A Prototype Software Tool for Recording and Playback of API-level Function Calls in a Cycle-Accurate Simulator

By

Mario Guerra

CSE-593 Applied Project

Fulton Schools of Engineering

Arizona State University, Tempe, Arizona, SUA

Supervised by

Hessam S. Sarjoughian

May 2012
Table of Contents

Abstract ............................................................................................................................. 3
Introduction ...................................................................................................................... 4
Recording and Playback of System Platform Simulations ............................................. 9
Design and Implementation ............................................................................................ 12
API Instrumentation and Transaction Record Format .................................................. 14
Time Stamping of API Events ....................................................................................... 16
Playback of Transaction Record .................................................................................... 19
Feature Validation ........................................................................................................ 22
Conclusion ..................................................................................................................... 29
References .................................................................................................................... 30
Appendix I - Future Work .............................................................................................. 30
Appendix II – Cycle-accurate versus Event-based Simulation ....................................... 31
Appendix III – Interaction Between Models in a System Simulation Platform ............. 31
Appendix IV – Sample Transaction Record .................................................................. 32

List of Figures

Figure 1 - Simplified diagram of a typical system simulation platform ....................... 5
Figure 2 - Expanded view of system simulation platform ............................................. 6
Figure 3 – DSP Simulator with API recording instrumentation .................................... 10
Figure 4 - Playback of API-level transactions on the DSPS ........................................ 11
Figure 5 - System simulation platform, revisited ....................................................... 27
Figure 6 - Playback utility, revisited ............................................................................ 29

List of Examples

Example 1 - Simple interaction between models ......................................................... 8
Example 2 - Template for transaction record entries .................................................. 15
Example 3 - Sample entry in transaction record .......................................................... 15
Example 4 - Sample lines from beginning of transaction record .............................. 15
Example 5 - "AddBusAccessCallback()“ instrumentation example .......................... 17
Example 6 - Interception callback for bus access callback handler ........................... 18
Example 7 – Example playback utility callback handlers .......................................... 20
Example 8 - Interaction between models and the quantum boundary ...................... 32
Abstract

This report discusses the design and implementation of a method for recording and playback of API-level function calls in a cycle-accurate DSP simulator, for use in reproduction and debug of failure conditions of the DSP simulator using a simplified test environment. When the DSP simulator is integrated into a more complex SystemC-based system simulation platform, the DSP simulator interacts with several other models such as an ARM simulator, bus model, and system timer model. These models interact with the DSP simulator in ways that quite often cannot be reproduced outside of the SystemC platform. Due to the dependency of the simulated software applications on these interactions between models, software bugs in the DSP simulator that manifest in this environment are very difficult to debug. The difficulty stems primarily from the complexity involved in installing and maintaining the platforms, which often require specialized knowledge and costly licenses. Plus, all of the various teams involved in development and use of these platforms are located in different geographical areas across the globe, making timely communication more challenging. Much valuable software development time is lost in trying to overcome these hurdles when attempting to investigate a potential bug in the DSP simulator model.

The goal of this project was to explore the possibility of reproducing the DSP simulator’s behavior when integrated in a larger system simulation platform, without the need to actually install and maintain the entire platform. All interactions with the DSP simulator and other models in the platform occur through a well-defined API that is part of the DSP simulator deliverable. From the perspective of the DSP simulator, it makes no difference as to what devices exist on the other side of the API boundary. Therefore, it is possible to present an abstraction to the DSP simulator that behaves like the rest of the platform. This abstraction was prototyped by instrumenting the API to record all interactions across the API boundary, with each API function instrumented to write an entry into an ASCII text file. The entry includes details on what function was called, when it was called in relation to the value of the DSP processor cycle counter, and what the values of any function-level parameters were at the time of the call.

A playback utility was written that provides the necessary system abstraction, and using the transaction record as a sequential list of steps to follow, the utility “replays” the original simulation by essentially controlling the DSP simulator with the same API functions that were called during the original recorded simulation. Using this record and playback methodology, it is possible to accurately reproduce the behavior of the DSP simulator without requiring the rest of the models used in the original platform.

Prototyping of this record and playback feature was done using a proprietary DSP simulator written in C/C++, compiled to run on a 64-bit SUSE Linux distribution. All code was compiled using the GNU C/C++ Compiler (GCC) version 4.1.2. Other platforms to be supported are listed in Appendix I, which discusses areas for future work.
Introduction

I am employed with a company that develops chips for use in the wireless communications industry, in a division that produces a digital signal processor (DSP) core. My role is to manage a team of software engineers. One of the main deliverables we develop is a cycle-accurate instruction-set simulator of the DSP core, coupled with an application programing interface (API) that can be used to extend the simulator’s functionality. The DSP simulator, referred to as the DSPS, and the API are implemented in C/C++. These products are supported on Windows 7 and the SUSE Linux distribution. The API allows the application of the simulator to be extended to suite different use cases, including the integration of the DSPS in larger system simulation platforms that model entire sub-systems of a system-on-chip (SoC) design.

These system simulation platforms allow application developers to begin optimizing software well in advance of actual silicon chips or even FPGA-based development boards being available. This reduces the time to market in a very competitive industry, which makes these platforms a very important part of my company’s overall software development strategy.

A typical platform consists of several simulator models provided by different design groups. These models are collected by a separate platform group and connected to each other either via one of two ways – using an integration methodology which adheres to the open SystemC standard\(^1\), or using a proprietary third-party system simulation tool such as Virtualizer\(^2\), developed by Synopsis. The Virtualizer tool is based on SystemC, but provides extensions to the standard to support a GUI-based simulation interface.

A simplified block diagram of an example platform is shown in Figure 1. It is important to note that although a typical platform consists of many more models than shown in the simplified diagram, the other models in the platform have no direct interaction with the DSPS. Therefore they are not included in the diagram for the sake of clarity.

---


The bus model is the primary method by which models communicate with each other in a system simulation platform. The bus model is intended to model a bus protocol, in terms of arbitrating bus requests between several master devices such as the DSPS and the ARM simulator. It also models bus access latency to more accurately reflect how the software would behave on real hardware. All models in the platform must conform to the Transaction-Level Modeling (TLM) standard\(^3\) by convention of the platform. Therefore, the DSPS is “wrapped” with a TLM interface by the platform integration team for compliance with the system platform requirements. Within this wrapper layer, DSPS API transactions are mapped to equivalent TLM transactions and vice-versa. Other models in the system may provide native TLM interfaces or may be wrapped with a TLM transaction layer similar to the DSPS.

A secondary method of communication between models in the system is through interrupt pins modeled as SystemC ports. The interrupt pins are essentially one-way communication, where one model can trigger an interrupt on another model via a write to the proper registers in the interrupt controller model. The interrupt controller model then asserts an interrupt on the relevant model’s interrupt pin. Acknowledgement of the interrupt is done through the bus model in the form of a read to the interrupt controller model.

Figure 2 provides an expanded look at how a system simulation platform is structured, with emphasis on how the DSPS interacts with the TLM wrapper. By extension, the DSPS interacts with the rest of the platform through this wrapper. The API layer provides the

---

interface by which the DSPS can be externally configured and controlled. Within the DSPS, the API layer interacts directly with the core simulator library routines using internal commands and interfaces.

**Figure 2 - Expanded view of system simulation platform**

There are three main types of functions in the DSPS API:

- **Simulator configuration** – Functions dedicated to configuration of simulator options prior to starting a simulation, such as core clock frequency, memory access tracing, and performance profiling. The end of configuration is marked by a call to a dedicated end-of-configuration API function. These functions may only be called once.

- **Simulator control** – Functions for runtime control of the DSPS and access to its internal memory and registers. Calls to control functions can be made only after the configuration phase is complete as marked by the call to the end-of-configuration API function. These functions can be called multiple times.

- **Callback registration** - Callback registration functions used by system simulation platforms to register for event notification callbacks with
the DSPS. All registration functions can be called before or after the configuration phase is complete. Each of the callback registration/unregistration functions can be called multiple times.

In the system simulation platform shown in Figure 2, interaction with the model and the TLM wrapper happens through two channels. The first is through the regular API interface in the form of API calls that take the form of one of the three types listed above.

The second type of interaction is through run-time callbacks. The callback registration portion of the API provides methods that specifically allow an external device model to register for a callback notification when certain pre-defined events occur. An example of a typical callback used in these platforms is a bus access request callback. When a device model registers for a bus access callback to a range of memory addresses, any instance of the DSPS attempting to access memory in the specified range will result in a callback notification being initiated to the device model. The device model will temporarily have control of the simulation in order to respond to the request.

