SoC Performance Evaluation with ArchC and TLM-2.0

Jörg Walter
OFFIS Institute for Information Technology
Oldenburg, Germany
info@syntax-k.de

Jörg Lenhardt, Wolfram Schiffmann
Faculty of Mathematics and Computer Science
Computer Architecture Group
University of Hagen, Germany
Joerg.Lenhardt,Wolfram.Schiffmann@FernUni-Hagen.de

Abstract—ArchC is an architecture description language that provides instruction set level simulation and binary tool chain generation. It is based on SystemC and can communicate with other SystemC components using transaction level modeling (TLM). In this article we present an upgrade of ArchC that allows TLM-2.0 usage and makes it available in timed simulations. These extensions enable performance evaluation of complete System-on-Chip designs built around an ArchC processor model.

As a proof-of-concept, we examine various TLM-connected memory hierarchies. We outline how model designers can use a combination of fast functional simulation and slow timed simulation to determine an optimal system architecture for a given workload.

Keywords—system-on-chip;ADL;transaction level modeling;

I. INTRODUCTION

A. TLM and ArchC in System Design

The hardware description language SystemC [22] has gained a lot of popularity. It has the advantage of being easily accessible to software engineers. Consisting of a library of C++ classes and templates it facilitates hardware design in a high-level programming language. From the viewpoint of systems engineering, plain SystemC is still low-level, able to work down to the register-transfer level similar to hardware description languages like VHDL or Verilog. While SystemC code is easier to understand for most software engineers, creating an accurate SystemC simulation still takes a lot of effort.

Design teams that build platforms from third-party components seldom work at the register-transfer level. They perform electronic system-level design, choose system components, manage interconnect mechanisms and optimize whole-system behavior. Keutzer et.al. [8] propose a platform-based design methodology. It relies on heavy reuse of hardware models and independence of components. This leads to the concept of transaction level modeling (TLM). It facilitates easy reuse of existing hardware models through a loosely-coupled communication model. Another advantage is that algorithmic models can be mixed with register-transfer level models. Cai and Gajski [2] formalize the approach by defining a hierarchical set of abstraction levels. They show a top-down work flow from specification to final implementation.

SystemC supports TLM through a set of classes and specifications, first shown by Grötker et. al. [5]. SystemC TLM connections can be reconfigured at run-time allowing easy design of reconfigurable logic. Current versions offer a choice between simulation accuracy and simulation speed. In software development simulations a fast but inaccurate mode [1] can be used. For timing analysis a slower but more exact mode is available.

ArchC [17] is an architecture description language (ADL) based on SystemC. It offers another level of abstraction on top of plain algorithmic models. ArchC descriptions operate at the ISA level, which makes them useful for coordinating hard- and software designers. Processor models with basic functionality are fast and easy to write. ArchC can automatically generate development tool chains that further speed up development efforts. For system-level design, ArchC offers TLM support. In this paper we examine the capability of this combination of two native SystemC technologies. For that reason we modified ArchC to support TLM-2.0 in all of its simulation variants.

B. Related Work

ADLs are hardly scarce or new. MIMOLA, for example, can be traced back to the 1970s [25]. Tomiyama et. al. [23] conducted a survey of ADLs and classified them by their main objective. Mishra and Dutt [12], [13] gave an updated overview of current developments.

LISA [24] is widely used, commercially and in academia. The syntax loosely resembles C. It can generate compiler, binary utilities [7], simulators [16] and synthesizable hardware [19] from a single architecture description. Simulators employ just-in-time compilation [14] and can interface with SystemC [9]. LISA is available commercially in tools from Synopsys.

ISAC [11] was derived from LISA with the objective to simplify its syntax and to introduce improvements. It includes automatic compiler generation that only relies on instruction behavior information already present. In contrast to LISA, it does not need separate semantic specifications for the compiler. The Lissom Project has developed generators for an LLVM-based C compiler generator, binary utilities, simulator, and hardware synthesis. The resulting software suite is sold commercially as Codasip Studio.

Initially, nML [3] was just a behavioral architecture description language modeling instruction sets. Operations are specified in a syntax resembling C. More complex instructions and the instruction set as a whole are composed of simple
operations using composition operators much like a formal grammar. In its current form, nML models hardware resources like memory, cache and functional units. It includes its own retargetable C compiler [10], a hardware description generator, and an instruction set simulator that can interface with SystemC. nML is part of the IP Designer tool set sold by Target Compiler Technologies.

