A Simulation Framework for Multiprocessor SoCs by Integrating SystemC with High-Level Processor Models

Somasundaram Meiyappan
HT023601A
School of Computing
National University of Singapore
meisoms@ieee.org

Nirmala Ramakrishnan
HT035271W
School of Computing
National University of Singapore
nirmala.ramakrishnan@hp.com

Abstract

Simulation is an important technique in functional verification and performance analysis of System-on-Chip architectures. Complex multi-processor based SoC devices are simulated at RTL level. This paper presents a framework for combining the simulation strengths of SystemC with Simplescalar for simulating processors in a multi-processor SoC. The system is modeled as a distributed event simulation problem and simple socket based communication provides the back-bone between the different simulators. The framework can be used to perform both functional and time-accurate simulations.

1. Introduction

Today, embedded systems have grown in complexity with more components in the system that ever before. This trend would continue as miniaturisation continues. With shrinking feature size of semiconductors, designers pack more transistors into the modern integrated circuits in the form of System-on-chips (SoC). Many of the SoC based embedded systems are multi-processor systems. With this increasing complexity and time-to-market pressures, careful analysis of the system design is very important. Performance analysis and hardware/software integration have been pushed early in the development cycle. In most cases, system simulation is a technique that is capable of addressing both these needs of developers. There are some commercial tools like Seamless from Mentor Graphics [5] that support these activities.

Simulation of multi-processor systems are done either in high level processor modeling packages or at *Register transfer logic* (RTL) level. While the former provides a way to develop software for multi-processor systems, it does not enable us to perform analysis on the entire system which also consists of other hardware in the SoC. One of the popular choices for system level simulation is SystemC. Though it is very powerful, good high level processor models have already been written in C/C++ and hence, rewriting those instruction-set simulators in SystemC would be unproductive. SystemC allows us to link in other C/C++ libraries. If the instruction-set simulators were written in C++ with good object oriented techniques, then it would be easy to integrate the simulator into the SystemC application. But most often, it would be very difficult to integrate a simulator written in C into SystemC. This paper presents a way to harness the best characteristics of both simulation tools $i.e.$, SystemC and a high level processor simulator, Simplescalar.

2. Simplescalar

Simplescalar[6] is a widely used high level model/simulator of a processor. It is written in C. It has its own suite of compilers based on gcc. Many researchers have developed models of different processors in SimpleScalar. It can provide a fast simulation not taking into account the effect of micro-architectural features to a full processor simulation with out-of-order execution and memory hierarchy. Simplescalar provides a good mix of traditional instruction set simulator, micro-architectural simulator and a basic C library for the simulator to perform I/O operations like reading from files(s) form the hard-disk, etc.... It is a good tool for profiling an embedded program and any application-level program for that matter. Since it is written in C, it is hard to instantiate two instances of a processor within the same application. Several researchers have used Simplescalar to simulate ARM, Alpha and many other processors. The standard distribution of Simplescalar support Alpha line of processors and a proprietary architecture called PISA, which is a simplified form of MIPS architecture but has a similar instruction set, if not exactly the same. The number of papers that get published referring to Simplescalar is a testimony to its popularity among researchers.

3. SystemC

SystemC provides hardware constructs in the context of C++ as a library to enable us develop simulation models using standard C++. The application can be generated and executed in standard desktop machines. It is supported in most unix systems, linux and Win32 platforms. It is distributed in source under OSCI licensing. It can be viewed as a system level simulation platform that can be used to perform simulations at varying levels of accuracy, both timed and untimed. Tools that can synthesise hardware from the RTL subset of SystemC are being developed by popular EDA vendors like CoWare. It is developed by Open SystemC Initiative and can be downloaded from a book[7].

Many new SoC designs may start with an analytical exploration of the design space and then use simulation to choose the final design or to be able to satisfy the business factors mentioned above. As mentioned above, SystemC is suited for system architectural modeling to gate-level modeling. SystemC is quite popular in the industry as well. One example is that the Open Core Protocol or OCP [4] uses SystemC for its modeling purposes. Thus, it is clear that SystemC has an important place in the design flow of SoCs. There are other system design frameworks and SystemC has a jumpstart in terms of commercial adoption as it is based on standard C/C++ language. SystemC allows developers to link in standard C/C++ libraries into the models and thus enabling faster modeling and simulation.

3.1. SystemC 101

In this section, we introduce some terms to the reader that would be used in the upcoming sections. To get a very good understanding of SystemC, we refer the reader to [8] and [9]. [11] covers a great deal of material on the synthesisable subset of SystemC.

In SystemC, a system would be defined as modules that has various sub-modules, called threads or methods, which have ports to communicate with other modules. The ports can be connected through channels. SystemC supports standard C/C++ types and also other types that are similar to types found in hardware description languages like VHDL and Verilog. Please note that some types are not synthesisable.

SystemC threads, SC_THREAD, are similar in concept to threads that are found in operating systems. These are free running threads of execution and can suspend until a signal that it is sensitive to changes its value. Its sensitivity list can change dynamically through the wait command that suspends execution of the thread until the event happens.