In the platform shown in Figure 2, the TLM wrapper acts as a device model and registers for the bus access callback. When the DSPS attempts to access memory that is external to its own internal memory caches, a bus access callback notification is sent to the TLM wrapper in the form of a function call. The function call passes execution control from the DSPS to the TLM wrapper. The TLM wrapper uses the address and access type specified by the DSPS to initiate a TLM-style bus transaction request to the bus model. Execution control is then passed to the bus model, which handles the request appropriately. After a simulated time interval \( \Delta \), where other models in the system may be allowed to run, the bus model returns control to the TLM wrapper callback function along with the status of the request. The TLM wrapper then passes this status back to the DSPS by exiting the callback function and returning control to the DSPS.

To illustrate the complexity of modeling a transaction between devices in the platform, consider a common producer/consumer interaction between the ARM simulator and the DSPS provided in Example 1. The ARM simulator writes data to a shared memory buffer, and then signals the DSPS to read the data from the buffer. In this example, \( t_0 \) represents an arbitrary simulation point marking the beginning of a new transaction. \( \Delta_1 \) through \( \Delta_{18} \) represent increasing simulation time increments meant to convey the sequential order of events happening in the system. Given that the system simulation platforms operate on a cycle-driven basis, \( t_0 \) and \( \Delta_1 \) through \( \Delta_{18} \) represent discrete simulated clock cycle values.
Example 1 - Simple interaction between models

$t_0$: ARM simulator initiates a bus transaction request
$t_0+\Delta_1$: Bus model accepts request
$t_0+\Delta_2$: ARM simulator writes data to memory shared between ARM and DSP
$t_0+\Delta_3$: Memory device signals acceptance of write
$t_0+\Delta_4$: Bus model signals completion of bus request
$t_0+\Delta_5$: ARM simulator initiates another bus transaction request
$t_0+\Delta_6$: Bus model accepts request
$t_0+\Delta_7$: ARM simulator writes DSP interrupt enable to interrupt controller model
$t_0+\Delta_8$: Interrupt controller model signals acceptance of write
$t_0+\Delta_9$: Bus model signals completion of transaction
$t_0+\Delta_{10}$: Interrupt controller model asserts interrupt on DSPS
$t_0+\Delta_{11}$: TLM wrapper for DSPS asserts interrupt on DSPS using appropriate API
$t_0+\Delta_{12}$: DSPS initiates a bus transaction request
$t_0+\Delta_{13}$: Bus model accepts request
$t_0+\Delta_{14}$: DSPS initiates read of appropriate register in interrupt controller model
$t_0+\Delta_{15}$: Interrupt controller model signals acceptance of read request
$t_0+\Delta_{16}$: Interrupt controller model provides read data, signals completion of read
$t_0+\Delta_{18}$: Bus model signals completion of transaction

Even though Example 1 only illustrates one of the simplest interactions between models, modeling the interaction correctly still requires nineteen discrete steps. These nineteen steps may be repeated several hundreds of thousands of times in a single simulation run. More sophisticated interactions between models can require dozens, or even hundreds, of discrete steps to be modeled. Various software teams within the company use these platforms for early software development in advance of hardware availability for applications such as modem protocol stacks, audio processing topologies, and computer vision algorithms. All of these applications require complex interactions between models. If a bug exists in any of the models that handle the steps necessary to correctly simulate the interactions, tracking down the model with the problem can be exceedingly difficult.

When crashes or unexpected results occur during use of the platform by one of the internal software development teams, the platform integration team performs an initial analysis. If the problem is determined to be with the DSPS, then the DSPS team must take over to isolate and fix the potential bug.

Due to the interaction dependencies between models required to successfully simulate the software application, quite often a failure of the DSPS cannot be reproduced on the model in isolation from the rest of the system simulation platform. We must therefore attempt to debug the issue on the larger platform. This is a cumbersome exercise, since the platforms are typically difficult to install and complicated to run, often requiring
specialized knowledge from the team that developed the platform. The difficulty in installing and maintaining these platforms also precludes us from using them in our regular regression testing. This hole in the testing only serves to increase the likelihood of complex interaction bugs being found by internal software development teams rather than by our dedicated testing. When debugging of a DSPS problem in a system simulation platform is required, much time is spent in trying to get the platform installed and configured properly before any work on investigation can be done. This is time that could be better spent on more productive development work. A methodology for reproducing these types of DSPS failures outside of the system simulation platform is required.

Recording and Playback of System Platform Simulations

It is the API layer that allows the DSPS to be integrated into a larger system simulation platform. All interactions with the simulator must occur via this API layer, either through direct function calls or through registered callbacks. If it were possible to time stamp and record all transactions that cross the API boundary, then the recording could be used to replay the simulation scenario that caused the failure.

An illustration of this idea is shown in Figure 3. By instrumenting every method in the API and extending the API to include a callback-recording layer, it is possible to keep track of all interactions between the DSPS and the TLM wrapper.

Every time an API function is called, the instrumented function writes an entry to a file stored on the host system. The entry includes information specific to the function called, including the name, values of all relevant parameters, and a cycle stamp that marks the moment in simulation time that the function was called. This cycle stamp consists of the simulated processor cycle count at the time of the API function call, where a cycle represents one tick of the processor core clock. On real hardware, this processor core clock will run at 500MHz.

Callback registrations from the TLM wrapper are first relayed through a callback-recording layer, implemented as part of the API. The callback-recording layer intercepts callback notifications from the core DSP simulator library on the way to the TLM wrapper. When intercepted, the callback type is recorded as an entry in the transaction record file, along with a cycle-accurate time stamp. The eventual return from the callback is also recorded, since it also passes through the callback-recording layer.

Over the course of a simulation, an exact record of API transactions and callbacks that occur between the DSPS and the TLM wrapper can be constructed. For simplicity while prototyping, the transaction record is stored as an ASCII text file on the host system.
The API layer serves as a means of abstracting the DSPS from external devices, so from the perspective of the DSPS it makes no difference as to what is on the other end of the API layer. If an exact record of all transactions across the API boundary can be created, it is possible to use this record, along with the original software application image, to recreate the same simulation environment as the platform that generated the recording.

This type of simulation abstraction can be accomplished via a playback utility that creates an instance of the DSPS. The utility uses the transaction record to load the software image, configure the simulator, and register for the same callback notification events as was done in the original platform. When callbacks occur during the simulation, the utility handles the callback events exactly the same way they were handled during the original simulation. All of the required information on how to configure the DSPS and respond to callback events is contained in the transaction record.

Figure 4 illustrates how the playback utility interacts with the DSPS, essentially taking the place of all the system simulation platform elements in Figure 2 and using the transaction record to stimulate and control the DSPS. The main purpose of the playback utility is to read the list of API transactions from the transaction record and supply them
back to the simulator at the exact times they occurred in the original simulation as recorded via the cycle time stamp. The handling of callback notification events is also performed by the playback utility. Since the rest of the simulation platform has been abstracted away, the output of program data from the DSPS intended for other models can be safely ignored.

This record and playback feature potentially provides a way for the DSPS development team to efficiently reproduce and debug failures of the DSPS. Once the simulator has been instantiated by the playback utility, it can be debugged using standard debug tools such as the GNU debugger on Linux, or the Visual Studio debugger on Windows. It is important to note that in failure cases the transaction record would only contain a record of transactions up to the point of failure. However, this is more than enough information needed for debugging purposes.

In order for playback of the original simulation to be feasible, and to ensure the playback simulation is equivalent to the original simulation, the following conditions must be met:

- **Equivalent bring-up state** – When the core DSPS library is first instantiated as an object, the initial state must be the same every time.
- **Equivalent configuration** – The playback simulation must be configured exactly as the original prior to beginning the simulation.
- **Equivalent run-time behavior** – Given the same start state, the simulation must behave exactly the same in both environments.
• **Equivalent event notification/handling** – The same callback events must occur at the same time in both environments.

The DSPS is a cycle-accurate simulator meant to accurately model a DSP processor core. As such, it is designed to closely model the hardware it represents, including the initial power-on state of the processor core. Every register and state machine in the processor core has a known state when power is first applied, and the simulator model is designed to replicate this initial state when it is first constructed. Therefore, the first condition of equivalent bring-up state is met.

The requirement of equivalent configuration is met by the nature of the recording process itself. The DSPS records exactly how it is configured and controlled during a simulation. All the playback utility does is follow the same list of configuration options, callback registrations, and callback handling. Aside from the transaction record, the only other file needed to replay the simulation is the original software application image, which can be obtained from the software development team that originally reported the issue.

