Automatic TLM Generation for Early Validation of Multicore Systems

Samar Abdi
Concordia University

Yonghyun Hwang
Qualcomm

Lochi Yu
Universidad de Costa Rica

Gunar Schirner
Northeastern University

Daniel D. Gajski
University of California, Irvine

Editor's note:
This article suggests a methodology to validate software applications for a multicore platform by automatically generating transaction-level models from task-level specification of the applications. Software vendors developing applications for multicore platforms can leverage this methodology for early validation.
—Sandeep Shukla, Virginia Tech

The shift toward multicore platform architectures for embedded systems is driven by the need for higher performance without the cost of retooling for a new technology. These benefits come at the cost of higher design complexity, however, which makes design with conventional cycle-accurate models impractical. Designers are using transaction-level models (TLMs) as virtual platforms for early design validation and software development, although TLM development for multicore platforms can be time-consuming and error-prone. The problem can be alleviated by automatic TLM generation, which can provide designers with early and accurate validation of both functionality and performance.[1]

This article presents methods we have developed for automatically generating TLMs that can be used for early high-level validation of multicore systems. The inputs to automatic TLM generation are application C/C++ tasks mapped to processing units in the platform. On the basis of the mapping, we generate two types of TLMs: functional and timed TLMs. The functional TLM is generated by instantiating the application tasks inside a SystemC model of the platform. For timed-TLM generation, the basic blocks in the application tasks are analyzed and annotated with estimated delays. The delay-annotated C code is then linked with a SystemC model of the hardware and software platform. Both TLMs can be natively compiled and executed on the host machine, making them much faster than conventional cycle-accurate models. As our results demonstrate, TLMs of industrial-scale multicore designs—such as the JPEG encoder, MP3 decoder, and H.264 decoder—are generated and simulated in just a few seconds. Estimation error was 15% less than that in board measurements, making the TLMs ideal for the early validation of multicore systems.

TLM-based validation approach

Figure 1 shows our TLM-based multicore design validation methodology. The input to the TLM generation process is the system definition, which consists of the application mapped to a multicore platform. As Figure 1 shows, both functional and timed TLMs are generated. The *functional TLM*, which does not include any timing, lets designers validate the task distribution on selected cores. The validation effort determines whether or not the distribution preserves the application execution semantics. The functional TLM also serves as a functional debugging platform.

The *timed TLM* provides accurate performance feedback that the designer can use to optimize the application, platform, or the mapping. The goals of automatic TLM generation are to speed up the generation and simulation loop, as well as to accurately estimate multicore design performance for an early and reliable design space exploration. The scientific challenges are to define TLM semantics and automatic TLM generation methods in order to meet these goals.

Multicore design validation approaches can be evaluated on the basis of speed, accuracy, abstraction level, and retargetability. In their research,[2] Russell and Jacome proposed a static estimation based on a designer-specified evaluation scenario. However, the estimation is not specific to input data and is not applicable to custom-hardware processing elements (PEs). Software performance estimation techniques claim to provide estimation at the transaction level, but they do not consider the processor data path structure.[3,4] Unlike the aforementioned techniques, Lee and Park's method can take the data path structure into account,[5] but the generated models are slow. Fast system models using binary translation compromise on accuracy and retargetability[6] (also see http://www.vastsystems.com/solutions-architecture-systems.html).

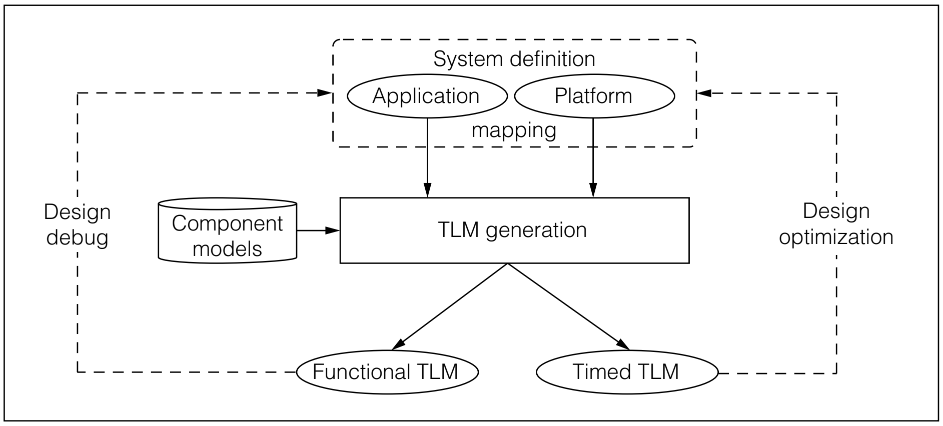

Figure 1. A transaction-level model (TLM)-based multicore design flow starting from a mapping of the application model to the multicore platform.