Among contemporary ADLs, the mentioned languages are the strongest competitors to ArchC. They have SystemC connectivity and they excel at high-level language compilation. In contrast to them, ArchC is freely available as open source software, which makes it more accessible to research and development. Moreover, it has a moderate learning curve for new developers due to the fact that it uses straight C++.

Others have explored system-level design methodologies as well. Pasricha [15] presents a real-world system-on-a-chip model using SystemC and TLM. The exchangeability of components helps during development by swapping a slow ARM instruction set simulator with a fast minimal bus transaction generator during peripheral testing. While that approach doesn’t aim for run-time reconfigurability, it shows the versatility of TLM models.

Rigo et. al. [18] transfer the concept of system-level design to ArchC and TLM, using the existing TLM-1 support in ArchC. Most of the tools are described and various practical examples of building simulators and virtual platforms are presented. It is shown how ArchC benefits from using SystemC as underlying platform, as it allows integration with other SystemC based technologies.

C. Structure

Section II explains details about TLM and ArchC that go beyond general SystemC knowledge and which are required for full understanding of the remaining sections. It also presents the processor models used for our test platform. Section III contains a functional explanation of the upgrades we made in order to get full TLM-2.0 support in ArchC timed simulators. We show a performance analysis in section IV, and section V summarizes our results.

II. PRELIMINARIES

A. TLM-2.0 for SystemC

Transaction level modeling is a technique that allows hardware designers to disregard details of interconnection mechanisms. Instead of modeling individual signal transitions, system designers model communication transactions as an abstract series of events. They use generic transactions under the assumption that signaling details are well-known and/or do not need further specification until the model is synthesized. Designers do not need to decide on a specific bus implementation if they just want to build a behavioral simulator instead of a synthesizable model.

While TLM in general is a tool-independent technique, SystemC in particular contains an API for it, TLM-2.0. It consists of the core and recommended but optional classes for payload and communications protocol to ensure interoperability between SystemC models from different sources. TLM-2.0 covers the trade-off between timing accuracy and simulation speed by specifying a set of recommended coding styles. If Generic Payload or Base Protocol are not sufficient to model a given component interconnect, or if additional meta-data is needed, TLM-2.0 can be extended in different ways.

In the terminology of TLM-2.0 components have the role of initiator and/or target. A SystemC module modeling a bus system would act as target to a bus master module, while it would itself be initiator when forwarding requests to peripherals as determined by address decoding. These roles are represented through socket objects with an initiator socket connected to exactly one target socket. A SystemC module can have an arbitrary amount of initiator and target sockets.

The following subsections summarize information found in the SystemC Reference Manual (IEEE 1666-2011 [22]).

B. TLM-2.0 Generic Payload

The TLM-2.0 Generic Payload defines a standard set of attributes associated with transactions. A read or write operation (TLM_READ_COMMAND or TLM_WRITE_COMMAND) is represented with the command attribute. Attribute address refers to a location within the simulated address space of the target that the transaction operates on. data_ptr is a pointer to a host array containing transferred data, while data_length represents the number of bytes to be copied from/to this location. Attribute byte_enable_ptr in combination with byte_enable_length refer to an array of values (0x00 or 0xff). Only bytes marked with 0xff are copied from/to data_ptr. Targets reset address to its initial value every streaming_width bytes. Streaming is disabled if streaming_width is set to data_length.

A typical transaction will have at least the command, address, data_ptr, data_length, and streaming_width attributes set. System designers wanting to model a parallel bus more realistically may use byte_enable_ptr and byte_enable_length for writing a single byte to memory having a 16-bit data bus, or streaming_width for repeated transfers from/to a single memory address. Targets return data by modifying the array data_ptr points to and should set transaction meta-data attributes to indicate if the target supports direct memory interface (DMI) with dmi_allowed and the response status (response_status).

With standardized error codes, initiators can handle errors automatically. A response of TLM_ADDRESS_ERROR_RESPONSE can be mapped to a bus exception in a simulated processor, while a status of TLM_COMMAND_ERROR_RESPONSE would be ignored (silently discarding writes to read-only targets), and a TLM_BYTE_ENABLE_ERROR_RESPONSE might indicate a target that is not compatible, triggering a simulation abort.