Given that the simulator is a cycle-accurate model of a DSP processor core with deterministic behavior, the simulator is also designed to behave in a deterministic way. The determinism inherent in the DSPS model means that given the same software image, the same initial state, and the same configuration options, the playback simulation will perform exactly as it did on the original platform. This also means that callback events on the playback simulation will occur on the same cycle as they occurred in the original simulation. The cycle-accurate model also conveniently sidesteps the problem of simultaneously occurring events – the events are processed serially within the simulator and therefore will be recorded in the same order that they are processed. A comparison between cycle-accurate versus event-based simulation is provided in Appendix II for reference.

**Design and Implementation**

The DSPS is designed as a library that can be dynamically loaded by a system simulator. It supports two simulation modes:

- Standalone simulation as a command-line tool
- Simulation as part of a larger system simulation environment.

The simulator library exports approximately 90 functions that can be called by the system simulation environment. These functions make up the DSPS Application Programming Interface. The original reason for the creation of the API was to allow the functionality of the standalone simulator to be extended through the use of co-simulation modules that are loaded by the simulator prior to beginning a simulation. Co-
simulation modules can be used to model functionality required by the application software that does not exist within the scope of the DSPS model. Some typical examples of co-simulation modules are:

- Bus co-simulator – Model bus transactions and memory access latencies.
- System monitor co-simulators – Monitor specific performance aspects of a simulation, such as cache hit/miss ratios.

Using co-simulation modules, it is possible to create a simulation environment that more closely represents actual hardware devices. However, the increased pressure to reduce the time-to-market for new devices has resulted in decreased use of co-simulation modules in favor of SystemC-based system simulation platforms that more accurately represent actual hardware. In this type of environment, only a subset of the full API is utilized, with perhaps 25-30 functions at most ever being used. However, a sizeable number of internal developers still make use of the API for development of co-simulation modules, and therefore the full API will continue to be supported in its current form for the foreseeable future.

The DSPS API is implemented entirely in a single C++ class. All methods are defined to be visible externally to any device that needs to interact with the simulator. In a system simulation platform, the TLM wrapper serves as the external device that interacts with the DSPS through the API. Through interaction with this single class, a system simulation environment can instantiate an instance of the DSP simulator, configure run-time performance settings, register for callbacks, and control the time the simulator is allowed to execute cycles.

Since the API is entirely contained in a single class, the work required to instrument the API for recording is primarily confined to the main *.cpp and *.h files which define the API class. Some side work in other files was required to support a command line switch that enables the recording feature and the addition of a few global variables. However, the majority of work is encapsulated within two files.

Each API method is instrumented to output a trace entry into a transaction record. The transaction record is an ASCII text file on the host system that is opened by the API class constructor. The constructor creates and sets a class-level file pointer that points to the transaction record. Using this file pointer, all functions within the API can access the transaction record. When recording is enabled and an API function is called, the first action that is taken on entry to the function is a write of the function name, cycle time stamp, and function parameter values into the transaction record.

Similarly, callback events that are intercepted by the callback-recording layer are also recorded with a write of the callback name, cycle time stamp, and function parameters into the transaction record before the callback call is passed on to the TLM wrapper. If the TLM wrapper invokes any of the API methods during a callback, these methods are
also recorded as described above. When the callback eventually returns control to the simulator, simulation control first flows back through the callback-recording layer and the return value of the callback is recorded at this time.

The DSPS is designed as a single-threaded program, so all calls to API methods are processed serially. There is no risk of API calls overlapping each other in time, so the recording of transactions is deemed reliable in terms of accurately representing the order and timing, in cycle counts, of API function calls and callback notifications.

Currently, the transaction record consists of a single file with the name hard-coded to “APIRecord.out”. A new file is created for each simulation, overwriting any existing transaction record. No account is taken for system simulation platforms that make use of more than one instance of the DSPS. Although this scenario is not yet in use within the company, the results would be undefined if the recording functionality were to be used this way. Future work on this project will explore the following options regarding the implementation:

- Allow specification of a custom name for the transaction record
- Avoid overwriting of existing transaction records through the use of a numerical naming convention
- Provide support for recording transactions of multiple DSPS models in the same platform

In addition to the API instrumentation, a playback utility was written, primarily in C, to parse the transaction record and reproduce the recorded simulation. The playback utility also provides dummy callback handlers that are used to handle callback notification events that occur during playback of the simulation. I will describe implementation details of the API instrumentation and the playback utility in more detail over the course of the following sections.

**API Instrumentation and Transaction Record Format**

The prime factor motivating the choice of format for the API recording was ease of debugging, for both the initial development of the feature and investigation of DSPS failures once the feature is deployed. In both cases, a format that is easy to decipher with entries that are easily matched to their corresponding API methods is an absolute necessity. This need drove the choice of an ASCII text file as the format, with each API method being assigned a short identifier consisting of an abbreviated form of the method name. The API functions all have different numbers and format of arguments, but for simplicity the recorded argument values are printed in hexadecimal format for numerical values and string format for alphanumeric values such as system paths and file names. The basic template for an entry in the transaction record is as follows:
Example 2 - Template for transaction record entries

<API name> <cycle> <arg1> <arg2> <arg3> <etc>

For example, the API method used most frequently in a system simulation platform is named “AddBusAccessCallback()”, which allows the TLM wrapper to specify an address range and a callback function that is invoked whenever the DSPS attempts to access an address in external memory that falls in the specified range. The abbreviated identifier for the bus access callback registration function is “AddBusClbk”, and this string is written to the transaction file whenever the “AddBusAccessCallback()” function is invoked. A sample of what this entry may look like in a transaction record is shown in Example 3.

Example 3 - Sample entry in transaction record

AddBusClbk 0 0 ffffffff

From left to right, Example 3 shows a call made to “AddBusAccessCallback()”, at cycle 0, for the full 32-bit address range starting at address 0x0 and ending at address 0xffffffff.

For the sake of illustration, I have included the first ten lines of a sample transaction record generated from a test below.

Example 4 - Sample lines from beginning of transaction record

Constructor 2105
ConfExecBin ./hello_world
ConfBusPen 186a0 4
ConfAppCmdLine 1
ConfTimeMode 2
Cycle2Time 0 32
ConfStatFile stats.txt 2
EndConfig
AddBusClbk 0 0 ffffffff
Run 0

As can be seen from the example events list, the first item in the list is a “Constructor” event. All transaction records, regardless of the eventual sequence of events, will begin with a “Constructor” event since this is the point where the simulator library gets instantiated as a runtime object. This is also the point at which recording begins for any simulation. Consequently, all successfully completed recorded simulations should generate a “Destructor” event at the end of the simulation when the simulator runtime object is destroyed. Transaction records that represent a DSPS simulation failure may not necessarily include the “Destructor” event, since the possibility of abnormal system exit during a failure would prevent the call to the destructor from occurring.
Given that the format of the transaction record is ASCII text rather than some kind of compressed format, I was initially concerned that the size of the file might make it unworkable for some customer scenarios. Although the cost of disk space is minimal, copying files in excess of 1Gb across a network can still be cumbersome. During testing with an actual customer platform, the size of the transaction record rarely exceeded 200Mb. These files, while large, are still manageable in terms of consumed disk space and time spent copying these files across a network. However, the test simulations I ran to prove this concept are at least one or two order of magnitude shorter in terms of actual simulated cycles, and therefore actual customer files are likely to be much larger. Future work in this area will investigate the use of compression to reduce the size of the transaction record, as well as decompression to restore the transaction record to a human-readable format.

A more extensive sample of a transaction record is included in Appendix IV for reference.

**Time Stamping of API Events**

When integrated into a system simulation platform, interaction with the DSPS via the API occurs in two distinct phases, with the phases divided by the start of the simulation. Prior to beginning the simulation, execution control resides with the TLM wrapper and by extension the system simulation platform. During this time, all calls to the DSPS API are related to configuring the simulator. Once configuration is complete and the DSPS is allowed to start running, execution control flows to the DSPS and all further interaction between the platform and the DSPS happens through callback notifications. When a callback occurs, the callback type is recorded in the transaction record and execution control is passed back to the system platform via the TLM wrapper. The TLM wrapper can either interact with the DSPS simulator at this time through any of the run-time API functions, or it can pass control to the main SystemC event scheduler in the platform.