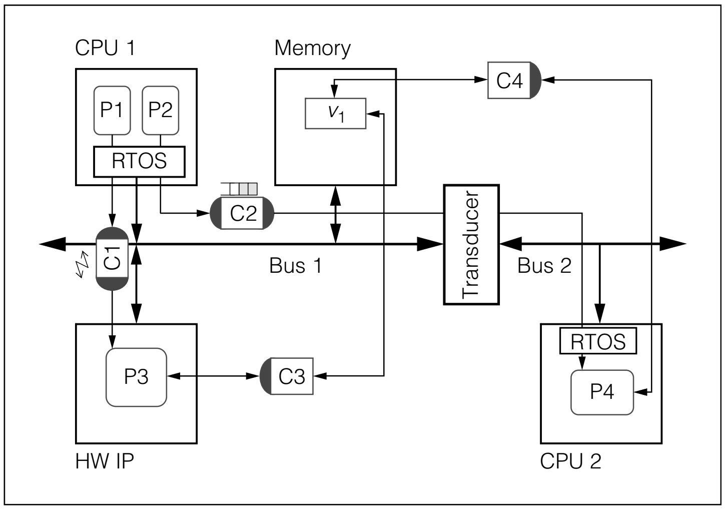

Figure 2. Multicore system specification as an overlay of the application model on the platform. The model consists of concurrent tasks (P1 to P4), communicating with one another using shared variables (v_1) and channels (C1 to C4). Tasks are symbolic representations of functions specified in C or C++.

Multicore system definition

The key elements defining a multicore system are the application model, the platform definition, and the mapping.

Application model

The application model is a platform-independent specification of the system's functionality. This model consists of concurrent tasks (P1 to P4 in Figure 2), communicating with one another using shared variables (v_1) and channels (C1 to C4). Tasks are symbolic representations of functions specified in C or C++. The C or C++ code for the tasks must be free of any platform-specific services to enable true orthogonality of the application and platform. The shared variables and channels are symbolic representations of the communication between concurrent tasks. The application model uses three channel types with distinct communication semantics:

- rendezvous synchronization (C1),
- blocking FIFO (C2), and
- nonblocking access to shared variables (C3 and C4).

All channels are point-to-point connected and lossless.

These modeling constructs enable development of a wide variety of application specifications. The application models can be modified independently of the hardware or software platform, in accordance with our platform-based design methodology. However, such independence also requires the task code to be completely free of any direct manipulation of physical addresses. For instance, task code cannot directly modify memory-mapped registers in hardware peripherals, because it would dictate a mapping to PEs and addresses. By using shared variables and channels instead, this register concept can be modeled through the shared variables, which will eventually be mapped to hardware PEs. The task code then manipulates the variables using the channels connected to those variables.

Platform and mapping

Figure 2 shows a typical multicore platform consisting of PEs, such as CPU cores (CPU 1 and CPU 2), custom-hardware components (HW), and IP blocks (IP). Shared memories may be instantiated as standalone components (memory), or they can be instantiated inside PEs. The communication architecture of the platform consists of two buses (bus 1 and bus 2) connected by a transducer component. Buses are generic elements that can represent a shared bus, point-to-point connection, or network links.

Transducers can create routes between PEs that are not directly connected with a bus. For example, in Figure 2, the route from CPU 1 to CPU 2 can be defined as *Bus 1 → Transducer → Bus 2*. The software platform is defined by a real-time operating system (RTOS) instantiation inside the CPU cores. The platform elements are instantiated from a library that contains data models of the available hardware and software components. The data models are used to check the validity of the platform composition—for instance, the compatibility of the core to the protocol of the connected bus—and for generating the executable TLMs.

The problem of mapping the application to the platform has been widely studied and is related to the TLM generation problem. Indeed, automatic TLM generation provides early and accurate feedback for making reliable mapping decisions.

Figure 2 shows a possible mapping as an overlay of the application model on the platform. This mapping assigns application tasks to CPU or hardware cores, channels to buses or routes, and shared variables to shared memories. Multiple concurrent tasks may be mapped only to CPUs that host an RTOS. On each bus, a unique address space is allocated for each channel mapped to the bus. We do not explicitly model the program or data accesses in our host-compiled TLMs; therefore, the task address space is not specified. The mapping must also be complete: that is, all the application objects must be mapped.