All TLM-2.0 sockets are templates that allow to specify a specific bus width. Only sockets of matching width can connect to each other. The default bus width is 32 bit. All transfers are done at bus width and components are free to assign any order to the bytes of a bus word. IEEE 1666-2011 states that host byte order defines least and most significant byte (LSB and MSB), but which byte is associated with the lowest address is specific to each component. The relationship
between MSB and LSB is preserved while transferring data between components of different endianness. At the same time the local address of each byte inside a bus word necessarily changes. As a result, transfers at full bus width do not need byte order conversion.

For example, the second byte of a 32-bit word 0xAABBCDDE located at address 0 lies at address 1. A little-endian model would see 0xCC at that address, a big-endian model would see 0xBB. On a little-endian host, the little-endian model would access it at offset 1 of the TLM data array that is passed around, while the big-endian model would access it at offset 2. While they disagree on which byte is second, they build the same TLM data array. Thus, byte transfers can be modeled by modifying the address instead of swapping data bytes. This is called address swizzling. Aligned partial-width transfers (e.g., half-word access that does not cross word boundaries) work along the same lines. The intention of these rules is that the common case is fast and simple and the uncommon case, albeit complex, is still possible.

C. TLM-2.0 Base Protocol

The Base Protocol specifies the order of events that make up a single transaction. Four phases are defined: BEGIN_REQ, END_REQ, BEGIN_RESP, and END_RESP, specifying begin and end of request and response, respectively. Connections can have multiple unfinished transactions, but only one transaction is allowed to be in the BEGIN_REQ stage. That way, request pipelining can be modeled.

IEEE 1666-2011 associates each phase with either initiator or target. Initiators can cause a state transition to the BEGIN_REQ and END_RESP phase, while targets trigger END_REQ and BEGIN_RESP. Components can ignore and/or skip most phases, and both sides can prematurely complete a transaction if they consider their part done. Components that do not provide enough timing accuracy to give reasonable meaning to a phase can still interact with components that do.

D. TLM-2.0 Timing Styles

SystemC allows different timing and synchronization styles, and TLM is independent of them. Two coding styles are explicitly supported by TLM-2.0: loosely-timed and approximately-timed. Loosely-timed transactions only have timing points for start of request and start of response, expressed through call and return from a blocking API call. This is enough accuracy to support software development and to do basic performance analysis of software and architecture.

TLM-2.0 offers two performance-enhancing techniques for loosely-timed models: temporal decoupling allows modules to run ahead of current simulation time in order to reduce scheduling overhead and to increase locality of reference. DMI allows initiators to access a target’s memory space directly. System designers can model more timing points in a transaction through the approximately-timed coding style, which uses non-blocking transport calls. Simulations usually run slower due to the added function call and scheduling overhead. A component can use convenience sockets provided by SystemC that translates between the interfaces.

TLM-2.0 does not directly support cycle-accurate modeling of buses. IEEE 1666-2011 points at the possibility to extend the approximately-timed coding style with enough timing points to get cycle-accurate timing, but no standardized protocol is available.

E. ArchC

ArchC offers functions related to processor design. As baseline feature, it can create a behavioral simulator for a processor. Other functions need a more detailed model and thus are optional. For a more thorough explanation, refer to the ArchC documentation [20], [21], and to [18].

This section covers details later explanations depend on. It reflects the state of ArchC version 2.1 unless otherwise noted. Section III presents all modifications to ArchC.

Some features supported by older versions of ArchC have not been updated to the latest version of ArchC. To give a universal overview, this section covers all features of the 1.6 and 2.1 release series. No ArchC version combines all of them. Some features are incompatible with others. Table I shows a summary of the combinations that are (or have been) possible.

Functional simulation is the the base feature of ArchC. Developers specify a new processor architecture in terms of its instruction set, and they specify instruction behavior as snippets of C++ code. They work in a top-down style, specifying the bare instruction set first and adding details later.

The basic simulator can run executables for the simulated instruction set. ArchC accepts two file formats for target code: plain binary code in a simple text format based on hexadecimal numbers and industry-standard ELF executables. ArchC can load preexisting executables for the target platform. This requires system call emulation support. Since version 2.0 of ArchC, processor models can interact with other SystemC models via TLM-1.

Timed simulators offer cycle-exact instruction timing. Processor designers can model pipelines and multi-cycle instructions. Timed simulations need more detailed instruction behavior and declaration of pipeline stages. ArchC supports formatted registers that can serve as registers between pipeline stages. It does not manage resource conflicts. Instruction behavior code has to do that manually. This feature is still declared beta quality [20]. Part of our efforts were dedicated to fixing this and adding TLM support to timed simulators.