After the start of simulation, it is necessary to have a way to mark the exact point in simulation time when callbacks occur in order to be able to verify the playback simulation is accurately reproducing the original simulation. Since the DSPS is a cycle-accurate model, time in the context of the simulation is measured in terms of clock cycles, and therefore the value of the system cycle counter at the time of the callback event occurrence serves as a convenient time stamp for the transactions.

The API provides a method to obtain the current simulated cycle count, called “GetSimulatedCycleCount()”. This method takes a pointer to a 64-bit variable as an argument and uses the pointer to update the variable with the current value of the DSPS cycle counter register. When called, each method in the API will make a call to this method to determine the current cycle count, and the return value is printed along with the abbreviated method name into the transaction record. Calls to this method from...
within the API are differentiated from external calls that would otherwise be recorded by the setting of an internal recording bypass flag prior to the call. The flag is cleared after the call is complete.

To illustrate how the API methods are instrumented to record the event name and cycle time stamp, I have provided the code for the previously mentioned “AddBusAccessCallback ()” method in Example 5. Although the instrumentation for recording adds complexity to the simulator code and increases the run time execution of the code by a negligible amount, no change is made to the basic simulator functionality. In other words, the recording instrumentation does not affect the functional behavior of the simulation. The DSPS behaves the same with recording enabled as it would if the feature did not exist.

Example 5 - "AddBusAccessCallback()" instrumentation example

```
1. API_Status APIWrapper::AddBusAccessCallback(void *deviceHandle,
2. unsigned int phyStartAddr,
3. unsigned int phyEndAddr,
4. bus_request_callback brc)
5. {
6.     if(recordFlag) // If enabled, record this event in the transaction record.
7.         {
8.             // Create a structure to save the relevant pointers.
9.             ptr_struct_t *brc_callbackPtrs = (ptr_struct_t *) calloc(1, sizeof(ptr_struct_t));
10.        }
11.    // Get the cycle count for this event for use as the timestamp.
12.    // "timestamp" is a class-level variable.
13.    if(endofConfiguration) { GetSimulatedCycleCount(&timestamp); }
14.    else { timestamp = 0; } // Simulation has not started running yet.
15.    recordBypassFlag = false;
16.    // Save the pointers that will be needed to record the callback events.
17.    brc_callbackPtrs->deviceHandlePtr = deviceHandle;
18.    brc_callbackPtrs->APIWrapperPtr = this;
19.    brc_callbackPtrs->user_callback = (void *) brc;
20.    // Write the event and relevant info into the transaction record.
21.    fprintf(fpRecord, "AddBusClbk %llx %p %x %x\n", timestamp,
22.        deviceHandle,
23.        phyStartAddr,
24.        phyEndAddr);
25.    // Register the callback, using the newly created pointer structure
26.    // as a device handle.
27.    return ((SimFunctions *)simFunc)->addBusCallback( pIss,
28.        brc_callbackPtrs,
29.        phyStartAddr,
30.        phyEndAddr,
31.        RecordBusAccessCallback);
32. }
33. // Recording is not enabled, proceed normally.
```
As shown in the example above, when recording is enabled and registration for a callback occurs, the callback registration method will save the pointer to the user-defined callback handler and register an interception callback handler instead. Referring once again to the bus access callback event, the interception callback handler implemented in the callback-recording layer of the API for a bus access callback event is named “RecordBusAccessCallback()”.

The code for this interception callback handler is provided for reference in Example 6. When a bus access callback event occurs, control of the simulation first enters the interception callback handler. An entry is made in the transaction record with abbreviated callback type, cycle time stamp, and argument values before calling the externally defined callback handler. On return from the external callback handler, the interception callback handler will record the return value before returning control back to the DSPS.

**Example 6 - Interception callback for bus access callback handler**

```c
1. BusTransactionStatus RecordBusAccessCallback(void *callbackPtrs,
2.                     unsigned int address,
3.                     unsigned int lengthInBytes,
4.                     unsigned char *data,
5.                     unsigned int requestID,
6.                     BusAccessType type,
7.                     unsigned int tnum,
8.                     BusBurstType burst)
9. {
10.    BusTransactionStatus status;
11.    unsigned long long timestamp = 0;
12.    unsigned int pcval = 0;
13.    ptrStruct *p_callbackPtrs = (ptrStruct *)callbackPtrs;
14.    // Restore the saved pointers.
15.    APIWrapper *ApiPtr = (APIWrapper *) p_callbackPtrs->ApiWrapPtr;
16.    void *deviceHandle = p_callbackPtrs->deviceHandlePtr;
17.    bus_transaction_request_callback user_btrc = (bus_transaction_request_callback) p_callbackPtrs->user_callback;
18.    // Get the timestamp and the current PC value (useful for debugging playback issues)
19.    ApiPtr->recordBypassFlag = true;
20.    ApiPtr->GetSimulatedCycleCount(&timestamp);
21.    ApiPtr->ReadThreadRegister(tnum, TH_REG_PC, &pcval);
22.    ApiPtr->recordBypassFlag = false;
```
// Record callback event in transaction record.
fprintf(ApiPtr>fpRecord, "BusClbk %llx %x %02x %08x %x %x
CYCLE: %llx TNUM: %d PC: 0x%x\n",
timestamp,
address,
lengthInBytes,
requestID,
type,
tnum,
burst,
timestamp,
tnum,
pcval);

// Call the user-defined callback.
status = user_btrc(deviceHandle, address, lengthInBytes, data, requestID,
type, tnum, burst);

// Record the return value of the user-defined callback.
HexPtr>recordBypassFlag = true;
HexPtr>GetSimulatedCycleCount(&timestamp);
HexPtr->recordBypassFlag = false;
fprintf(HexPtr->fpRecord, "BusClbkRetVal %llx %x\n",
timestamp, status);

return status;
}

Playback of Transaction Record

In order to play back the recorded simulation, a utility was written that parses the transaction record and uses the information to configure an instance of the DSPS with the same options and software image that were used in the original simulation.

It was fairly straightforward to write a C/C++ utility that would open the transaction record, parse each recorded entry line by line, and take action based on that entry. For example, referring back to the sample list of transaction record events in Example 2, the playback utility would first construct a simulator, configure the software application image, configure the bus penalty, and so on up to and including the beginning of the simulation.

In addition to handling the creation and configuration of the simulator, the playback utility registers the same set of callbacks as the original simulation, and handles the callback events in the same way they were originally handled. Handling the callbacks proved to be more complicated, but still feasible. The primary design goal for the record and playback feature is to enable the DSPS developers to easily debug model failures that occur within a complex system simulation platform. This goal provides latitude in the design of the playback utility with regards to how callback events are handled. Therefore, behavior that happens within the context of a callback event, but outside the scope of the DSPS, can be abstracted out of the design with no impact on the ability to reproduce DSPS failures using this feature.
This system abstraction allows callback events to be initially handled in the playback utility using a set of simple dummy callback handling functions. The dummy callback functions are used for callback registration by the playback utility, but really only serve as pass-through functions to a single generic callback handler function. The generic callback handler parses the transaction record for the relevant callback return value, and also makes any API calls encountered in the transaction record that occur between entry and exit of the callback. Once the callback return entry is found, the callback handler will return the same value back to the DSPS as was done during the original simulation as per the information provided by the transaction record. In order to maintain the integrity of the playback simulation, the callback handler uses the cycle count time stamp to verify the callback occurred at the same cycle as in the original simulation.

Using this approach, the DSPS can function as if it was in the larger system simulation platform, interacting with other hardware modules, without knowing that it is really only being fed a set of scripted information via the playback utility.

To illustrate how a dummy callback handler works, I refer once again to the bus access callback event. The code for the dummy callback handler that handles bus access callbacks, along with the full generic callback handling routine that handles all callbacks, is provided in Example 7.

Example 7 – Example playback utility callback handlers

```c
1. // Dummy callback for registration of bus access notification event.
2. BusTransactionStatus BusAccessCallback(void * handle,
3.     unsigned int address,
4.     unsigned int numBytes,
5.     unsigned char *data,
6.     unsigned int reqID,
7.     BusBurstType burst)
8. {
9.     unsigned int status;
10.    unsigned int pcval = 0;
11.    status = CallbackHandler("BusClbk", "BusClbkRetVal");
12.    return (HEXAPI_TransactionStatus) status;
13. }
14.
15. // Callback handler function that handles all callbacks.
16. unsigned int CallbackHandler(char *ClbkType, const char *ClbkRtrn)
17. {
18.    unsigned int status = 0;
19.    unsigned int pcval = 0;
20.    unsigned int tnum = 0;
21.    if(fgets(line, sizeof(line), fp)!=NULL)
22.    {
23.       sscanf(line, "%s %llx", ApiName, &timestamp);
24.    }
```
Figure 5 illustrates the software execution flow of the playback utility. Calls to configuration API methods prior to start of simulation will flow straight through the execution path from top to bottom, iterating through the transaction record till a “Run” entry is encountered, signifying the beginning of the simulation. Once the simulation starts running, control passes from the playback utility to the DSPS, marked by the execution state labeled “Run simulation”.