Functional TLM

Figure 3 shows the semantic structure of the functional TLM's elements. We define the functional-TLM semantics using SystemC.[7] Application tasks are modeled by `sc_thread` wrappers around the task functions. PEs are modeled as instantiations of `sc_modules` that wrap application tasks, a model of the interprocess communication (IPC) part of the RTOS, and a hardware abstraction layer (HAL). Buses are modeled as `sc_channels` with well-defined methods for synchronization, arbitration, and memory access. Shared memories are modeled as `sc_modules` containing a memory buffer and a bus interface thread. Multiple buses can be connected to each other using transducer `sc_modules` that contain a buffer for holding the communicated data, corresponding to each application-level channel, and an `sc_thread` for interfacing to the buses and controlling buffer access.

In the untimed functional TLM, only the IPC part of the RTOS is modeled: it's modeled as a channel that wraps all local application-level channels between tasks mapped to the CPU. The dynamic scheduling of tasks is then handled by the SystemC kernel. The HAL is modeled as an `sc_channel` that implements the abstract communication methods for the application-level channels atop the platform-specific bus channels. Therefore, the communication interface remains unchanged for the application tasks.

Functional-TLM generation

Once the TLM semantics are defined, we generate models of the application tasks, PEs, RTOS, and shared-memory. The generation of the bus, transducer, and HAL models, however, requires several additional steps.

Bus and transducer model generation. For each bus in the platform, a unique bus channel implementation in SystemC is generated according to a well-defined template.[8] The high-level bus channel provides methods for blocking transactions (send/receive) between two PEs and nonblocking memory

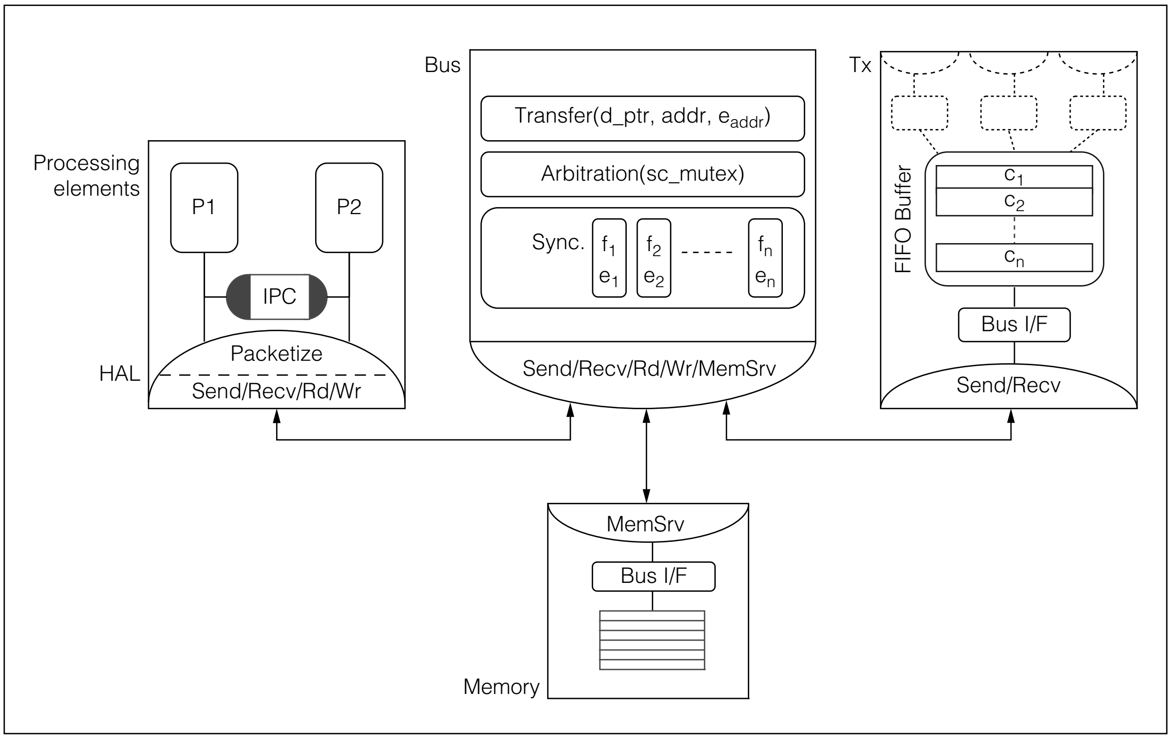

Figure 3. Semantic structure of a functional TLM. (HAL: hardware abstraction layer; IPC: interprocess communication; P: concurrent task.)

access (read/write) from PEs. Shared-memory modules expose their local buffers by calling the memory service function provided by the bus channel.

The memory access methods are generated as follows. For a shared bus, arbitration is modeled via an `sc_mutex` channel within the bus channel. A bus-accessing task on the master PE acquires the mutex and sets an address variable in the bus channel. The task then notifies all slaves with an address-set event e_{addr}. The memory service method, sensitive to this event, wakes up and loads or stores the data addressed by the address variable into or from the bus data pointer for a read/write operation.

