A case study on modeling shared memory access effects during performance analysis of HW/SW systems

Marcello Lajolo [*]
Politecnico di Torino
Torino, Italy
lajolo@polito.it

Anand Raghunathan
NEC C&C Research Labs
Princeton, NJ, USA
anand@ccrl.nj.nec.com

Sujit Dey[*]
UC San Diego
La Jolla, CA, USA
dey@ece.ucsd.edu

Luciano Lavagno
Politecnico di Torino
Torino, Italy
lavagno@polito.it

Alberto Sangiovanni Vincentelli
University of California at Berkeley
Berkeley, CA, USA
alberto@eecs.berkeley.edu

Abstract

Behavioral simulation with timing annotations derived from performance modeling and analysis is a promising alternative for use in evaluating system-level design tradeoffs [1, 2]. The accuracy of such approaches is determined by how well the effects of various HW and SW architectural features, like the Real Time Operating System (RTOS), shared memories and buses, HW/SW communication mechanisms, etc *are modeled at this level.*
We present a study of the effects of shared memory buses during system-level performance analysis in the POLIS co-design environment, using the example of a TCP/IP Network Interface System. We demonstrate how the effects of the memory arbiter and shared memory bus can be modeled efficiently at the behavioral level, and used to evaluate various design tradeoffs. Experimental results demonstrate that modeling these effects can significantly increase the accuracy of system-level performance estimates.

1 Introduction

Efficient exploration of system-level design tradeoffs depends heavily on the availability of fast and accurate estimation and modeling techniques, for metrics such as performance, power, and cost, to guide various design decisions. Various techniques have been proposed for performance analysis of hardware [3, 4, 5] and software [6, 7]. In this paper, we focus on performance modeling for mixed HW/SW embedded systems. Hardware-software co-simulation [8] remains the most popular approach to performance estimation for such systems. There are several flavors of hardware-software simulation, with varying degrees of efficiency and accuracy. The techniques that involve simulating (RTL) hardware models of the embedded processor(s) along with the models of the hardware components tend to be the most accurate, but are also the slowest. Moreover, detailed hardware models for embedded processors are often not available to system designers. A popular alternative is to use instruction set simulators (ISS) to simulate the software components of the system, and HDL simulators to simulate the hardware components. Instruction set simulators may be cycle and bit-accurate, or may abstract out some architectural details of the target embedded processor such as pipelines and superscalar ordering for efficiency. The efficiency of this approach may still be limited due to the (assembly or binary instruction) level of detail in software simulation, and the communication overhead required to synchronize the execution of the ISS and hardware simulator. While there has been some work on attempting to reduce the synchronization overhead [9, 10], such approaches are still not very efficient for use in exploring tradeoffs during HW/SW co-design. Bus functional models of the embedded processors may be used to exercise the hardware components without needing to run an ISS concurrently, however, only the hardware functionality is simulated in this approach, making it more suitable for validation of the hardware and HW/SW interface. Using an interface-based design methodology [11] helps separate the behavior of the components from their interface protocols, and allows the use of time and space abstractions for efficient validation and analysis.

Behavioral simulation coupled with timing annotations based on performance modeling techniques offers a promising alternative for use in evaluating system-level design tradeoffs [12, 2]. In such approaches, behavioral models of the software components are simulated, and performance estimates for blocks of code are used to annotate timing information. In the POLIS co-design environment [12], a ho-

[*]This work was started when the authors were at NEC C&C Research Labs, Princeton, NJ

mogeneous behavioral representation is used for hardware as well as software components. The behavioral simulation, analysis, and evaluation is performed using the PTOLEMY heterogeneous simulation environment [13]. Timing information for software modules during simulation is maintained based on performance estimates derived using the technique presented in [1]. The accuracy of behavioral simulation based approaches is determined by how well the effects of various HW and SW architectural features, like the Real Time Operating System (RTOS), shared memories and buses, HW/SW communication mechanisms, *etc* are modeled at this level. For example, the effects of the RTOS are modeled in POLIS during performance analysis, and the user can select between several scheduling policies (*e.g.* round-robin, static priority based, *etc.*) and evaluate their impact on the system performance.