When a callback notification event occurs, control passes back to the playback utility through the appropriate dummy callback handler. The dummy callback handler calls the generic callback handler with the appropriate arguments, and the generic handler reads the next line in the transaction record and compares it to the callback to verify the callback was expected.

If the callback is expected, the generic handler invokes the file parsing logic to iterate through the transaction record until it encounters a return from callback entry, at which point control returns back to the DSPS. Any calls to run-time control API methods that occur while the callback is active will fall through the execution path from top to bottom, and loop back to the top for another iteration through the transaction record.

The playback simulation will continue to run in this manner until there is no more data in the transaction record or the simulation ends as expected. In this context, an expected ending could mean the simulation has run to completion, or that a reported DSPS failure condition has been reproduced. If an unexpected callback notification occurs, or if a callback occurs at a cycle different than what was recorded in the
transaction record, the playback utility treats this as an error condition and ends the simulation abnormally.

**Feature Validation**

In order to validate the playback feature, it was necessary to ensure that the simulation being played back exactly matched the behavior of the original simulation. One benefit of the implementation is that it is self-validating to a certain extent. All entries prior to the beginning of simulation have an effective time stamp of 0, so nothing needs to be
done aside from ensuring that the configuration functions are called in the same order and with the same arguments as was done originally. The sequential processing nature of the DSPS and subsequent transaction record ensure the ordering is maintained. For run-time callback notification events, checks are done to ensure that the callback events are expected and are taking place at the same cycle time as originally recorded. If a callback event occurs at a cycle that is different from the recorded cycle time stamp, or if an unexpected callback event occurs, these are error conditions that cause playback to be aborted. An aborted playback simulation means that the utility was not able to successfully reproduce the simulation represented by the transaction record. Therefore, it is guaranteed that a playback simulation that runs to expected completion matched the original simulation in terms of timing of events.

To illustrate this process, consider the bus access callback once more, since this is the most complex callback and also the most commonly used API method for integrating the DSPS into a system simulation platform. The complexity stems from the fact that it is the only callback that requires the device that registered the callback to provide a multi-level response. In the case of the bus access callback, the callback registrant is required to model a bus interface. This means the device handling the bus callback must inform the simulator whether or not the transaction can be accepted, and also must inform the simulator when the bus transaction is complete. In addition, the device must model the backing memory storage at some level, providing data when requested through reads and accepting data when directed through writes.

In the context of the system simulation platform, the TLM wrapper registers for the bus access callback, and passes all bus access callback requests to the bus model. The bus model makes a determination as to whether or not a transaction can be accepted, and the status of the transaction is returned to the DSPS via the TLM wrapper. Modeling of the backing memory storage for reading and writing of data is done via a separate dedicated memory device model.

A two-phase directed test was developed to verify that bus access callbacks were correctly handled. The first phase of the test consisted of a bare bones system simulation platform, which directly instantiates a DSPS object, configures a software application image and runtime parameters, and registers for a bus access callback over the entire 32-bit address range. The software application image consists of a simple test that does a series of reads and writes, compiled to run on the DSP using a dedicated C/C++ compiler. When the simulation is run with recording enabled, the fetching of instructions and reading/writing of data all result in bus access callbacks. The second phase of the test uses the playback utility to parse the transaction record generated by the first phase, load the same software application image, and replay the simulation. Once the playback simulation is complete, a successful completion of the test results in the same output printed out to the console as was done in the first phase. The total cycles executed by the second phase also matches the cycles executed in the first phase.
If the two simulations differed in any way, for example if one modeled a cache miss or a bus stall that the other did not, the program output might be the same but the cycle counts would not be. In addition to matching program output, matching final cycle counts between a recorded simulation and a playback simulation indicate that the simulations ran in exactly the same way.

The second most widely used callback in a system simulation platform is the timed callback. A timed callback allows a system simulation platform to register for a periodic callback at an interval designated during the callback registration. The timed callback is used for many purposes, but the main use of this callback in the system simulation environment is to restrict execution on the DSP simulator to a time interval defined at the system level as a “quantum”. In the context of the system simulation environment, a quantum is defined as a simulated time period in which the SystemC scheduler in the platform will allow a model such as the DSPS to control the simulation and run simulation cycles.

Once the quantum has expired, the DSPS must yield simulation control back to the scheduler, and the scheduler will give control to another model in the system. Simulation control is passed to each model in the system in round-robin order. Using this approach, all models have a chance to run and interact with each other when necessary. This execution ordering is necessary because the platform is designed as a single-threaded application. Therefore only one model can run simulation cycles at any given time, and interaction between models must occur at quantum boundaries. An explanation of how interaction between models occurs on quantum boundaries in system simulation platforms is provided in Appendix III.

In order to ensure the DSPS conforms to platform convention and only runs during its allotted quantum periods, the TLM wrapper registers for a timed callback notification, to be triggered after the simulated time quantum interval has passed. The quantum interval is converted into a discrete number of clock cycles to be simulated by dividing the quantum value by the inverse of the clock frequency. For example, a typical quantum value used in our system simulation platforms is 1000 nanoseconds. Since the DSPS is simulating a processor core running at 500MHz, or 2 nanoseconds per clock tick, the quantum is equivalent to simulating 1000/2 = 500 cycles on the DPSS.

When the timed callback event occurs after 500 cycles pass, the callback handler in the TLM wrapper takes this opportunity to yield control back to the SystemC scheduler, and the scheduler selects another model to run. When the scheduler eventually works its way back to the DSP simulator, control is returned to the callback handler in the TLM wrapper, and the callback handler returns control to the DSP simulator. The simulator then continues to run until the next timed callback event and the process repeats. During a callback event, the TLM wrapper is free to call any of the run-time API functions. A common API function called during a callback event is “GetPowerStatistics()”, which returns an estimate for the power that the actual
hardware device would have consumed at that point in program execution. An example of the DSPS yielding control of the simulation at a quantum boundary is provided at the end of Appendix IV.

To test that timed callbacks are handled properly by the playback utility, another two-phase test was developed, which once again uses a skeleton system simulation platform to run a basic test compiled for the DSP simulator. Instead of registering for bus access callbacks, the skeleton system registers for a series of overlapping timed callbacks, each registered with a different callback handler function. The intent of this test is to make sure that overlapping timed callbacks are properly recorded and properly played back. Once again, in the second phase the playback utility takes the generated transaction record and test image and uses them to play back the simulation. The output of both phases, including the number of simulated cycles, must match exactly in order for the test to pass.

Once the testing proved that recording and playback of simulations making use of the two most widely used callbacks worked correctly, the next step was to start combining callbacks. For example, a typical use case for co-simulation models is to model a bus interface using these two callbacks. When a bus access request callback occurs, a timed callback is registered before signaling to the simulator that the transaction is accepted. Notification that the transaction is complete does not occur until the timed callback is triggered, and data is either read or written at this point. Once the transaction is complete, the timed callback is no longer needed, and therefore the callback is unregistered prior to exit from the bus access callback handler. Using the combination of these two callbacks, it is possible to model different bus access latencies. I therefore developed another test that models bus accesses using this approach. The test makes use of another simple application that performs a series of reads and writes to verify that the playback results match the recorded results.

Once all three tests were consistently passing, I created several variations of the third test, using other callbacks and calling different API functions from within the various callback-handling functions. All together, seven directed tests were created to refine the playback utility before attempting a recording on a full system simulation platform. These tests can easily be automated via the use of scripts to run the original and playback simulations and perform the comparison of the output. Once automated, any test of this type can be used to create a set of regression tests that run against the DSPS to verify the integrity of the recording and playback features on a regular basis. Future work on this project will involve automating the testing process for regular testing of the recording instrumentation and the playback utility.