Blocking transaction methods are built on memory access methods by adding a synchronization step. Synchronization is modeled using a flag-and-event pair for each blocking application channel mapped to the bus. Synchronization semantics require that the task on the slave PE indicate its readiness by setting the synchronization flag and notifying the synchronization event. When the task on the master side enters the transaction and the flag is not set, the task waits for the event and subsequently resets the flag. The aforementioned functional-TLM semantics ensure that tasks are synchronized before the transfer is initiated.

A transducer module translates between two or more protocol-incompatible buses. A transducer model consists of a controller task for each bus interface and FIFO buffers—one buffer for each channel routed over this transducer. Each controller task waits for a data request on its own bus and forwards any incoming request to the corresponding FIFO buffer for transmission. The controller tasks behave like any other application task for the purposes of synchronization, arbitration, and data transfer on the bus channel.

HAL model generation. The bus channels and transducer modules are sufficient for providing an end-to-end communication service for any two tasks in the multicore system. However, the application channel methods must be implemented atop the bus channels and transducers in the TLM. Specifically, the HAL handles the routing and packetization of application channel data. The HAL is modeled using `sc_channels` and is instantiated inside the PE modules.

Table 1. Functional-TLM generation results.

Design	Cores	TLM code size (lines of code)	Generation time (s)	Simulation time (s)
JPEG	2	897	3.23	0.010
	3	1,170	3.48	0.012
	4	1,443	3.98	0.015
	5	1,716	4.14	0.023
	6	1,989	4.26	0.029
MP3	2	2,894	6.60	0.03
	3	3,148	6.68	0.12
	5	3,653	7.42	0.55
H.264	2	868	4.87	2.601
	3	994	5.39	3.261
	4	1,120	6.59	3.429
	6	1,408	7.55	3.605

For each application channel to be implemented in the HAL, we first identify the route to which the channel is mapped. If the route contains transducers, we assign the packet size to be the lowest buffer size allocated for the route among all the transducers. A loop is then generated inside that HAL to slice the application data into packets. Each packet is sent to the receiver PE or the first transducer, as the case may be, using the address of the channel on the first bus segment. For rendezvous channels, a separate acknowledgment packet is sent from the receiver to the sender to preserve the application execution semantics.

Functional-TLM results

Here, we present results we obtained for functional-TLM generation using three industrial benchmark applications: JPEG encoder, MP3 decoder, and H.264 decoder, as Table 1 shows. The reference C model sizes, in lines of code, were 2,000 for the JPEG encoder, 12,000 for the MP3 decoder, and 7,000 for the H.264 decoder. The JPEG encoder had six tasks that could be pipelined and therefore executed on as many as six cores. We used a Xilinx MicroBlaze soft-processor core as the software PE.

The MP3 decoder has two concurrent streams of decoding, with each stream having computationally intensive functions for discrete cosine transform (DCT) and inverse modified DCT (IMDCT). In the two-core MP3 design, we mapped the DCT function from one stream to a custom-hardware core, with the rest executing on the MicroBlaze. In the three-core design, DCT and IMDCT functions from one stream were mapped to hardware cores. Finally, in the five-core design, DCT and IMDCT functions from both streams were mapped to hardware cores.

The H.264 is a dataflow model, similar to the JPEG, with six tasks, although with high computational complexity and larger inputs. Similar to the JPEG, the H.264 application could be mapped to as many as six MicroBlaze cores.

TLM quality. The functional TLM is primarily intended for early software development and debug. Therefore, it must compile and execute natively on the host machine at speeds similar to execution speed of the application's reference C model. We validated the TLM's functional correctness by comparing the TLM simulation results with the application reference code execution. The execution speeds of JPEG, MP3, and H.264 reference C models were 0.01 seconds, 0.01 seconds, and 2.534 seconds, respectively. Table 1 shows that the multicore TLMs' simulation speed was close to these numbers. Since the TLMs can be generated on the order of a few seconds, the modify-generate-execute cycle of design development was extremely fast, even for complex multicore platforms.

Productivity gains. The obvious advantage of automatic TLM generation is the savings in manual model development time. As Table 1 shows, the size of the SystemC TLMs (excluding the application C code) for multicore designs could be well over a thousand lines of code. This code may take several days, if not weeks, to develop and debug. With our automatic TLM generation, the code was generated in under a minute, and any modifications to the design were reflected in the TLM almost immediately. We conclude that this can lead to significant savings in multicore design and validation time.