SystemC methods, SC_METHOD, are functions that are executed when the inputs it is sensitive to, has changed. Once the method begins execution, it has to complete its execution and it cannot suspend as SC_THREAD can do. As any HDL programmer would see, this is similar to structural or behavioral models in VHDL and Verilog. SC_METHODs are part of the synthesisable subset of SystemC; but SC_THREADs are not synthesisable.

3.2. Internals

We would like to introduce the reader to some internals of SystemC, specifically version 2.0.1. Though internals are generally bound to change with versions, we do not expect these presented here to change in future versions of SystemC as the features mentioned in this section are best implemented this way to reduce the latency of scheduling and simulation. Also, the framework makes use of these characteristics of SystemC.

1. SystemC is a single threaded simulation model.
2. SystemC has its own scheduler to schedule SC_THREADs and SC_METHODs. The scheduler does not pre-empt execution of its threads/methods.
3. An SC_THREAD does not translate to a thread in the host operating system that the program is running on.
4. SystemC is an event based scheduler and SystemC advances its time to the time of the earliest event.
5. Since events are generated as a consequence of past events (be it generated from a testbench or be it a process feeding another process with some input), the scheduler stops the simulation when there are no more events to process.
6. It can be noticed that because of reasons above the simulation can be blocked $i.e.$, paused for a while by blocking the SystemC scheduler. This can be done by blocking any SC_THREAD or SC_METHOD using the host operating system synchronisation primitives like semaphores.

4. Motivation

SystemC is a powerful system simulation platform. Simplescalar is a widely used instruction set and micro-architecture simulator. By harnessing the capabilities of both these powerful tools, an effective high-level SoC system simulation environment can be developed. This would also give researchers a cost-effective benchmark, to compare the results obtained using newly developed analytical techniques for analysing a design. Also, other benefits of such a simulation environment that were discussed above can also be realised.

5. Simulation framework

Two different approaches were identified to make Simplescalar and SystemC communicate with each other. They are

1. Compile SimpleScalar as a library and link it with SystemC based models to get a single application.
2. Compile SystemC part of the model into an application and the SimpleScalar part into separate application(s). Start them in the host as separate applications and communicate with each other using *Interprocesscommunication* (IPC) or *Remote Procedure Calls* (RPC) methods.

The first approach would lead to less overhead and faster simulators. Since there are other highlevel processor models other than SimpleScalar, significant effort is needed to compile a chosen processor simulator with SystemC. In somecases, it may become impossible to instantiate two processors within a single application. In the second approach, common features of any processor simulator and the characteristics of SystemC based models can be exploited to develop a framework such that most of the processor simulators can communicate with SystemC based models. The second approach also helps us run the simulation in distributed fashion thus increasing the performance of some kind of simulations. Please note that this approach would introduce overheads in terms of communication between the various applications. For certain kinds of simulation, this overhead can degrade the performance of the simulation environment. In this paper, we choose the second approach and the forthcoming sections would present an implementation based on that approach.

5.1. Overview

The distributed simulation approach that was chosen can result in different set of frameworks from one which is optimised for speed and for functional/time-accurate simulation or another framework that is very general and has a wide scope of application; but not optimised for speed. Here, we choose a very simple bus based multi-processor SoC and implement a simulator that is time accurate. The framework can also be used to do functional simulation of the application.

When writing software for multi-processor systems, synchronisation between the programs running in each of the processor is key to correct program output. Programs written for multi-processor systems synchronise using standard synchronisation primitives like spin-locks. In spin-locks, program running on one processor continuously reads data from a particular memory location. When it reads a 1, then it assumes that its input from the other processor is ready and begins to process it. More information from this can be found from the various literature in the web. Since this automatically takes care of synchronisation between the two processors, functional simulation should also lead to correct results if the program is well written.

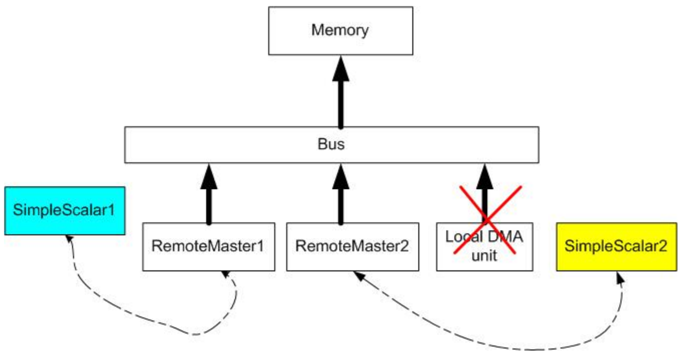

Figure 1. Assumed bus based system to be simulated

Timing-accurate simulation gives us more accuracy by taking into account the latencies involved in the system, the effect of bus contention and arbitration, etc. . . This class of simulation is in concept similar to cycle-accurate simulations; but we do away with the generation of a clock signal in order to speed up the simulation. In developing the framework, we also made the assumption that modules involved in the simulation communicate only through the bus and do not have any other form of direct communication among them like interrupts from a DMA module to the processor. To demonstrate the framework, we assume a simple system shown in figure 1. The framework has two parts. One of them is compiled with the SystemC part of the model. This acts as the server and the other part is compiled into the remote modules $i.e.$, the Simplescalar applications acting as the client.