The next step in the testing process was to attempt to record a simulation running on an actual system simulation platform. As mentioned in the introduction section, there are two main types of system simulation platform in use within the company. The first is based on standard SystemC, and the second is based on a third-party tool called
Virtualizer, developed by Synopsis. Although Virtualizer is SystemC-based, the tool has been heavily customized and platforms developed with this tool can only be run with the specialized Virtualizer system scheduler. The Virtualizer platforms also require a license to run, which takes time to procure and incurs significant expense to any department that uses them. Therefore, initial testing of the recording and playback features on a full platform was done with one of the SystemC-based platforms. Testing the features for compatibility with the Virtualizer tool is noted as an area for future work in Appendix I.

A SystemC platform was obtained from one of our internal software development teams that develop the wireless modem protocol stack. This software is heavily dependent on interaction with various aspects of hardware, similar to what was initially illustrated in Figures 1 and 2 in the introduction. The system simulation diagram is reproduced below as Figure 5 for reference.

The platform is designed to model the various hardware components that would be required if the software was running on actual hardware, and therefore it interacts with the DSPS API via the TLM wrapper in many complex ways. Example 1, also provided in the introduction, is an illustration of just one of the more basic ways that multiple models in the system interact.
Since the API recording feature is entirely new, the required changes were incorporated into the tip of our simulator code development branch, with support only for the most recent version of the DSP processor core. However, modem software development is still focused on development for the previous version of the DSP core, which is only now making it out into the market. The modem team will soon be transitioning off the previous architecture version and onto the new version to get a jump on software development before tape out of the next architecture version, but the timing of my project puts me right at the tail end of development for the previous architecture version. This means that the modem software and the system simulation platform they use are currently designed around the previous version of the architecture, which my recording feature does not yet support.

The version discrepancy poses a problem with testing the API recording and playback changes against an actual customer platform, since making all the software changes required to support the previous version would take more time than is available to finish the project. I decided instead to see if I could build the platform and software for the latest architecture version. I was able to rebuild the platform to incorporate support for the new version, but I discovered that the modem software itself requires extensive
rewrites to account for hardware changes between architecture versions, and trying to modify the modem protocol stack to account for all the changes would take me far outside the scope of this project.

I therefore decided to take only the platform and discard the modem software image, and instead use a stripped down test case that makes use of the same proprietary Real Time Operating System (RTOS) kernel as the modem software. The simplified test uses the RTOS kernel to create several memory pools and launch multiple threads that perform arbitrary tasks before closing down and returning control back to the kernel. Although this test is simplified, just booting the kernel and having multiple threads run does exercise various components of the platform sufficiently enough for me to verify that the recording feature works on a full system simulation platform, which is the ultimate goal for the initial stage of this effort.

Referring back to figure 5, using this platform and the simplified test, I am able to generate a transaction record from a simulation that involves interaction with the TLM wrapper, bus model, and memory device model. The playback utility takes the transaction record that is generated and uses it to correctly play back the same simulation, providing abstractions as described in the implementation to eliminate the need for the devices of the system simulation model to be present. Interaction with the ARM core and interrupt model are not tested in this scenario. However, the ARM core interacts only indirectly with the DSPS through the bus model, and this scenario does test that interaction between the DSPS and bus model is recorded and replayed successfully. Although there is no substitute for actual testing, this is a good indicator that having the ARM core active in the system will pose no problem.

To reiterate the goal of the recording and playback feature, the reason for developing it is to reduce the complexity required to reproduce DSPS failures that occur in simulations on a platform as shown in Figure 5. Recording the API transactions in the simulation and using the recording to replace the platform complexity with a simplified abstraction accomplishes this goal. The original diagram showing how the playback simulation works is reproduced in Figure 6.

Future work will involve testing full interactions between all models, as well as adding the recording feature to versions of the DSPS that model older architectures.
Conclusion

The idea for the DSP simulator API record and playback feature is something I had previously discussed with the rest of the DSPS team over a year ago, but we have not previously had the time to focus on developing it due to higher priority commitments. Given that I needed to work on a substantial project for completion of my Master’s program, implementing this feature seemed like an excellent project to undertake. The work done on this project has the potential to benefit the whole team in terms of time saved when attempting to reproduce simulator failures that only occur in scenarios where the simulator is integrated into complex system simulation platforms.

Another potential benefit to having this functionality is the ability to increase the scope of our regression testing on the DSP simulator. As mentioned previously, many of the more robust software applications that run on the DSP require interaction with other external devices. The ability to record successful runs of these software applications on the system simulation platform and play them back would potentially allow us to add these more robust software applications to our nightly simulator regression suite as playback sessions, where the transaction record can be run against the tip of our simulator development tree to ensure changes to the simulator code do not affect overall functionality or introduce new bugs.

The goal for this project was, upon completion, to have a working prototype that demonstrates the record/playback feature using a Linux-based system simulation

Figure 6 - Playback utility, revisited
platform of moderate complexity to obtain a recording. The playback utility takes the transaction record and the software application image, and uses them to replay the simulation events that occurred on the original platform without the need to model all the complexity of the original platform. The prototype successfully handles data interactions between the DSP simulator and other models in the platform, such as bus traffic and timed notifications. The prototype handles static configuration options of the simulator such as debug options and enabling of various traces. Dynamic configuration options such as changing the clock frequency mid-run are also handled. The primary intent of this project was to develop a working proof-of-concept which shows the idea is feasible. Areas identified for future work are noted in Appendix I, to be undertaken after the class is completed.

References


Appendix I - Future Work

Several items were identified over the course of the project that did not necessarily fall within the scope of the stated goal, but that could be done to improve or advance the state of the project. These items have been identified throughout the report as areas for future work, and are collected here for reference.

Future work on this project will explore the following areas:

- Support for simulations running on Windows 7
- Allow specification of a custom name for the transaction record
- Avoid overwriting of existing transaction records through the use of a numerical naming convention
- Provide support for recording transactions of multiple DSPS models in the same platform
- Investigate the use of compression/decompression to reduce the size of the transaction record, and restore the transaction record to a human-readable format when necessary
- Automation of the testing process
- Testing of the record and playback feature with platforms created using the Synopsis Virtualizer tool
- Testing of full interactions between relevant models
- Back-porting of the recording feature to older architecture versions

**Appendix II – Cycle-accurate versus Event-based Simulation**

SystemC supports two main simulation methodologies: cycle-accurate simulation and event-based simulation.

Cycle-accurate simulation takes advantage of the fact that most digital circuits are synchronous in nature. In a cycle-accurate simulator, all of the logic that ultimately computes a value is collapsed into a single operation that is evaluated once per clock cycle, with no evaluation done of any intermediate states between cycle boundaries. Therefore, there is no concept of “when” a particular aspect of the computation occurs within a cycle. As a result, cycle-accurate simulation is typically faster than event-based simulation, although the level of detail that can be accurately modeled is greatly reduced compared to event-based simulation.

Event-based simulation typically traces every signal transition that can produce a state change in the system. Change in the system is evaluated based on logical events; whenever a change in an input event occurs, the corresponding output is evaluated. If the output acts as an input to another event, then the output of the secondary event is evaluated. This evaluation process continues throughout the system until a stable state is reached. An event-based simulator may not have any notion of timing, but transitions at each of the various points in the computation process occur in order. This ordering allows for a more detailed modeling of hardware at the cost of a decrease in simulation speed.

**Appendix III – Interaction Between Models in a System Simulation Platform**

Example 8 illustrates how communication between models occurs on quantum boundaries, using the first four steps of the simple transaction sequence from Example 1, which involve three different models: the ARM simulator, the bus model, and the memory device model. When the ARM simulator initiates a data access transaction to the memory model, the first step in the sequence is a request to the bus model for
access to the bus. The ARM simulator makes the request, but the ARM simulator must first yield simulation control back to the SystemC scheduler before the bus model can acknowledge the request. The ARM simulator will continue to run until expiration of the quantum, and only then does it yield control to the scheduler. At a point later in the simulation measured by $\Delta$ simulated cycles, but before the ARM simulator is allowed to run again, the bus model will be given control of the simulation. At this point the bus model can choose to acknowledge the pending bus request from the ARM simulator. The bus model will continue to run until expiration of the quantum before yielding control back to the scheduler. The ARM simulator does not process the acknowledgement of the transaction until it has once again been given control of the simulation by the scheduler.

A similar sequence occurs for every interaction between models in the platform. Given that the system simulation platforms operate on a cycle-driven basis, $t_0$ and $\Delta_i$ through $\Delta_{18}$ represent discrete simulated clock cycle values.