Timed TLM

The top-level structure of the timed TLM is similar to that of the functional TLM that Figure 3 shows. In the timed TLM, however, timing delays are annotated to the application tasks and the bus channel methods. The C or C++ code for the application tasks is cross-compiled for the low-level virtual machine (LLVM), which is an open-source compiler infrastructure.[9] SystemC wait calls are added to the basic blocks of the LLVM assembly to incorporate the basic-block delays into the TLM. The timing-annotated code is then disassembled and compiled for the host machine.[10] Consequently, the application tasks in the timed TLM could lose their internal structure, although the functionality stays the same. As such, our timed TLMs are well-suited for performance estimation but not for source-level debug.

Timed-TLM generation

During timed-TLM generation, the cycle-approximate execution delay of basic blocks in the application tasks is estimated. This computation estimation is made on the basis of the PE's data path, memory hierarchy, and branch prediction. Communication estimation is done using the bus protocol specification.

Platform data modeling. Our PE model is generic enough to model most embedded CPU cores and custom-hardware components. The PE data model contains

- the PE data path definition, including pipelining;
- stochastic hit rates of the caches and memory access delays; and
- the stochastic branch misprediction rate.

In the PE data path definition, we define a mapping from the LLVM instructions to the PE data path components. For example, in the MicroBlaze data model, the LLVM add instruction is mapped to three pipeline stages: instruction fetch (IF), instruction decode (ID), and execute (EX). In the EX stage, the function unit used is the arithmetic logic unit (ALU) and the execution mode is addition. Furthermore, the delays (in cycles) are defined for all modes of each data path unit.

Custom-hardware PEs can be modeled if the data path is known a priori. The basic-block dataflow graphs of the task implemented in hardware are scheduled statically for the given data path to determine the basic-block execution cycles. This approach allows our technique to be retargetable for both CPU and custom hardware PEs. However, a mapping from the LLVM instructions to the PE data path must exist.

In the stochastic-memory model, we define parameters *instruction-cache miss rate* (IMR) and *data-cache miss rate* (DMR) for all possible cache sizes and configurations allowed by the CPU. We also define the *cache delay and main memory access delays* (MMDs). The stochastic branch model of the PE stores the branch penalty (BP) and the *branch misprediction rate* (BMR).

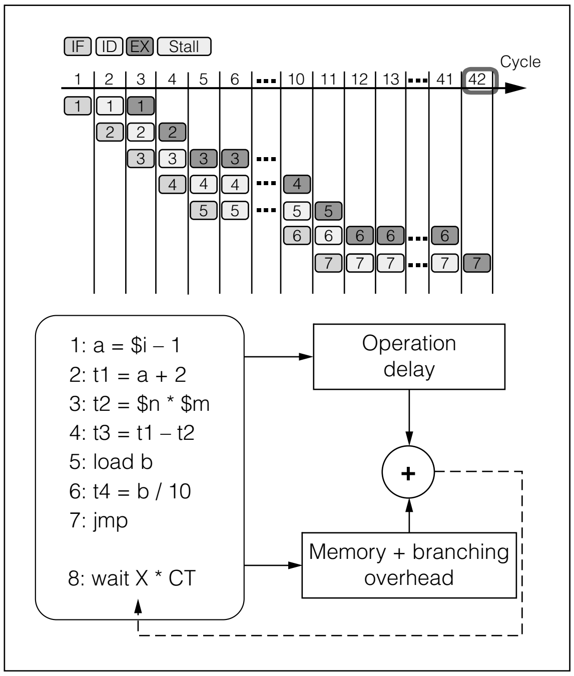

Figure 4. The computation timing estimate consists of the operation delay obtained via pipeline scheduling and the memory or branching overheads obtained from the stochastic models. Timing estimates are annotated to the basic block. Instructions shown in the basic block are LLVM instructions. (EX: execute; ID: instruction decode; IF: instruction fetch; X: basic-block cycles; CT: cycle time.)

Basic-block delay calculation. We calculate the execution delay of basic blocks by adding the results of basic-block operation delay, memory access overhead, and branch misprediction overhead, as Figure 4 shows. The top half of Figure 4 shows the result of scheduling a given basic block on a pipelined PE core.

We make some basic assumptions for scheduling. The PEs are assumed to be in-order, single-issue processors. Therefore, the operations are issued one per cycle, in the same order they are specified in the basic block code. Another important assumption during operation scheduling is that all memory accesses are completed in one cycle. The operation

Table 2. Timed-TLM generation, simulation, and estimation results.