5.2. Implementation: SystemC

This section explains the problems that have to be addressed to obtain a distributed simulation environment as stated above. This section also desribes the server part of the architecture $i.e.$, the SystemC part of the system described in figure 1. Please refer to section 3.2 for supportive arguments on the statements made in this section.

1. We have to integrate the inter-process communication, that happens with the distributed application, with SystemC environment. SystemC has some characteristics described in section 3.2. These characteristics must also be taken into account when solving this challenge of integration with SystemC.
2. To enable timing-accurate simulations, the SystemC application must synchronise the clocks of all the remote subsystems involved. To precisely model the effect of simultaneous requests to the bus, the requests from the independent processor simulators that run as a separate application should be forwarded to the bus only when it is absolutely safe to do so. This means that in a two processor system, when we have a bus request from processor 1 at time t_1, the bus cannot begin to process the request at that time and should instead wait for the time in processor 2, t_2 to be greater than or equal to t_1. In the mean while, if the SystemC application receives a request from processor 2 when t_2 is less than t_1, then that request from processor 2 is forwarded to the bus. This is not an issue when we are interested in just functional simulation.

SystemC needs events to keep the simulation alive. This means that when the framework should be able to predict the condition when there would be no events to keep the simulation alive and also take into consideration the above requirement for a timing accurate simulation. Under these conditions, the SystemC scheduler needs to be blocked from executing further. This is needed so that the SystemC scheduler does not see the condition that there are no events being generated in the system and stop the simulation. This blocking is achieved in the framework by making the *Synchroniser* wait for a message from any of the remote simulators.

Synchronisation of clocks is important to achieve time/cycle accurate simulation. The assumptions that were made in section 5.1 mean that the processor simulators can run independently as long as they do not have a bus request. But when we have a bus request, then Simplescalar's clock should be synchronised with the SystemC clock as mentioned above. Instruction set simulators or processor simulators use the concept of clock cycles rather than the physical unit of time. This has to be converted into time that SystemC can understand. For this conversion and to perform any pre-processing, like address translation, on the bus requests from the processor simulators, the framework needs a proxy module for every remote bus master. This proxy receives bus requests and forwards them to the *Synchroniser*. The *Synchroniser* basically waits for the *safe* time for the request to be passed onto the bus. The *safe* time condition is met if the smallest time among the time in all remote masters is greater than or equal to the time at which the request has to be processed. If we have local masters in the system, even then the same condition will hold. The reason behind this is that SystemC processes events after sorting them in time. If the local master generates a request at time t_l and the $safe$ request from all of the remote processors is a request at time t_r such that t_m is less than t_r, then SystemC processes the request from the local master first. If it generates further requests before t_r, then they are processed before the request from the remote master is processed. So, the main job of the *Synchroniser* is to time-synchronise requests from the remote masters. The proxy for any remote master has some common functionality and its functionality is encapsulated into a C++ class called *RemoteModuleInterface*.

It is theoritically possible in a multi-processor system with each processor having instruction cache that the processors do not generate a bus-request for a long time. And since bus requests are the only way for SystemC application to know the current time in a processor, the *Synchroniser* will not be able to know the current time of, say, Processor 2, if its simulator does not generate bus requests for a while. It could happen when it is computing on the data that is available in its data cache and executing instructions from instruction cache. If the SystemC application receives a bus-request from Processor 1 and if we cannot certify that as a *safe* request, then that request has to wait for a long time before it can be considered *safe* after Processor 2 generates a bus request. To take care of such conditions, the *Synchroniser* queries all processors for their current time if they do not have any outstanding bus requests and the instruction-set simulators of these processors return the current processor cycle they are in. We derived inspiration for this from Chandy-Misra-Bryant algorithm [12] and its main focus is on a way to run distributed discrete-event simulations to produce the same results as sequential discrete-event simulation by avoiding dead-locks that can happen in trying to detect if an event is *safe* to be processed. [13] presents an evaluation of that dead-lock resolution algorithm for digital logic simulation and they were negative about its adoption because of performance and complexity reasons. But, our problem is set at a more higher-level where only bus transactions are transferred from the instruction-set simulators to the SystemC model. Also, we have simlified the problem by synchronising events from the remote systems in only the SystemC application and thus eliminating the dead-lock conditions that is common in distributed discrete-event simulation. However, we might have to deal with the dead-locks if interrupts to the processor are to be supported as well. And hence, we have made a mention about that here.

The important classes of the framework are listed below with some description about their functionality.

- *Synchroniser*: An object of this class provides the time-synchronisation between requests from various remote bus-masters. It sits in-between the actual bus and the proxy for the remote modules. The single instance of this class in the system spawns a POSIX thread that listens on a particular socket. All remote masters connect to the system by passing their registration information to this socket. The important registration information from the remote masters $i.e.$, the Simplescalar applications here are the SystemC module name to which they should connect to and the TCP/IP port at which the proxy can communicate with the remote master. This thread exits itself when it has received registration for all remote modules in the system. It was designed this way so that all remote masters have a single point of contact for registration and then create their own dedicated channels of communication. There is also a SystemC thread (SC_THREAD) implemented in this class. This SC_THREAD blocks the SystemC scheduler when there are no bus calls being processed. As mentioned earlier, this class acts as a wrapper to the bus for all proxies of the remote masters. So, this class also has methods similar to the bus request methods like word-read, word-write, burst-read and burst-write. After a bus request is completed, it immediately sends SystemC notification signals to the SC_THREAD in this class to wake-up and either block the SystemC scheduler or process another bus request.