Example 8 - Interaction between models and the quantum boundary

- $t_0$: ARM simulator initiates a bus transaction request
- $t_0+\Delta_1$: ARM simulator yields control after quantum expiration
- $t_0+\Delta_2$: SystemC scheduler promotes bus model to active running state
- $t_0+\Delta_3$: Bus model accepts ARM simulator bus request
- $t_0+\Delta_4$: Bus model yields control after quantum expiration
- $t_0+\Delta_5$: SystemC scheduler promotes ARM simulator to active running state
- $t_0+\Delta_6$: ARM simulator writes data to memory shared between ARM and DSP
- $t_0+\Delta_7$: ARM simulator yields control after quantum expiration
- $t_0+\Delta_8$: SystemC scheduler promotes memory device to active running state
- $t_0+\Delta_9$: Memory device signals acceptance of write

Appendix IV – Sample Transaction Record

A larger sample of a transaction record is provided below for reference. Due to the size of the original transaction record, the sample is truncated at the end of the first DSPS time quantum.

```
Constructor 6005
ConfQProf ./qprof.cfg 0 NULL 0
ConfGProf NULL
ConfTimeMode 2
ConfExecBin bootimg.pbn
ConfStatFile ./stats.txt 2
ConfNULLPtr 1
AddSymFile 0 bootimg.pbn TNUM: 0 PC: 0x0
ConfMemFill 0
AddTimeClbk 0 0xc1f3f0 3e8 3
AddCoreRdyClbk 0 0xc1f3f0
AddBusClbk 0 0xc1f3f0 0 ffffffff
```
EVB 0 0 TNUM: 0 PC: 0x0
WrGlobReg 0 0 0
CoreFreq 0 1dcd6500 TNUM: 0 PC: 0x0
CoreFreq 0 1dcd6500 TNUM: 0 PC: 0x0
ConfAHB 8000000 8fffffff
ConfTCM 8000000
EndConfig
BusClbk 0 2e000000 18010 ffffffff b 0 2 CYCLE: 0 TNUM: 0 PC: 0x0
GetElapSimTime 0 4
BusClbkRetVal 0 1
BusClbk 0 2e019000 3010 ffffffff b 0 2 CYCLE: 0 TNUM: 0 PC: 0x0
GetElapSimTime 0 4
BusClbkRetVal 0 1
BusClbk 0 2e01d000 84510 ffffffff b 0 2 CYCLE: 0 TNUM: 0 PC: 0x0
GetElapSimTime 0 4
BusClbkRetVal 0 1
BusClbk 0 2e0a1510 fdf0 ffffffff b 0 2 CYCLE: 0 TNUM: 0 PC: 0x0
GetElapSimTime 0 4
BusClbkRetVal 0 1
BusClbk 0 2e0b2000 148 ffffffff b 0 2 CYCLE: 0 TNUM: 0 PC: 0x0
GetElapSimTime 0 4
BusClbkRetVal 0 1
BusClbk 0 1e000030 04 ffffffff b 0 2 CYCLE: 0 TNUM: 0 PC: 0x2e000000
GetElapSimTime 0 4
BusClbkRetVal 0 1
BusClbk 0 1e000034 04 ffffffff b 0 2 CYCLE: 0 TNUM: 0 PC: 0x2e000000
GetElapSimTime 0 4
BusClbkRetVal 0 1
AssertReset 0
LoadExecBin 0 NULL
BusClbk 0 2e000000 18010 ffffffff b 0 2 CYCLE: 0 TNUM: 0 PC: 0x0
GetElapSimTime 0 4
BusClbkRetVal 0 1
BusClbk 0 2e019000 3010 ffffffff b 0 2 CYCLE: 0 TNUM: 0 PC: 0x0
GetElapSimTime 0 4
BusClbkRetVal 0 1
BusClbk 0 2e01d000 84510 ffffffff b 0 2 CYCLE: 0 TNUM: 0 PC: 0x0
GetElapSimTime 0 4
BusClbkRetVal 0 1
BusClbk 0 2e0a1510 fdf0 ffffffff b 0 2 CYCLE: 0 TNUM: 0 PC: 0x0
GetElapSimTime 0 4
BusClbkRetVal 0 1
BusClbk 0 2e0b2000 148 ffffffff b 0 2 CYCLE: 0 TNUM: 0 PC: 0x0
GetElapSimTime 0 4
BusClbkRetVal 0 1
BusClbk 0 1e000030 04 ffffffff b 0 2 CYCLE: 0 TNUM: 0 PC: 0x2e000000
GetElapSimTime 0 4
BusClbkRetVal 0 1
BusClbk 0 1e000034 04 ffffffff b 0 2 CYCLE: 0 TNUM: 0 PC: 0x2e000000
GetElapSimTime 0 4
BusClbkRetVal 0 1
EVB 0 0 TNUM: -1 PC: 0x0
WrGlobReg 0 0 0
GetPwrStats 0 1
Run 0
TranslateV2P 0 8d793140 ffffffff
TranslateV2P 0 8d793193 ffffffff
AddMemRdClbk 0 0xa7b8e0 8d793140 8d793193
AddMemWrClbk 0 0xa7b8e0 8d793140 8d793193
TranslateV2P 0 8d757e47 ffffffff
TranslateV2P 0 8d757e47 ffffffff
AddMemRdClbk 0 0xa7b8e0 8d757e47 8d757e47
AddMemWrClbk 0 0xa7b8e0 8d757e47 8d757e47
TranslateV2P 0 8d778840 ffffffff
TranslateV2P 0 8d778840 ffffffff
AddMemRdClbk 0 0xa7b8e0 8d778840 8d778840
AddMemWrClbk 0 0xa7b8e0 8d778840 8d778840
TranslateV2P 0 8d77c0dc ffffffff
TranslateV2P 0 8d77c0df ffffffff
AddMemRdClbk 0 0xa7b8e0 8d77c0dc 8d77c0df
AddMemWrClbk 0 0xa7b8e0 8d77c0dc 8d77c0df
TranslateV2P 0 8d77783e ffffffff
TranslateV2P 0 8d77783e ffffffff
AddMemRdClbk 0 0xa7b8e0 8d77783e 8d77783e
AddMemWrClbk 0 0xa7b8e0 8d77783e 8d77783e
TranslateV2P 0 8d672e00 ffffffff
TranslateV2P 0 8d672e03 ffffffff
AddMemRdClbk 0 0xa7b8e0 8d672e00 8d672e03
AddMemWrClbk 0 0xa7b8e0 8d672e00 8d672e03
TranslateV2P 0 8d7a2df8 ffffffff
TranslateV2P 0 8d7a2dfb ffffffff
AddMemRdClbk 0 0xa7b8e0 8d7a2df8 8d7a2dfb
AddMemWrClbk 0 0xa7b8e0 8d7a2df8 8d7a2dfb
EVB 0 0 TNUM: -1 PC: 0x0
WrGlobReg 0 0 0
DeassertReset 0
Run 0
BusClbk 1 2e000000 04 ffffffff a 0 2 CYCLE: 1 TNUM: 0 PC: 0x2e000000
GetElapSimTime 1 4
BusTransFinish 1 4 ffffffff 1 0 c0 0 78
BusClbkRetVal 1 1
BusClbk d 2e000000 20 00000000 0 0 2 CYCLE: d TNUM: 0 PC: 0x2e000000
GetElapSimTime d 4
BusTransFinish d 20 0 1 48 fc 7c c3 4c 2b 0 0 d8 fb 7c c3 4c 2b 0 0 0 0 0 0 0 0 0 a9 9c 81 0 0 0 0 0
BusClbkRetVal d 1
BusClbk 77 2e000000 04 ffffffff a 0 2 CYCLE: 77 TNUM: 0 PC: 0x2e000000
GetElapSimTime 77 4
BusTransFinish 77 4 ffffffff 1 0 c0 0 78
BusClbkRetVal 77 1
BusClbk 77 2e000004 04 ffffffff a 0 2 CYCLE: 77 TNUM: 0 PC: 0x2e000004
GetElapSimTime 77 4
BusTransFinish 77 4 ffffffff 1 6 c0 0 67
BusClbkRetVal 77 1
BusClbk 79 2e000004 04 ffffffff a 0 2 CYCLE: 79 TNUM: 0 PC: 0x2e000004