Since needs vary and maximum performance is needed for some usage scenarios, ArchC used to offer compiled simulators. Instead of interpreting machine code, it generates a C++ translation of a target executable that is then compiled and run. This feature has not been updated for ArchC 2.0 yet. This may be due to the problem that static binary translation does not allow modification of program memory. This excludes

| Table I. Comparison between simulator variants of ArchC. Speed ratings are given in million instructions per second. |
|--------------------------------------------------|--------------------------------------------------|--------------------------------------------------|
| Simulator Generator: Functional, Timed, Compiled | Functional, Timed, Compiled | Functional, Timed, Compiled |
| Simplicity | interpreter | interpreter | static binary translation |
| Timing | loosely timed | cycle accurate | untimed |
| Speed | 30 Mi/s | 0.25 Mi/s | 240 Mi/s |
| System Call Emulation | available | available | available |
| Dynamic Loader | available | available | available |
| Debug Support | -- | -- | -- |
| TLM Connectivity | available | available | available |
use cases where simulation performance matters, e.g. platforms running operating systems that load and unload applications at run time, or programs running in a (just-in-time compiling) virtual machine.

ArchC is able to generate a GNU binutils tool chain containing assembler, linker, and other tools for the target platform and provides full-featured debugging support through the GNU debugger using the GDB client-server protocol. This is independent from building an actual debugger for the target platform. If a simulator has debug support, any preexisting GDB for the target architecture can connect to it. If no GDB port exists so far ArchC can port it to the target architecture automatically.

In order to allow testing real-world executables ArchC offers basic support for frequently used POSIX system calls. Realistic test applications can be run on the simulator, possibly even unmodified existing executables. The set of system calls is limited to simple file I/O (which operates transparently on the host file system) and memory management. Some more system calls are just no-operation stubs.

ArchC can load executables that refer to dynamically linked libraries. Existing binaries for the target architecture can be run with an ArchC-specific dynamically linked C runtime library, assuming they only rely on system calls supported by ArchC.

There is experimental work in progress to provide hardware synthesis from an ArchC model. No implementation has been published at the time of this writing, but Goto [4] has shown the general approach.

ArchC models can communicate with other SystemC components via TLM-1. There are initiator ports for memory-like resources and target ports to model external interrupts. Initiator ports can replace internal memory models and possibly even registers/register files. System designers can use interrupt ports to model interrupt lines or internal exceptions. These have no non-TLM equivalent. TLM support in ArchC version 2.1 is restricted to functional simulators and TLM-1. In contrast to TLM-2.0, TLM-1 has no provisions for general-purpose interoperability. There are no equivalents to TLM-2.0 Generic Payload or TLM-2.0 Base Protocol, and there is no built-in support for temporal decoupling or DMI. In order to provide some degree of interoperability, ArchC itself defines a generic payload and protocol.

F. Structure of ArchC Models

ArchC processor models consist of several independent parts, most of which can be skipped if the associated functionality is not desired. The very minimum needed to create a working processor simulator are specifications of architecture resources, instruction set architecture, and instruction behavior.

The Architecture Resource Declaration describes architecture resources. These include basic declarations like name, word size and byte order, ISA registers, memory interfaces, as well as internal registers available to instruction behavior descriptions, and in the case of timed simulators details about the processor pipeline(s).

The Instruction Set Architecture Declaration lists available instructions and defines their encoding. Instructions can be grouped by instruction type, sharing a common opcode layout per type. For binary utilities generation it contains data about assembler syntax. ISA declarations can also contain additional assembler information like register names and pseudo-instructions.

The Instruction Behavior Description is an almost plain C++ source file that contains SystemC source code for the behavior of each instruction. Instruction behavior is organized hierarchically. Each instruction’s behavior is contained in its own method. Another set of methods contains common behavior for each instruction type, and a global behavior method contains instruction-independent behavior. Initialization and cleanup methods are available as well.

The SystemC Main Module contains code to instantiate an actual simulator object, initialize it and finally start simulation. If the processor model contains TLM ports model writers can instantiate SystemC modules and set up TLM connections in this file.