- *RemoteModuleInterface*: This class provides the underlying proxy for the remote masters participating in the simulation. Each instance of this class spawns a new thread in the host operating system. This thread waits for requests (or communication packets, in general) from the remote masters. On receiving the requests, it posts them to a queue that is dedicated to receiving messages from that particular master and signals the *Synchroniser* through a POSIX semaphore to let the SystemC scheduler unblock and continue with the simulation. Ofcourse, it would later block again if it does not have a *safe* message or if there are no requests to process. Once SystemC scheduler wakes up, the *Synchroniser* SC_THREAD then sends a wake-up event notification to an SC_THREAD implemented in the *Processor* module, which is described next. *RemoteModuleInterface* also provides the general functionality of accessing the dedicated message queues and ma! king the actual bus request (through the *Synchroniser* ofcourse. It also implements helper methods to receive and send messages through the communication channel to the actual remote master application. Figure 2 describes the basic sequences of registration and two bus requests from the two Simplescalar instances. The order in which the memory requests are processed in figure 2 is true only for the a time-accurate simulations. For functional simulations, the memory request that reaches the system first gets executed. Also, the cycle number given in figure 2 is the bus-cycle number at which the request was supposed to be received by the bus and not the cycle number of the processor at which the request was made.

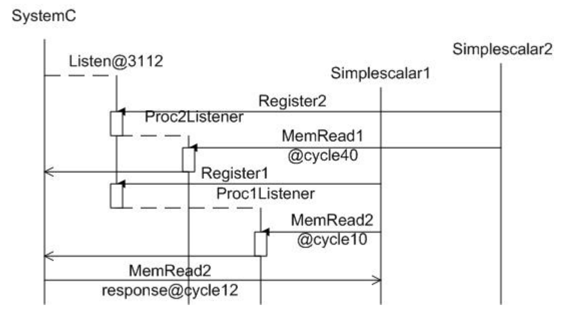

Figure 2. Sequence diagram of basic communication in the framework

- *Processor*: In figure 1, the SystemC proxies of the two processors in the system are encapsulated into this single C++ class. It allows customisation to do some necessary tasks that processor wrappers would do in SoC. It allows the frequency of the processor clock to be varied and also allows a custom address translation routines. The *RemoteModuleInterface* class, described above, is not a SystemC module (SC_MODULE). It is just a part of the simulation framework. But this *Processor* class is an SC_MODULE and makes use of all the features provided by *RemoteModuleInterface* and they have an inheritance relationship. It mainly implements an SC_THREAD as described in the description of the *RemoteModuleInterface* class. This class is not really a part of the framework; but it demonstrates the usage of the *RemoteModuleInterface* class.

As mentioned earlier, the remote masters involved in the simulation communicate with the SystemC model only and they communicate using messages over a TCP/IP based network. This can be optimised into local unix sockets if the programs are going to run on the same machine. Communication can also be done through other forms of interprocess or remote-process communication. Other parallel programming techniques like MPI (Message Passing Interface) have not been explored during the implementation. Having a dedicated thread for receiving messages from a particular remote master enables us to to continue to accept bus-requests from an advanced processor simulator whose architecture supports outstanding memory loads. The message queue that we have in each of the *RemoteModuleInter-*

