Transfer Unit* (DTU). An implementation of the DTU is available for gem5[3].

The system employs a 4 by 4 NoC in mesh topology and has a total of 16 tiles. 14 tiles are processing elements, one tile is an accelerator, and one tile provides the memory interface. Each of the tiles is simulated in the gem5 domain and uses a similar setup as shown in Figure 1c. However, the local bus is not directly connected to the NoC. Instead, the DTU acts as a bridge between local bus and global interconnect. Figure 7 visualizes the complete system.

One of the processing elements hosts the kernel. The remaining 13 processing elements run a benchmark that performs a series of system calls which creates traffic on the NoC. The simulation further includes the initialization of the system which also creates NoC traffic as the kernel loads binaries from the main memory to the processing elements.

We analyzed the simulation using the callgrind tool [20] of the valgrind instrumentation framework [21]. This analysis shows, as expected from the discussions in Section V-A, that 7.58% of the total host CPU time are spent on the components managing the coupling of gem5 and SystemC. More than half of this overhead (4.1% of total simulation time) is spent on managing and synchronizing the gem5 event queue. 3.48% of the total simulation time is spent for transacting TLM transaction and gem5 memory accesses.

This case-study illustrates how the coupling of gem5 and SystemC widens the range of platforms that can be explored. It also shows that the coupling itself only induces a relatively small overhead to the total execution time of the simulation.

C. Case-Study: External DRAM Simulator

Furthermore, we studied the coupling of gem5 with an advanced DRAM TLM model called DRAMSys [22]. DRAMSys is a design space exploration framework, which features functional, power and thermal modeling, as well as a sophisticated retention error model. Simulating a Linux boot using the DRAMSys model as main memory results in a slowdown of $1.9\times$ compared to the simulation with gem5's internal DRAM model [17]. This slowdown is not only due to the coupling but rather due to the higher level of details and debug capabilities of the DRAMSys tool. This simulation setup allows detailed explorations, e.g. the power reliability trade-off in the context of *Approximate DRAM* on specific applications [23].

VI. Conclusion

In this work, we presented for the first time a coupling and a detailed description of gem5 and SystemC TLM. Both frameworks, SystemC and gem5 play an important role for industry and academia. Therefore, the coupling opens a huge set of possibilities for system level design space exploration. We have shown that the coupling itself only induces a relatively small overhead to the total execution time of the simulation in realistic MPSoC and DRAM use-cases.

[3]https://github.com/TUD-OS/gem5-dtu

Acknowledgment

This work was partly supported by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden" (cfaed), the DFG grant no. WE2442/10-1, the EU grant no. 732631 and the *Fraunhofer High Performance Center* for *Simulation- and Software-based Innovation*. Furthermore we thank Synopsys for their support and Kira Kraft for her helpful and valuable suggestions.

References

[1] Tom De Schutter. *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press, USA, 2014.
[2] *IEEE Standard for Standard SystemC Language Reference Manual*. IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005), Jan 2012.
[3] Imperas Software Limited. *Open Virtual Platforms - the source of Fast Processor Models & Platforms*, 2017.
[4] Infineon. *Aurix Family*. http://www.infineon.com/cms/de/product/microcontroller/32-bit-tricore-tm-microcontroller/aurix-tm-family/channel.html?channel=db3a30433727a44301372b2eefbb48d9, 2016.
[5] Nathan Binkert, et al. *The gem5 simulator*. SIGARCH Comput. Archit. News, 39(2):1–7, August 2011.
[6] Mingyan Yu, et al. *A Fast Timing-Accurate MPSoC HW/SW Co-Simulation Platform based on a Novel Synchronization Scheme*. In Proceedings of the International MultiConfernce of Engineers and Computer Scientists, 2010.
[7] Alexandre Romana. *SystemC Integration*. In gem5 User Workshop, International Symposium on Microarchitecture (MICRO), Vancouver, BC, USA., December 2012.
[8] Matthias Jung et al. *Coupling gem5 with SystemC TLM 2.0 Virtual Platforms*. In gem5 User Workshop, International Symposium on Computer Architecture (ISCA), Portland, OR, USA., June 2015.
[9] Matthias Jung, et al. *Virtual Platforms for Fast Memory Subsystem Exploration Using gem5 and TLM2.0*. In Ninth International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES13), volume 9, pages 153–156. Academia Press, Ghent, Belgium, 2013.
[10] Radhika Jagtap, et al. *Exploring System Performance using Elastic Traces: Fast, Accurate and Portable*. In IEEE International Conference on Embedded Computer Systems Architectures Modeling and Simulation (SAMOS), July, 2016, Samos Island, Greece, 2016.
[11] A. F. Rodrigues, et al. *The Structural Simulation Toolkit*. SIGMETRICS Perform. Eval. Rev., 38(4):37–42, March 2011.
[12] T. Kuhn, et al. *FERAL - Framework for simulator coupling on requirements and architecture level*. In 2013 Eleventh ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2013), pages 11–22, Oct 2013.
[13] Thorsten Grötker, et al. *System Design with SystemC*. 2002.
[14] John Aynsley. *TLM-2.0 in Action: An Example-based Approach to Transaction-level Modeling and the New World of Model Interoperability*. 2009.
[15] John Aynsley. *OSCI TLM-2.0 Language Reference Manual*. Open SystemC Initiative, JA32 edition, jul. 2009.
[16] Matthias Jung, et al. *TLM modelling of 3D stacked wide I/O DRAM subsystems: a virtual platform for memory controller design space exploration*. In Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, RAPIDO '13, pages 5:1–5:6, New York, NY, USA, 2013. ACM.
[17] A. Hansson, et al. *Simulating DRAM controllers for future system architecture exploration*. In ISPASS 2014 - IEEE International Symposium on Performance Analysis of Systems and Software, pages 201–210, 2014.
[18] Radhika Jagtap, et al. *Exploring system performance using elastic traces: Fast, accurate and portable*. In Walid A. Najjar et al., editors, International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, SAMOS 2016, Agios Konstantinos, Samos Island, Greece, July 17-21, 2016, pages 96–105. IEEE, 2016.
[19] Nils Asmussen, et al. *M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores*. In Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, pages 189–203, New York, NY, USA, 2016. ACM.
[20] Josef Weidendorfer, et al. *A Tool Suite for Simulation Based Analysis of Memory Access Behavior*. In Marian Bubak, et al., editors, Computational Science - ICCS 2004, 4th International Conference, Kraków, Poland, June 6-9, 2004, Proceedings, Part III, volume 3038 of Lecture Notes in Computer Science, pages 440–447. Springer, 2004.
[21] Nicholas Nethercote et al. *Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation*. SIGPLAN Not., 42(6):89–100, 2007.
[22] Matthias Jung, et al. *DRAMSys: A flexible DRAM Subsystem Design Space Exploration Framework*. IPSJ Transactions on System LSI Design Methodology (T-SLDM), August 2015.
[23] Matthias Jung, et al. *Efficient Reliability Management in SoCs - An Approximate DRAM Perspective*. In 21st Asia and South Pacific Design Automation Conference (ASP-DAC), 2016.