An example how a main module for a TLM-enabled processor might look is shown in listing 1. Lines 2 to 4 allocate a TLM memory model and the processor simulator and line 6 creates the TLM connection between them. In line 7, the processor model processes all command line arguments. It loads the executable at that time. Line 9 finally starts the SystemC scheduler. After it finishes, the program prints execution statistics and exits.

G. ArchC Simulator Internals

ArchC routes all memory accesses through class ac_memport which forwards them to a subclass of abstract class ac_inout_if. A non-TLM processor model uses subclass ac_storage to model random-access memory while a TLM-enabled model uses ac_tlm_port. In the latter case, the ac_tlm_port object is accessible with a suffix of _port.

The actual simulator runs as an SC_THREAD. Method mipsi::behavior contains a loop that decodes the instruction ac_pc points to and then executes the generic behavior method, followed by an instruction-type-specific behavior method, followed by an instruction-specific behavior method. If an instruction needs to be aborted for some reason like an interrupt request behavior methods can call ac_arch::ac_annul(). This causes the execution loop to skip calling the remaining behavior methods for the current instruction.

Listing 1. SystemC initialization code for an ArchC simulator.

```c
int ac_main(int ac, char *av[]) {
  SimpleMemory mem("mem", 5242880,
  sc_time(70, SC_NS), sc_time(70, SC_NS));
  mipsi mipsi_proc1("mips1");
  mipsi_proc1.DM_port(mem.socket);
  mipsiProc1.init(ac, av);
  sc_start();
  return mipsi_proc1.ac_exit_status;
}
```
The behavior loop calls `sc_core::wait` once each `ac_module::instr_batch_size` instructions. This is a form of temporal decoupling in order to reduce SystemC scheduling overhead. The instruction batch size can be reduced or disabled in case of multi-processor simulations. The behavior thread waits for one nanosecond per instruction batch.

Timed simulators have a slightly different class structure. Instead of a single behavior thread in the simulator class, execution consists of one `SC_METHOD` per pipeline stage. Each pipeline stage is a separate class derived from `ac_stage` and is a member object of the main simulator class. These `SC_METHODs` call instruction behavior methods like functional simulators, but all pipeline stages execute in parallel each triggered by an `sc_clock`. The simulator main class contains a final `SC_METHOD` called `ac_verify` that handles pipeline stalls. Completion of the individual pipeline stages triggers it.

### H. MIPS Processor Models

Our test platform uses the MIPS architecture. This is the only architecture that has both, a functional (mips1) and a timed (r3000) simulator readily available on the SourceForge project page of ArchC. This approach gives adequate comparability between functional and timed simulation. We used version 0.7.8 of the functional MIPS-I model. It did not need any modifications except replacing the regular device memory with a TLM port in mips1.ac. For timed simulations we used MIPS R3000 model version 0.7.2. We adapted the model to use the non-blocking API, and we had to fix some minor concurrency bugs exposed by our simulator upgrade.

In order to demonstrate the process of comparing and evaluating multiple design choices, a Harvard variant of the R3000 processor model uses a different bus interface for data accesses. Instruction set logic uses one TLM port while instruction behavior methods use a different port.

### III. ArchC TLM-2.0 Upgrade

For the test platforms we used SystemC 2.3 (which includes TLM-2.0) and ArchC 2.1. We modified ArchC to work with SystemC 2.3, eliminated all calls to deprecated functions and some compiler warnings. We upgraded TLM connectivity to TLM-2.0 and added TLM support for timed simulators. All modifications directly relate to TLM-2.0 usage and merely allow use of existing functionality with TLM-2.0. In order to achieve the timing precision that a pipelined model allows we slightly modified the timed simulation API.

#### A. Memory Access

In order to get correct handling of byte order, we changed the way `ac_memport` handles reads and writes. In ArchC 2.1 it converts words to target byte order, but TLM-2.0 uses host byte order. The upgraded `ac_memport` does not do that anymore while `ac_tlm_port` does address swizzling, if required. We adjusted the internal memory model `ac_storage` so that non-TLM models continue to work.

There is no ArchC syntax to specify memory transfer width for individual ports, so the present implementation uses a constant bus width of 32 bit, which covers most contemporary on-chip bus systems. The width is implemented as a template parameter so future ArchC extensions can provide different widths.