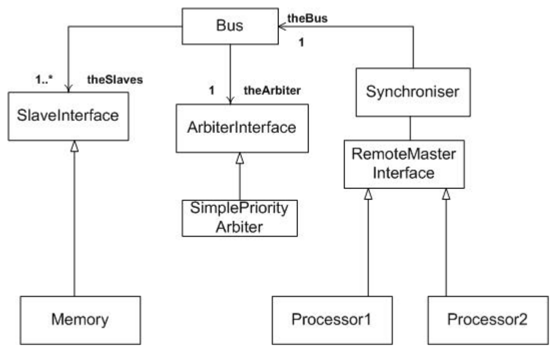

Figure 3. Class diagram of the SystemC application

face instances acts as a buffer for all bus requests that come in from a remote master when the proxy is already processing a bus-request. We do not have a flow control mechanism in the implementation; but that can be implemented if needed, and of course at the cost of performance. All requests that come from the remote masters are time-stamps in their own local units. For example, instruction set simulators or a micro-architectural simulators would time-stamp the requests in terms of the processor cycle at which the request was made just as shown in figure 2. The proxy of the remote master then converts the clock cycle to the real-time by using the clock frequency of the processor and the bus frequency to processor frequency ratio. Please note that both these values are passedf into the *RemoteModuleInterface* through the *Processor* class' constructor. Time synchronisation is done based on this common SystemC time-unit. The messages in the framework actually depend upon the capabilities of the bus being implemented. But the current protocol supports 32-bit word access with various byte valid masks and also supports up to 32-byte bursts. And all memory accesses always transmit the 32-bytes of data. This is not efficient and there is scope for improvement. The messaging protocol can be more optimised and less bytes being transferred might mean better speed of simulation; but that is not an established fact in this case as the underlying protocol, TCP in this case, always transmits a minimum packet size and our requests are already small in size. For all bus requests from the remote masters, the SystemC applications sends back a response. This response not only has the data that is read (if the request was a bus-read) but also the time-stamp at which this request was completed. The unit of the time-stamp is once again the unit used by the remote master, clock cycles in the case of processor simulators. This information can be used by the processor simulator to update its clock cycles and thus the simulators themselves can give a better picture of high-latency operations in the system either because of the bus contention or because of the natural latency involved in memory reads.

When a remote master has generated all the events (bus-requests rather) that it can generate and decide to leave the simulation because it is basically done, it sends a *disconnect* message to its *RemoteModuleInterface* which then enables the SystemC part of the simulation and other remote masters to continue with the rest of the simulation. It also enables the simulation to stop gracefully.

5.3. Implementation: Simplescalar

The only part of the framework that is needed in Simplescalar is the communication package to communicate with its proxy in the SystemC application. The communication subsystem of the framework is very similar to the one that we have in the SystemC application. It is best described by the source code and the documentation in the source code. So, we do not discuss much about that here. Instead, we describe some specific characteristics of Simplescalar and then present a list of possible things that can be done in a trade-off between accuracy and speed of the simulation.

As mentioned before, communication through TCP/IP involves a significant amount of overhead and causes the simulator to run at a much slower speed. We have three options before us.

1. *Forward all memory accesses to the bus*: If the Simplescalar application communicates with the SystemC application for every memory access, be it an instruction fetch or a data memory load/write, we get very accurate performance figures for the given set of inputs. But the speed of simulation takes a beating and the length of simulation increases tremendously.
2. *Forward only data access to the bus*: This is the next level of trade-off and send all the data to the SystemC part and let the instructions be fetched from the from local memory. This approach cuts down a significant portion of time and instruction fetches are more frequent that data memory accesses in any system.
3. *Forward only shared data memory access to the bus*: This is the fastest way to get the simulation results using this framework. Instruction fetches either as a burst read or as a word read does eat away some significant portion of the bus bandwidth. If we do not take that into consideration, then we loose some level accuracy. This is more suited for functional simulation in which case we just want to verify the functional correctness of the program and the shared memory operations rather than look at the timing information.

Interestingly, simulations of *type 1* listed above are easier to implement in Simplescalar than simulations of *type 3*.

The main reason behind the difficulty or trickiness is that Simplescalar assumes a fixed memory map for all the programs as shown in figure 4. So, it becomes tricky to separate the the shared memory requirements of a program into a separate shared memory section. Also, in simulations of *type 3*, we cannot use routines like 'malloc' to allocate memory in the shared memory region as malloc can be used to allocate memory only in a defined memory region in Simplescalar. But it is very easy to map the entire memory space or all data memory to the memory module in the SystemC application. Fundamentally, all memory accesses in Simplescalar are channelled to the macros called MEM_READ and MEM_WRITE. So, these macros provide an ideal place to check where a memory access has to be forwarded. When running simulations of *type 3* as mentioned above, these memory accessor macros are modified as shown in figure 5. If a more accurate simulation as mentioned in *type 1* is desired, then these memory accessor macros can always be made to refer to the remote memory irrespective of the virtual address space that the processor tries to access. This would also enable us to use memory allocation routines like 'malloc' on the shared memory. Please note that we have to have the correct address translation function in the processor's proxy module in the SystemC application. This address translation function coverts the virtual physical address from Simplescalar into the real physical address that can be used to access the memory module.

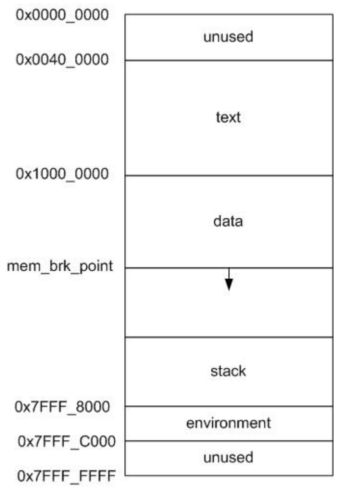

Figure 4. Memory map assumed by Simplescalar

But there is another important step in letting Simplescalar simulator to know that we have the address space that should be treated as shared memory. This is done by changing the linker script when compiling any source program to include a $.shrdmem$ or $.rmtmem$ sections to indicate a shared or remote memory section. Then the loader, when it loads the program, places this address region in the memory module in SystemC and makes some internal adjustment to the address space. For the sake of simplicity, we have demonstrated this with some changes to the anagram test program that comes with the standard Simplescalar 3.0d distribution. Anagram is as single processor application. But a more practical multi-processor application is also demonstrated in similar ways. It is described in the forthcoming sections. Sometimes, it is not possible to do what the researcher wants by simply changing the linker script. In such cases, we have to change the Simplescalar program even more. For example, if *type 1 or type 2* simulations are desired, then the address space checking macro, IN_REMOTE_MEM, that is defined by the framework for use in MEM_READ and MEM_WRITE macros have to be modified to forward the corresponding requests to the bus $i.e.$, to remote memory in the SystemC application. The remote memory also has to be correctly initialised in loader.c by hardcoding some information.