Design	Cores	Generation time (s)	Simulation time (s)	Estimated cycles (millions)	Estimation error (%)
JPEG	1	4.53	0.034	1.029	–7.63
	2	5.77	0.039	0.566	–1.98
	3	5.90	0.049	0.493	10.85
	4	5.95	0.055	0.413	2.59
	5	5.99	0.059	0.474	–2.38
	6	6.16	0.066	0.557	–1.27
MP3	1	31	11	4.99	–13.89
	2	50	22	4.74	–8.29
	3	47	25	4.17	1.57
	5	71	36	4.05	2.29

labels (1 through 7) identify the operation's progress in the PE pipeline.

The legend at the top left of Figure 4 indicates each respective pipeline stage: fetch, decode, and execute. If some operations take longer than one cycle, the subsequent operation will be stalled in the pipeline. For example, the multiplier takes 5 cycles, so operation 3, which is a multiplication, causes subsequent operation 4 to be stalled until the execution of operation 3 is completed in cycle 9. Similarly, operation 6, which is division, takes 30 cycles, therefore stalling the execution of operation 7 until cycle 42.

Assuming cache access delay to be one cycle, we can say that each instruction cache miss results in $(1 + MMD)$ cycles of overhead. This is because the instruction must be fetched from the main memory and then read from the cache. Therefore, according to the stochastic model, each instruction, on average, has an overhead of $IMR(1 + MMD)$. Similarly, the data access overhead depends on the number of loads and is $\#Loads[DMR(1 + MMD)]$. The branch misprediction overhead is calculated as $BMR(BP)$. Finally, the basic-block cycles are computed by adding the operation delay and the overheads. The real delay can be obtained by multiplying the number of cycles to the PE cycle time.

Timed-TLM results

Here, we present results obtained for timed-TLM generation and simulation time using the benchmark JPEG and MP3 designs that we used (and described earlier) for functional-TLM generation.

We compared the timed TLMs to corresponding implementations on a Xilinx Virtex-II FPGA board for speed and accuracy. The H.264 design had to be excluded because of onboard memory limitations. The JPEG designs fit completely on the FPGA chip because their memory requirement is sufficiently small. We had to use off-chip memory for the MP3 decoder designs, and we used an instruction cache of 32 Kbytes and a data cache of 16 Kbytes. When multiple tasks were mapped to the same core, we statically scheduled the tasks on the core to avoid the need for an RTOS.

Speed. Table 2 shows the results for timed-TLM generation and simulation time. Clearly, the generation time for timed TLM exceeded that for functional TLM because of the additional basic-block analysis step. The timed-TLM simulation was also slower than the functional-TLM simulation by a multiple of three to five due to the additional SystemC wait executions at the basic-block level. However, the timed-TLM simulations were still several orders of magnitude faster than full-system RTL simulation, which took between 15 and 18 hours, and instruction-set simulation, which took more than 3 hours.

Accuracy. In Table 2, we show the estimated execution cycles, in millions, for JPEG encoding of a bitmap image and the MP3 decoding of a 136-Kbyte MP3 file. The estimation error is shown with respect to the reference board designs, where a negative error indicates an overestimation by the TLM.

For the JPEG designs, the average accuracy of TLM estimation was 4.5%, with the worst-case error being less than 11%. For the MP3 decoder designs, the average error was less than 7%, with the worst-case error being less than 14%. The estimation errors were due to the fact that the LLVM bytecode does not incorporate processor-specific compiler optimizations. This fact, on the other hand, allows our technique to be retargetable. Therefore, the results indicate that our automatically generated timed TLMs are both fast and accurate.

Timed TLM with timed RTOS

To explore dynamic-scheduling effects, we integrated an abstract RTOS into the timed TLM.

Each mapped application task becomes a virtual process on the abstract RTOS. The RTOS model and the virtual processes are configurable in their scheduling parameters, letting us validate the software configuration (e.g., scheduling-policy selection, and priority distribution). Our RTOS model supports multicore architectures, where each core executes its own RTOS instance with statically mapped application tasks.

RTOS model semantics

The abstract RTOS executes as an `sc_thread` inside a PE. The RTOS provides services to start, stop, and control virtual processes inside its context. It communicates with other PEs through the bus channel. The RTOS model uses `pthreads`, native to the host operating system. Each virtual timed process is executed in its own `pthread`. The execution of each `pthread` is controlled by the RTOS model through condition variables to emulate the selected scheduling policy. At any given point, only one `pthread` for each PE is released through the condition variable, thereby overwriting the host's scheduling policy with that of the target RTOS.

The abstract RTOS maintains a *task control block* for each virtual process. Each virtual process is scheduled according to a task state machine, with states such as running, ready, pending, and suspended. Each primitive in the TLM, which could potentially trigger a context switch, is executed under RTOS control. These primitives include task control and interprocess transactions.