#### B. Functional Simulators

We upgraded the existing TLM support of ArchC to adhere to the TLM-2.0 Base Protocol. The blocking transport of TLM-2.0 has similar calling conventions to the ArchC TLM-1 transport interface. The upgrade uses the loosely-timed coding style including temporal decoupling and DMI. The abstract memory interface of ArchC, `ac_memport`, allowed to hide all implementation details so that the processor-side API remains source compatible. The peripheral side changed as that was the very point of these modifications.

Since there is no existing API for memory access errors in ArchC, the upgrade only produces diagnostic output if a transaction fails. For now a processor model that wants to provide exception handling can do so externally. It can route all TLM requests through a memory controller that uses the ArchC interrupt mechanism to invoke exception handlers.

Each TLM port keeps track of its accumulated delay, which can be retrieved through a new method `get_delay()`. When local simulation time is synchronized, all delays are added to the instruction execution time. This introduces inaccuracies, as an instruction fetch delay less than the clock period would usually lead to a total instruction delay of one clock period instead of a clock period plus memory read delay. It does not account for parallelism in accessing multiple buses at the same time.

#### C. Timed Simulators

Timed simulators run in `SC_METHODs`. IEEE 1666-2011 forbids to use the blocking API, so the existing `ac_memport` API cannot be used. We changed memory accesses to use the approximately-timed coding style via the non-blocking interface. The new implementation honors all aspects of the approximately-timed coding style as outlined in IEEE 1666-2011 so that all approximately-timed components should work, but only the `tlm::BEGIN_REQ` and `tlm::END_RESP` timing points are used.

Bus accesses are implicitly pipelined instead so that multiple stages trying to access a bus in the same clock cycle do not need locking. Due to the asynchronous API, this is fully transparent to the model writer. There is no prioritization mechanism. Which storage gets access first is indeterminate. With these changes memory read data will not immediately be available, so full source compatibility could not be maintained. We extended `ac_memport` and `ac_tlm_port` with an asynchronous API that uses the new lambda expressions available in C++11 as callback mechanism. That makes updates to the processor model simple.

We also added functions to make handling pipeline stalls easier. Model writers can now issue an indefinite pipeline stall until the memory access callback functions release the pipeline again. Two minor additions to the API deal with synchronization during extended stalls. These changes allow a pipeline to process multiple stalls in parallel, for example by accessing memory through a data and an instruction bus in parallel while hiding a stall normally triggered by data hazards within the pipeline.
MEM_PIPELINE_ACTION (release,);

if (G_MEM.is_valid() && (MEM_WB.rdest == rt))
  id_value2 = G_MEM.resume();
else if (G_MEM.is_valid() && (MEM_WB.regwrite == 1))
  MEM_WB.wbdata = DM.read_byte(EX_MEM.alures);
else
  id_value2 = MEM_WB.wbdata.read();

if (G_MEM.is_valid() && (MEM_WB.rdest == rt))
  id_value2 = G_MEM.resume();
else if (G_MEM.is_valid() && (MEM_WB.regwrite == 1))
  MEM_WB.wbdata = DM.read_byte(EX_MEM.alures);
else
  id_value2 = MEM_WB.wbdata.read();

if (G_MEM.is_valid() && (MEM_WB.rdest == rt))
  id_value2 = G_MEM.resume();
else if (G_MEM.is_valid() && (MEM_WB.regwrite == 1))
  MEM_WB.wbdata = DM.read_byte(EX_MEM.alures);
else
  id_value2 = MEM_WB.wbdata.read();

D. Future Directions

Our modifications are just a first step. ArchC needs several design decisions for full-featured TLM-2.0 support:

- TLM error handling needs to be improved. Currently, errors produce diagnostic output but are ignored otherwise. Some errors could be mapped to exceptions, but that would require a programming model for internal exceptions. These could be modeled after TLM interrupt handlers.

- Bus width needs to be configurable per memory port.

- Unaligned and multi-word transfers could be allowed, or they could be left for model writers to handle.

- Unaligned and partial transfers can use either byte enable masks or unaligned addresses. Unaligned addresses are easier to handle, byte enables can express additional cases. Targets are not required to support either. For maximum compatibility a fall-back mechanism should try both.

- The byte enable mechanism should be exposed to models. For example, the swl and swr MIPS instructions modify one to three bytes of an aligned word. In existing MIPS processor models they must read the word to be modified before writing the changed bytes. Using byte enables the changes could be written directly. The byte enable array follows the TLM byte order rules. Translation would be required to keep models independent of host byte order.