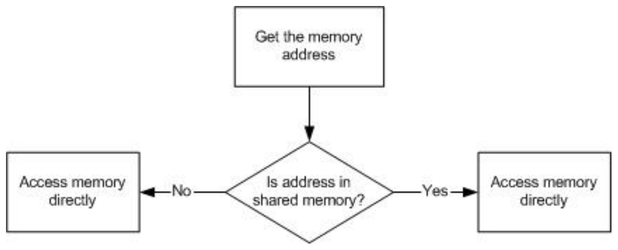

Figure 5. Modification of MEM_READ and MEM_WRITE macros in SimpleScalar to support simulations of type 3

There is another challenge. It was mentioned in section 5.2 that processor simulators provide the SystemC model with the processor clock cycle information as the timestamp of any bus request.While the *sim-out-of-order* simulator has the concept of clock cycle, the other Simplescalar simulators do not have such a concept of clock. But all of them have the instruction executed count which can also acts as a clock variable. But the best alternative is to use another variable that maintains the clock information separately.

Simplescalar provides standard C syscalls and these use a particular function in Simplescalar's $memory.c$ that is not optimised and accesses every byte of a 32-bit word separately. If time-accurate simulation was performed, then SystemC would consider each of the bus-access separately and time them separately. This results in wrong time measurements being reported unless the real system is designed in an unoptimised way. Also, it decreases the speed of simulation.

5.4. Performance analysis

The framework with this demonstrated system does not provide performance results/metrics on the application other than the execution time of the entire system for a given set of inputs and exit condition. The processor simulator gives us more detailed profile of the program. Buffer fill levels can be obtained by tracing certain signals/address in SystemC. This not only helps us get the maximum buffer fill level; but also help us put this information in the context of time and sequence of inputs. Buffer-fill levels can also be obtained by the simulated program themselves. Though the framework itself does not provide performance analytical tools (not implemented because of time constraints), it is very easy to build one as the bus implementation in on SystemC. Bus band-width analysis can be done by measuring the period of bus activity over any monitoring period. This information can also be used to identify peak band-width requirement and maximum latency experienced by any bus-request. But these results would be meaningful only if *type 1* simulations are enabled in Simplescalar (see section 5.3) so that all types of memory accesses are passed to the system bus. Even when *type 2 or type 3* simulations are done, the bus-bandwidth analysis gives the bandwidth usage when inter-processor communication is performed. In effect, the tracing and profiling capabilities of the SystemC and Simplescalar can be used to provide a good suite of performance monitoring tools around this framework.

6. Demonstrated application

The framework was demonstrated with a simple partitioning of a JPEG image flipper. The original source of the decoder was [1]. The JPEG source was partitioned such that the tasks map onto two processors. Synchronisation between the tasks across the two processors is taken care of by the application itself $i.e.$, the application must be designed to take care of the synchronisation of the software since we are executing the same code that eventually would be executed by the two processors in the real system. There are at-least two advantages of this approach against other more abstract approaches and they are

1. Early hardware/software integration and verification can be achieved.
2. Performance analysis can be done also taking into account the micro-architectural features of the processors without simulating at RTL level which takes even more time.

At the same time, this requires the software developers to partition the code and change the target application code to take care of synchronisation. This could take some time and also needs the involvement of the application developer(s) to change to code to take care of multi-processor communication. Thus it may not be suitable for design space exploration of a large design space. But this approach can be used to choose between two or three similar designs in a reasonable amount of time if the designers want to measure the performance when the micro-architectural features of a micro-processor are also considered.

6.1. Partitioning the application

The demonstrated application uses the JPEG codec pipeline to perform a horizontal mirroring or flippinf of a JPEG image using lossless image transformation routines supported by the JPEG source [1]. This application uses the JPEG codec pipeline as shown in the figure below: The decoder engine generates the DCT coefficients

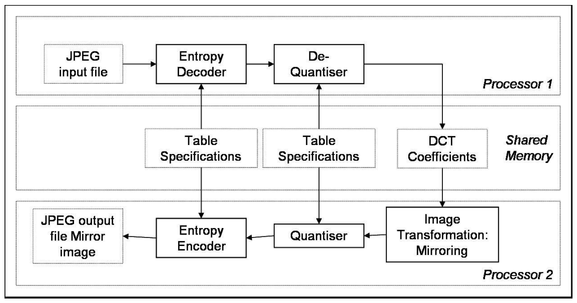

Figure 6. Flow of the JPEG mirroring application

of the input JPEG file. A horizontal mirroring is performed on this array of DCT coefficients.