RTOS timing-overhead model

An RTOS overhead delay model is needed to provide feedback about the overhead in a multitasking application, capturing the delays due to IPC, context switching, and preemption. Modeling such overheads is essential to guide the developer in partitioning the code (e.g., for deciding data granularity and communication handling).

We have developed a time-stamping approach to analyze RTOS overheads on the RTOS API level without source code analysis.[11] We characterize an RTOS on the actual processor in supported configurations, and the determined overhead characteristics are stored in an RTOS library. The TLM generator reads the library to instantiate an abstract RTOS inside a PE model with specific delay parameters.

In our experiments with timed RTOS, we developed a special test application that captured time stamps and invoked RTOS primitives in a controlled environment in which we knew the scheduling outcome beforehand. We used an external timer to measure time. The time stamp code and its data were exclusively placed in a noncached fast local memory to minimize impact on caching and execution time. We disabled timer interrupts while analyzing timing unrelated RTOS primitives to eliminate the impact of unexpected interrupts.

Although our analysis and modeling approach abstracted away many influences on RTOS overhead (e.g., the number of total, waiting, and manipulated tasks; and scheduler implementation) and therefore was not cycle-accurate, it has already yielded valuable feedback for estimating system performance.

Timed-RTOS model results

We applied our timed RTOS modeling approach to three benchmark designs based on the JPEG encoder, the MP3 decoder, and a combined design (MP3 + JPEG) running both applications. The MP3 decoder was modeled with three concurrent tasks, and the JPEG encoder with five concurrent tasks. All tasks were mapped to a MicroBlaze processor running at 100 MHz, and scheduled by the RTOS Xilkernel.

To fit the design on the board, we reduced the MP3 decoder test data to a single frame. Three kinds of TLM were generated:

- TLM with no RTOS instantiated concurrent SystemC threads and relied on the SystemC scheduler to perform dynamic scheduling.

Table 3. Timed-RTOS (T-RTOS) TLM simulation and estimation results.

Design	Model type	Simulation time (s)	Estimated time (s)	Estimation error (%)
JPEG	No RTOS	0.02	0.21	–75.00
	With RTOS	0.25	0.53	–35.56
	With T-RTOS	0.27	0.75	9.98
	XVP	168.00	1.25	50.00
MP3	No RTOS	0.01	0.20	–41.00
	With RTOS	0.08	0.25	–25.95
	With T-RTOS	0.08	0.36	5.29
	XVP	60.00	0.37	7.00
MP3 + JPEG	No RTOS	0.04	0.20	–83.00
	With RTOS	0.32	0.79	–32.25
	With T-RTOS	0.33	1.10	–6.20
	XVP	213.00	1.60	37.00

* RTOS: real-time operating system; T-RTOS: timed-RTOS; XVP: Xilinx Virtual Platform.

- TLM with an untimed RTOS used an RTOS model without the overheads incorporated.
- TLM with timed RTOS incorporated all the RTOS overhead delays.

We used the Xilinx Virtual Platform (XVP) as an industrial model to provide comparison points for the speed and accuracy of the TLMs.

Speed. Table 3 shows the results of model simulation for the benchmarks just discussed. The TLM generation time is not noted here, because it was only a marginal increase over the timed-TLM generation. Clearly, increasing the model complexity increases simulation time. The timed TLM with no RTOS executed the fastest (a few milliseconds) for all designs because the task switching is handled by the SystemC simulation kernel.

The timed TLM with an integrated, untimed RTOS model ("With RTOS" in Table 3), which modeled the virtual threads, executed in fractions of a second. No significant increase in simulation time was measured for timed TLM with timed RTOS ("With T-RTOS" in the table). Note that the timed TLM with timed RTOS executed about three times faster than the actual time required by the equivalent design on the board. Therefore, our host TLM was more efficient for validation of our designs than onboard testing.

Comparisons with the XVP model are even more favorable to the TLMs. The XVP executed the same multithreaded applications on the Xilkernel at speeds of more than two orders of magnitude slower than the TLMs. This was because the application binary and the Xilkernel libraries were interpreted on the MicroBlaze instruction set simulation (ISS) model inside the XVP. Again, the host-compiled TLMs were shown to be more efficient than cycle-level ISS models.

Accuracy. The timed TLM without the RTOS model showed an average error rate of 66%, and up to 83%, depending on application parallelism. The underestimation of execution time was because the individual threads executing on the PE model accrued time in parallel. From the perspective of the SystemC scheduler, these threads were independent. In reality, however, the threads were dynamically scheduled on the PE. Adding dynamic scheduling by an RTOS model dramatically improved accuracy. However, with the fine-grained IPC, the designs exhibited significant system overhead and thus the TLM with untimed RTOS ("With RTOS" in Table 3) underestimated the execution time by 32% on average. This result can be compared to other state-of-the-art solutions, none of which model any RTOS overhead.