GetElapSimTime 79 4
BusTransFinish 79 4 ffffffff 1 0 0 a2
BusClbkRetVal 79 1
BusClbk 79 2e000008 04 ffffffff a 0 2 CYCLE: 79 TNUM: 0 PC: 0x2e000008
GetElapSimTime 79 4
BusTransFinish 79 4 ffffffff 1 0 c0 0 a2
BusClbkRetVal 79 1
BusClbk 7b 2e000008 04 ffffffff a 0 2 CYCLE: 7b TNUM: 0 PC: 0x2e000008
GetElapSimTime 7b 4
BusTransFinish 7b 4 ffffffff 1 0 c0 0 a2
BusClbkRetVal 7b 1
BusClbk 7b 2e00000c 04 ffffffff a 0 2 CYCLE: 7b TNUM: 0 PC: 0x2e00000c
GetElapSimTime 7b 4
BusTransFinish 7b 4 ffffffff 1 0 d0 c0 56
BusClbkRetVal 7b 1
BusClbk 7d 2e00000c 04 ffffffff a 0 2 CYCLE: 7d TNUM: 0 PC: 0x2e00000c
GetElapSimTime 7d 4
BusTransFinish 7d 4 ffffffff 1 0 d0 c0 56
BusClbkRetVal 7d 1
BusClbk 7d 2e000010 04 ffffffff a 0 2 CYCLE: 7d TNUM: 0 PC: 0x2e000010
GetElapSimTime 7d 4
BusTransFinish 7d 4 ffffffff 1 0 d0 c0 57
BusClbkRetVal 7d 1
BusClbk 87 2e000000 20 00000001 0 0 2 CYCLE: 87 TNUM: 0 PC: 0x2e000010
GetElapSimTime 87 4
BusTransFinish 87 20 1 1 0 e2 67 c2 4c 2b 0 0 83 4f dd d 4c 2b 0 0 0 0 0 0 0 a9 9c 81 0 0 0 0
BusClbkRetVal 87 1
BusClbk f1 2e000010 04 ffffffff a 0 2 CYCLE: f1 TNUM: 0 PC: 0x2e000010
GetElapSimTime f1 4
BusTransFinish f1 4 ffffffff 1 2 c0 c0 57
BusClbkRetVal f1 1
BusClbk f1 2e000014 04 ffffffff a 0 2 CYCLE: f1 TNUM: 0 PC: 0x2e000014
GetElapSimTime f1 4
BusTransFinish f1 4 ffffffff 1 f6 c1 0 58
BusClbkRetVal f1 1
BusClbk fb 2e000000 20 00000002 0 0 2 CYCLE: fb TNUM: 0 PC: 0x2e000014
GetElapSimTime fb 4
BusTransFinish fb 20 2 1 0 e2 67 c2 4c 2b 0 0 83 4f dd d 4c 2b 0 0 0 0 0 0 0 0 a9 9c 81 0 0 0 0
BusClbkRetVal fb 1
BusClbk 165 2e000014 04 ffffffff a 0 2 CYCLE: 165 TNUM: 0 PC: 0x2e000014
GetElapSimTime 165 4
BusTransFinish 165 4 ffffffff 1 f6 c1 0 58
BusClbkRetVal 165 1
BusClbk 165 2e000400 04 ffffffff a 0 2 CYCLE: 165 TNUM: 0 PC: 0x2e000400
GetElapSimTime 165 4
BusTransFinish 165 4 ffffffff 1 0 de 3a 72
BusClbkRetVal 165 1
BusClbk 16f 2e000400 20 00000003 0 0 2 CYCLE: 16f TNUM: 0 PC: 0x2e000400
GetElapSimTime 16f 4
BusTransFinish 16f 20 3 1 0 e2 67 c2 4c 2b 0 0 83 4f dd d 4c 2b 0 0 0 0 0 0 0 0 a9 9c 81 0 0 0 0
BusClbkRetVal 16f 1
BusClbk 1d9 2e000400 04 ffffffff a 0 2 CYCLE: 1d9 TNUM: 0 PC: 0x2e000400
GetElapSimTime 1d9 4
BusTransFinish 1d9 4 ffffffff 1 0 de 3a 72
BusClbkRetVal 1d9 1
BusClbk 1d9 2e000404 04 ffffffff a 0 2 CYCLE: 1d9 TNUM: 0 PC: 0x2e000404
GetElapSimTime 1d9 4
BusTransFinish 1d9 4 ffffffff 1 b de 3b 72
BusClbkRetVal 1d9 1
BusClbk 1db 2e000404 04 ffffffff a 0 2 CYCLE: 1db TNUM: 0 PC: 0x2e000404
GetElapSimTime 1db 4
BusTransFinish 1db 4 ffffffff 1 b de 3b 72
BusClbkRetVal 1db 1
BusClbk 1dd 2e000408 04 ffffffff a 0 2 CYCLE: 1dd TNUM: 0 PC: 0x2e000408
GetElapSimTime 1dd 4
BusTransFinish 1dd 4 ffffffff 1 0 c0 3a 71
BusClbkRetVal 1dd 1
BusClbk 1dd 2e00040c 04 ffffffff a 0 2 CYCLE: 1dd TNUM: 0 PC: 0x2e00040c
GetElapSimTime 1dd 4
BusTransFinish 1dd 4 ffffffff 1 80 e1 3b 71
BusClbkRetVal 1dd 1
BusClbk 1df 2e000410 04 ffffffff a 0 2 CYCLE: 1df TNUM: 0 PC: 0x2e000410
GetElapSimTime 1df 4
BusTransFinish 1df 4 ffffffff 1 80 e1 3b 71
BusClbkRetVal 1df 1
BusClbk 1df 2e000414 04 ffffffff a 0 2 CYCLE: 1df TNUM: 0 PC: 0x2e000414
GetElapSimTime 1df 4
BusTransFinish 1df 4 ffffffff 1 18 c0 9 6a
BusClbkRetVal 1df 1
BusClbk 1e1 2e000418 04 ffffffff a 0 2 CYCLE: 1e1 TNUM: 0 PC: 0x2e000418
GetElapSimTime 1e1 4
BusTransFinish 1e1 4 ffffffff 1 18 c0 9 6a
BusClbkRetVal 1e1 1
BusClbk 1e1 2e00041c 04 ffffffff a 0 2 CYCLE: 1e1 TNUM: 0 PC: 0x2e00041c
GetElapSimTime 1e1 4
BusTransFinish 1e1 4 ffffffff 1 0 c0 38 71
BusClbkRetVal 1e1 1
BusClbk 1e3 2e000418 04 ffffffff a 0 2 CYCLE: 1e3 TNUM: 0 PC: 0x2e000418
GetElapSimTime 1e3 4
BusTransFinish 1e3 4 ffffffff 1 0 c0 38 71
BusClbkRetVal 1e3 1
BusClbk 1e3 2e00041c 04 ffffffff a 0 2 CYCLE: 1e3 TNUM: 0 PC: 0x2e00041c
GetElapSimTime 1e5 4
BusTransFinish 1e5 4 fffffff 1 9d 3a f3
BusClbkRetVal 1e5 1
BusClbk 1e5 2e000418 04 fffffff a 0 2 CYCLE: 1e5 TNUM: 0 PC: 0x2e000418
GetElapSimTime 1e5 4
BusTransFinish 1e5 4 fffffff 1 9d 3a f3
BusClbkRetVal 1e5 1
BusClbk 1e5 2e00041c 04 fffffff a 0 2 CYCLE: 1e5 TNUM: 0 PC: 0x2e00041c
GetElapSimTime 1e5 4
BusTransFinish 1e5 4 fffffff 1 32 48 e0 1
BusClbkRetVal 1e5 1
BusClbk 1e5 2e000420 04 fffffff a 0 2 CYCLE: 1e5 TNUM: 0 PC: 0x2e000420
GetElapSimTime 1e5 4
BusTransFinish 1e5 4 fffffff 1 9c c4 0 78
BusClbkRetVal 1e5 1
BusClbk 1ef 2e000420 20 00000004 0 0 2 CYCLE: 1ef TNUM: 0 PC: 0x2e000420
GetElapSimTime 1ef 4
BusTransFinish 1ef 20 4 1 0 e2 67 c2 4c 2b 0 0 83 4f dd d 4c 2b 0 0 0 0 0 0 0 0 a9 9c 81 0 0 0 0 0
BusClbkRetVal 1ef 1
TimeClbk 258 TNUM: 1 PC: 0x0
GetPwrStats 258 0
GetElapSimTime 258 4
GetPwrStats 258 0

The “TimeClbk” event marks the end of the first DSPS time quantum. At this point, control is passed from the DSPS to the TLM Wrapper. The TLM wrapper makes calls to two run-time API functions, “GetElapsedSimulationTime()” and “GetPowerStatistics()”, before yielding control to the SystemC scheduler.