6.1.1. Horizontal mirroring The image transformation routines work on DCT coefficient arrays and thus do not require any lossy decompression or recompression of the image. In the JPEG decoder pipeline we do not perform the IDCT (Inverse-DCT), the last stage in the decoding process. The output is intercepted before this stage and this is the DCT coefficient array. The horizontal flipping is done 'in-place', using a single top-to-bottom pass through the virtual source array and no extra memory is required. Horizontal mirroring of the DCT blocks is achieved by swapping pairs of blocks in-place. Within a DCT block, we perform horizontal flipping by changing the signs of the odd-numbered columns.

6.1.2. Partitioning The application was partitioned after obtaining a basic runtime profile of the application when run on a single processor, into two stages:

1. Initial decompression
2. Image tranformation and Compression

It was found that stage 2 executed $\frac{1}{3}$ of the number of instructions executed by stage 1. The quantisation tables generated from stage 1 as well as the actual DCT coefficient array were stored in the shared memory since they were required by stage 2. This constrains the synchronisation between the two processors.

Once the partition was determined, the application code was changed to take care of synchronisation. Each processor to be simulated was started as a separate Simplescalar process in the host environment with its part of the JPEG decoder application. Before the Simplescalar process was started, the SystemC process that models the bus is started as it acts as a memory server and provides synchronisation services.

Stage 1 was executed as a *type 2* simulation and stage 2 was executed as a *type 3* simulation. This demonstrates the flexibility of the framework in adapting to different simulation needs.

6.2. Target application

When printing a document, the host machine sends down the document described in a page description language(PDL). On the printer, a parser reads the file and renders the entire document in the form of rasters for subsequent processing by the printing pipeline. Due to the nature of the PDL, wherein it describes objects to printed with their location and other attributes, a page needs to be parsed completely before it can be sent down to the pipeline. See the figure below: When the image/document is complex, the image rasters can require a lot of memory, and so they are stored as JPEG images. As and when the parser has enough information to be merged with the rasters, they are decoded, and compressed back. Eventually when the whole image is ready, the rasters are decoded and sent down to the pipeline. When performing a duplex printing, mirroring of the rasters takes place as shown below: Hence, mirroring of JPEG images can be used by printers during duplex printing. The rasters would form a stream of JPEG objects that get processed by the mirroring hardware and are stored back.

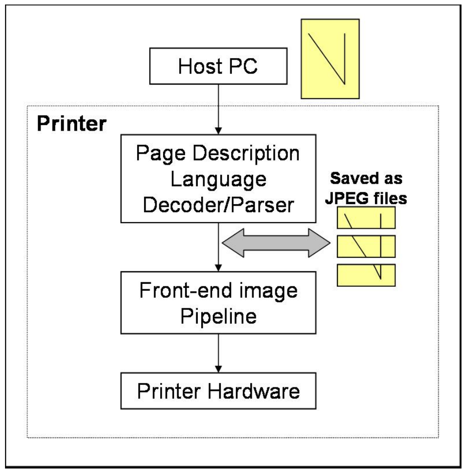

Figure 7. Use of JPEG Codec in a Printer

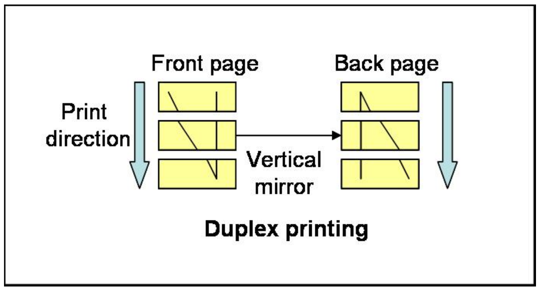

Figure 8. Mirroring in Duplex printing

6.3. Speed of simulation

Simplescalar gives us the instruction execution rate in a simulation. This is a measure of speed/performance of the simulation environment. This simulation was done using *sim-safe* on a 1.3GHz Pentium 4 machine with 384MB of memory.

Sim-safe, in a stand-alone mode, simulates the processor at 2 million instructions per second.

In this demonstrated application, we have used *type 2 and type 3* simulations for processor 1 and processor 2 respectively. Thus, instruction execution rate of both these two types of simulations are known. The instruction rate for the *type 2* simulations are around 44800 instructions per second and it was 75314 instructions per second for the *type 3* simulations. It has to be noted that all three applications $viz.$, Simplescalar1, Simplescalar2 and the SystemC application were running on the same system at the same time. There are a few main reasons for this significant drop in performance:

1. *Communication Overhead*: Communication be-

tween the three applications is currently done through TCP/IP protocol. This is a high-latency communication channel when all three application are running on a single machine. Using *shared memory* IPC is an option. Other inter-process communications or protocols can also be used to address this. In a distributed system, even the inter-connect network can be changed, say from Ethernet to Myrinet.

2. *Effects of bus-contention*: Bus-contention makes a request wait for a while before it is processed. This delay is worse in time-accurate simulations than in functional simulations because of the side-effects of requests having to be certified as *safe* before they can be processed by the bus.

3. *Bus-writes*: The Simplescalar part of the framework can be modified such that it does not wait for the bus-writes to complete, which it does today. This can speed up the simulation in cases where the application produces significant amount of data.

Type 3 simulations can potentially run faster if the program runs on another machine in a truly distributed fashion. But this would increase the communications overhead.