Adding RTOS overhead modeling reduced the error rate to less than 10%, yielding already sufficiently accurate timing information. The remaining error was due to our abstract analysis and modeling of RTOS overheads, which we chose to automate the TLM generation and support high simulation speed.

Finally, in our experiments we compared the TLM accuracy to that of the commercial XVP. The XVP overestimated the application execution time by 31% on average. These estimation errors can be traced back to inaccurate modeling of memory accesses in XVP.[6] Comparing all solutions, our TLM with timed RTOS yielded the most accurate timing estimation.

For future work, we will investigate fast cache modeling to accurately capture the dynamic effects of caches. Furthermore, the RTOS models will be extended to symmetric multicore architectures, in which multiple cores may share the same address space. ■

Acknowledgments

We thank Pramod Chandriah and Yonjin Ahn for providing the JPEG, MP3, and H.264 reference models. We also thank the reviewers for their detailed feedback.

■ References

1. D.D. Gajski et al., *Embedded System Design: Modeling, Synthesis and Verification,* Springer, 2009.
2. J.T. Russell and M.F. Jacome, "Architecture-Level Performance Evaluation of Component-Based Embedded Systems," *Proc. 40th Design Automation Conf.* (DAC 03), ACM Press, 2003, doi:10.1145/775832.775936.
3. C. Brandolese et al., "Source-Level Execution Time Estimation of C Programs," *Proc. 9th Int'l Symp. Hardware/ Software Codesign* (CODES 01), ACM Press, 2001, doi:10.1145/371636.371694.
4. T. Kempf et al., "A SW Performance Estimation Framework for Early System-Level-Design Using Fine-Grained Instrumentation," *Proc. Design, Automation and Test in Europe Conf.* (DATE 06), IEEE CS Press, 2006, doi:10.1109/DATE.2006.243830.
5. J.-Y. Lee and I.-C. Park, "Timed Compiled-Code Simulation of Embedded Software for Performance Analysis of SOC Design," *Proc. 39th Design Automation Conf.* (DAC 02), ACM Press, 2002, doi:10.1145/513918.513994.
6. Xilinx, *Embedded System Tools Reference Manual,* 2005.
7. T. Grötker et al., *System Design with SystemC,* Kluwer Academic Publishers, 2002.
8. L. Yu, S. Abdi, and D. Gajski, *Transaction Level Platform Modeling in SystemC for Multi-Processor Designs,* tech. report CECS-07-01, Center for Embedded Computer Systems, Univ. of California, Irvine, 2007.
9. LLVM (Low Level Virtual Machine) Project, "The LLVM Compiler Infrastructure," v. 2.8, 2010; http://www.llvm.org.
10. Y. Hwang, S. Abdi, and D. Gajski, "Cycle-Approximate Retargetable Performance Estimation at the Transaction Level," *Proc. Design, Automation and Test in Europe Conf.* (DATE 08), EDAA, 2008, pp. 3-8.
11. Y. Hwang, G. Schirner, and S. Abdi, "Automatic Generation of Cycle-Approximate TLMs with Timed RTOS Model Support," *Proc. 3rd IFIP TC 10 Int'l Embedded Systems Symposium* (IESS 09), Springer, 2009, pp. 66-76.

Samar Abdi is an assistant professor of electrical and computer engineering at Concordia University, Canada. His research interests include modeling and validation of multicore embedded systems. He has a PhD in information and computer science from the University of California, Irvine. He is a member of IEEE.

Yonghyun Hwang is a senior engineer at Qualcomm. His research interests include SoC modeling and real-time operating systems. He has a PhD in information and computer science from the University of California, Irvine.

Lochi Yu is an assistant professor of electrical engineering at the University of Costa Rica. His research areas include modeling of embedded systems and brain computer interfacing. He has a PhD in electrical and computer engineering from the University of California, Irvine. He is a member of IEEE.

Gunar Schirner is an assistant professor of electrical and computer engineering at Northeastern University. His research interests include embedded-system modeling, system-level design, and embedded-software synthesis. He has a PhD in electrical and computer engineering from the University of California, Irvine. He is a member of IEEE.

Daniel D. Gajski is a professor of electrical and computer engineering and director of the Center for Embedded Computer Systems at the University of California, Irvine. His research interests include are in design methodologies for embedded systems. He has a PhD in computer and information sciences from the University of Pennsylvania. He is an IEEE Fellow.

■ Direct questions and comments about this article to Samar Abdi, Dept. of Electrical and Computer Eng., Concordia University, 1515 St. Catherine West, S-EV005.183, Montreal, Quebec, Canada H3G 2W1; samar@ece.concordia.ca.