- With more knowledge about architecture resource dependencies, ArchC could suspend pipeline registers automatically. That knowledge is required to fully exploit parallelism.

IV. Performance Evaluation

In order to demonstrate the TLM capabilities of ArchC, we focus on system architectures that designers may encounter while creating models for current computing devices.
A. Peripheral Models

We wrote a library of functional models called SimpleTLM. It represents basic components used in many devices. All models were written from scratch with simplicity as foremost design objective. They comprise less than 1000 lines of code in total. Models are not fully realistic. We only implemented details needed to get more insight during exploration. The classes include a simple memory model that supports DMI, temporal decoupling, and both, non-blocking and blocking transport calls. It supports fixed per-byte read and write delays, mimicking simple static random-access memory timing. Another class implements a fully-associative write-through-cache with FIFO replacement policy.

B. Target Platform Software

We used the MiBench embedded benchmark suite published by Guthaus et al. [6] for performance analysis and selected the automotive/basicmath_small benchmark for all tests. For evaluation of the Harvard architecture processor model we wrote a trivial program that maximizes memory load in order to compare timing behavior of Harvard and Von Neumann architectures. All it does is to use memcpy() to copy the contents of a 64 kiB memory buffer to a second one ten times in a row.

C. Build and Execution Environment

The build and execution platform is Ubuntu Linux 12.10 running in 64-bit mode on an Intel i7 Q820 processor. The test system had 4 GiB of random-access memory and a solid-state disk as mass storage device. We used the operating system supplied versions of GNU binutils (2.22.90) and GNU make (3.81), and pre-release version 4.8-20130123 of GCC. We chose this compiler version due to an internal compiler error related to C++ lambda functions in the shipped version 4.7. All programs were compiled with compiler option -O3.

D. Host Performance

Table II lists the raw numbers. They were obtained by running the platforms under the conditions outlined in section IV-C three times, observing each simulation’s statistics output, and taking the median value. On the test system, functional TLM platforms (mips1-tlm) ran at up to 17 MI/s. Compiled simulations (mips1-tlm-compiled) are an order of magnitude faster, while timed simulations (r3000-tlm) run at just 1% of the speed of functional simulators. These relations are almost the same as with non-TLM platforms (mips1, mips1-compiled, r3000). The performance overhead of TLM is significant. About 77% run-time increase for functional model mips1-tlm over mips1. Timed simulators using the non-blocking interface need 35% more run time, compiled simulators slow down by 70%.

In contrast to the gains observed by Andrews [1], the non-DMI mips1-tlm-nodmi platform is only 26% slower than mips1-tlm. Without temporal decoupling (mips1-tlm-nodt), the benchmark takes six times as long. Disabling both (mips1-tlm-nodmitd) is only a bit worse. Most of that slowdown is probably due to disabled instruction batching: mips1-nobatch takes eight times as long as plain mips1. mips1-tlm-notd shows the timing inaccuracy introduced by the design outlined in section III-B.

E. Target Performance

The non-TLM platforms and the plain TLM variants use unrealistic (instant) memory timing and are not suitable for target performance analysis. The -nocache platforms access memory that has 70 ns read/write latency. This results in a significant reduction of system performance. There is only little difference between functional and timed simulators, mostly accountable to more exact simulation of overlapping instruction and data transactions.

An obvious choice when exploring system performance is addition of a cache. The -cache variants have 64 kiB of cache in between processor and memory. Functional simulators show a 96% increase in simulated performance, while timed simulators show an increase of 239%. The main reason for this difference are estimations done in the cache model. It uses a fixed cache-hit ratio of 0.3 for DMI. Timed simulations exhibit more precise timing as they simulate the exact timing of every memory transaction.

This means that when the cache-hit ratio can be estimated in advance (for example, by running the same benchmark repeatedly), functional simulators can be tuned to give useful results. Designers exploring multiple designs can use the increased host performance of functional simulation to select promising candidates for more thorough analysis with timed simulators.