7. Related work

From our research, it looks like this is probably one of the first works to be presented that creates a distributed simulation environment for simulating a SoC by combining heterogenous simulation platforms like SystemC and Simplescalar. The core part of this framework is still SystemC. We did find some related work in the area of distributed SystemC in [14]. *Distributed Simulation Tools (DiST)* package[3] is understood to be simulating a program in Simplescalar in a distributed way by splitting the application into multiple parts. [3] also has some interesting work on generating SystemC based instruction-set simulators (ISS). This would alleviate some productivity losses encountered when rewriting an ISS. This would also help us to perform multi-processor based SystemC simulation faster than this framework for the demonstration application in this report. [15] uses SystemC based ARM simulator and simulates a system as a single SystemC application. These techniques enable us to do faster simulations than what this framework produces today. But all of them require that a SystemC based ISS be used. But the framework proposed in this paper enables us to perform distributed simulation of an SoC and use existing instruction-set simulators as long as we have its source to compile in the framework with it. None of these works have been evaluated by us against our framework.

8. Future work

This framework is useful in helping us to reuse work done in other processor models like in Simplescalar. But a full system simulation using this framework takes lots of time. Optimisations can be done on improving the efficiency of communication and to reduce the simulation time. One example is that we currently wait for a word-write to complete before letting Simplescalar continue with any other operation. This is not representative of modern super-scalar processers that employ write-buffers. Here, count semaphores can be used such that Simplescalar continues execution after sending a word-write request to the bus. Modern micro-processors also employ outstanding memory loads. Though the SystemC half of the framework allows that today, the other half of the framework does not allow this. It was mentioned section 5.1 that interrupts are not supported in this framework today. That is another area in which this framework can be improved.

We also believe that the simulation framework can be extended to support a distributed SystemC environment similar to [14] by having a *Synchroniser* at the various SystemC applications and treating every other SystemC module as a remote master. Similar approach can be used to add interrupt support to this framework. Other interesting areas to explore would be the applicability of this framework to non-bus based inter-connection networks.

Though this framework was designed for a distributed simulation, it would be interesting to obtain information on the performance of the simulation environment when executed in a true distributed network in which the SystemC application and the Simplescalar applications run on different machines and communicate over the network.

9. Conclusions

There are several benefits to combining SystemC with higher-level processor models like Simplescalar. Performance analysis and early hardware/software integration are the key benefits. This paper presented a framework to achieve this. It still suffers from high running time and thus resulting in slower simulations. There are other work in the research community to achieve the benefits mentioned earlier in different ways. Seamless [5] is one such commercial environment. Other approaches could include generating a new ISS that runs as SystemC modules using tools from [3] or using LISATek [2]. Some companies also provide their own ISS and simulation environment like Tensilica [10]. The framework presented in this paper presents university researchers with an option to work with any chosen high-level processor simulator and do a full system simulation of a multi-processor based SoC using SystemC. It also enables the users to control the type of memory accesses that would be forwarded to the SystemC model and thus letting the user decide the trade-off between speed of simulation and accuracy of the timing results. We believe that if the work mentioned in section 8 is done, then this would be a very good simulation environment that provide embedded system researchers with one more effective full system simulation tool if the speed of the simulation can be improved.

Acknowledgement

This work was done as a project towards CS5271: Performance analysis of embedded systems in National University of Singapore. We thank Dr.Samarjit Chakraborty for the enthusiasm that he shared with us and for referring us to several interesting papers. We also thank him for guiding us through this work and for giving us sufficient time to do the interesting work that is presented in this report.

References

[1] *Independent JPEG Group*. http://www.ijg.org/.

[2] *LISATek from CoWare*. http://www.coware.com/.

[3] *Microlib*. http://www.microlib.org/.

[4] *Open Core protocol*. http://www.ocpip.org.

[5] *Seamless Hardware/Software co-verification, Mentor Graphics*. http://www.mentor.com/seamless/.

[6] *SimpleScalar Home page*. http://www.simplescalar.com.

[7] *SystemC Home page*. http://www.systemc.org.

[8] *SystemC Language Reference Manual*. http://www.systemc.org.

[9] *SystemC User guide*. http://www.systemc.org.

[10] *XTensa ISS and XTMP*. Tensilica, http://www.tensilica.com/html/xtensa_iss_xtmp.html.

[11] J. Bhasker. *A SystemC primer - Second edition*. Star Galaxy Publishing, http://www.stargalaxypub.com, 2004.

[12] K.M.Chandy and J.Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(4):198–206, April 1981.

[13] L. Soulé and A. Gupta. An evaluation of the chandy-misra-bryant algorithm for digital logic simulation. *ACM transactions on Modeling and Computer Simulations (TOMACS91)*, 1(4):308–347, October 1991.

[14] M. Trams. Conservative distributed discrete event simulation with systemc using explicit lookahead. *Digital-Force White paper (http://www.digital-force.net)*, February 2004.

[15] X. Zhu and S. Malik. Using a communication archtiecture specification in an application-driven retargetable prototyping platform for multiprocessing. *Proceedings of 2004 Design Automation and Test in Europe Conference*, February 2004.