To show another example of timing analysis using ArchC, we ran the membench test program on the r3000-tlm-nocache and r3000-tlm-harvard platforms. Simulated speed differed by just 13%: The single-bus platform ran at 8.09 million instructions per second, while the dual-bus version ran at 9.17 million instructions per second.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Instructions</th>
<th>Simulated Time (s)</th>
<th>MI/s</th>
<th>Time (s)</th>
<th>MI/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>mips1</td>
<td>1361274783</td>
<td>27.2</td>
<td>50.0</td>
<td>45.8</td>
<td>29.7</td>
</tr>
<tr>
<td>mips1-compiled</td>
<td>1361274783</td>
<td>27.2</td>
<td>50.0</td>
<td>377.0</td>
<td>3.61</td>
</tr>
<tr>
<td>mips1-nobatch</td>
<td>1361274783</td>
<td>33.6</td>
<td>40.5</td>
<td>81.2</td>
<td>16.8</td>
</tr>
<tr>
<td>mips1-tlm</td>
<td>1361274783</td>
<td>69.5</td>
<td>19.6</td>
<td>83.6</td>
<td>16.3</td>
</tr>
<tr>
<td>mips1-tlm-compiled</td>
<td>1361274783</td>
<td>13.6</td>
<td>100.0</td>
<td>9.70</td>
<td>140.0</td>
</tr>
<tr>
<td>mips1-tlm-nocache</td>
<td>1361274783</td>
<td>136.0</td>
<td>10.0</td>
<td>81.2</td>
<td>16.8</td>
</tr>
<tr>
<td>mips1-tlm-nodmi</td>
<td>1361274783</td>
<td>27.2</td>
<td>50.0</td>
<td>102.0</td>
<td>13.3</td>
</tr>
<tr>
<td>mips1-tlm-nodmitd</td>
<td>1361274783</td>
<td>27.2</td>
<td>50.0</td>
<td>521.0</td>
<td>2.61</td>
</tr>
<tr>
<td>mips1-tlm-notd</td>
<td>1361274783</td>
<td>33.6</td>
<td>40.5</td>
<td>496.0</td>
<td>2.74</td>
</tr>
<tr>
<td>r3000</td>
<td>1361274783</td>
<td>29.3</td>
<td>46.4</td>
<td>553.0</td>
<td>0.246</td>
</tr>
<tr>
<td>r3000-tlm</td>
<td>1361274783</td>
<td>29.3</td>
<td>46.4</td>
<td>744.0</td>
<td>0.182</td>
</tr>
<tr>
<td>r3000-tlm-cache</td>
<td>1361274783</td>
<td>36.2</td>
<td>37.6</td>
<td>1060.0</td>
<td>0.143</td>
</tr>
<tr>
<td>r3000-tlm-harvard</td>
<td>1361274783</td>
<td>118.0</td>
<td>11.5</td>
<td>1600.0</td>
<td>0.0852</td>
</tr>
<tr>
<td>r3000-tlm-nocache</td>
<td>1361274783</td>
<td>123.0</td>
<td>11.1</td>
<td>1630.0</td>
<td>0.0837</td>
</tr>
</tbody>
</table>
Investigation showed that this is a limitation of our ArchC upgrade. The current design does not allow executing any stage upstream of a stall even if that stage did not execute anything before the stall. ArchC would have to selectively resume and suspend pipeline registers but it does not know what registers belong to a given stage. Effectively, memory requests are still serialized.

Further analysis using custom debug output showed that reading a single byte on a 50 MHz CPU accessing 70 ns memory takes 120 ns, where 80 ns would be optimal. The single-bus platform needs 140 ns. An optimal solution could achieve a 75% speedup excluding loop control overhead. It would not achieve 100% speedup due to our implicit TLM request pipelining. When the 70 ns for instruction fetch elapse in the single-TLM case, the memory cycle for the memory stage begins immediately instead of waiting for the next clock cycle. The dual-TLM implementation synchronizes with the processor clock, so 10 ns are wasted for each request.

Running the MiBench benchmark on the same platforms shows even less impact for such a CPU-bound task with about 4% speedup. As a result, for this particular platform and workload the complexity of a second memory interface would not be worth the effort.

V. Conclusion

In this paper we have shown that using TLM-2.0 with ArchC is a viable strategy for electronic system-level design. The open source approach allows to update ArchC’s functionality so that we could create the prerequisites for an up-to-date exploration.

Regarding target performance analysis we have shown that timing analysis of full platforms using functional ArchC simulators includes guesswork. If TLM peripherals give decent estimates for DMI requests, accuracy can be improved.

Using TLM with timed simulators shows better results. We were able to exactly analyze and understand unexpected timing results. Overall, all observed results were conclusive.

Our work lays the foundation for further improvements in ArchC simulator versatility and exactness. This will require extension of the ArchC language itself, however.

REFERENCES