In this paper, we focus on modeling the effects of shared memory buses during system-level performance analysis, using the POLIS co-design environment. The performance of several designs, including graphics and telecommunications applications, may be dominated by memory accesses, making it important to accurately model memory-related effects during system-level design exploration. Using the example of a TCP/IP Network Interface System, we illustrate how the effects of the memory arbiter and shared memory bus can be modeled efficiently at the behavioral level, and used to evaluate various design tradeoffs. Experimental results are presented to indicate that ignoring the effects of the shared memory access bus would have led to significantly incorrect performance estimates, and possibly incorrect design decisions.

The paper is organized as follows. Section 2 provides some background about the TCP/IP Network Interface System used for our study, and the modelling of the system in the POLIS co-design environment. Section 3 presents the results of the evaluation of the effects of the shared memory bus on several design tradeoffs, and section 4 concludes the paper and discusses future work.

2 The TCP/IP System Model

This section provides some background relating to the TCP/IP system, and presents the model used for the system in the POLIS environment.

2.1 Background

A TCP packet consists of three parts:

- An IP header containing, among other fields, the source and destination IP address. The IP header is *usually*, but not always, 20 bytes long,
- A TCP header, containing TCP-specific information. This is usually another 20 bytes,
- The payload, a variable number of bytes (possibly odd) up to a maximum of 65535 bytes.

The TCP/IP protocol requires various tasks to be performed on incoming and outgoing packets, and to maintain the system state. We focus on the evaluation of a dedicated hardware implementation for one of the tasks that is part of the TCP layer - checksum computation. The factors that make this task a good candidate for hardware implementation are explained later.

The IP header is protected by its own 16 bits checksum, that is computed in the IP layer. Since this is computed over such a small number of bytes, it is (relatively) cheap even in software. The TCP data has a 16 bits checksum, carried in the TCP header. It is computed over:

- The 8 bytes of IP address and 16 bits of length field in the IP header,
- The TCP header excluding the 16 bits checksum,
- The payload, taken 16 bits at a time, padding the last byte as NULL if required.

The checksum treats the bytes in pairs, taking each pair of bytes as a 16 bits integer in *big-endian* byte order. Each 16 bits number is added in to the temporary result using unsigned 32 bit integer arithmetic. To obtain the final checksum, the most significant 16 bits of the temporary result are added to the least significant 16 bits, and the result is XOR'ed with $0xffff$.

The checksum computation is particularly inefficient on *little-endian* processors because the *big-endian* 16 bit numbers have to be generated by `shift-or` logic. Also, it is basically a repetitive operation over potentially large volumes of data and contains several bit-level operations. The above factors make the checksum computation a good candidate for hardware implementation. We attempted to model parts of the TCP/IP system relating to the checksum computation using POLIS with the motivations of quantitatively evaluating (i) the performance improvement obtained by implementing the checksum computation in HW, and (ii) the

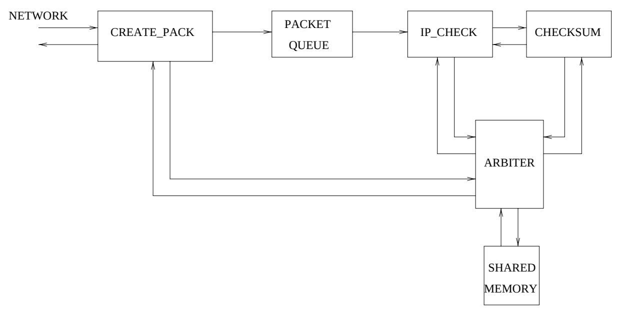

Figure 1. The modeled TCP/IP sub-system

possible adverse effects of SW and HW processes conflicting for accessing the shared packet memory. However, we believe that the effects of shared memory access on system-level performance evaluation that we present are applicable to any HW/SW system, and not limited to the design example or HW/SW configuration used for this study.

2.2 Modeling the TCP/IP subsystem in POLIS

Figure 1 shows the sub-system that has been described in POLIS for our case study. The system was modeled as ten interconnecting CFSMs, each specified in ESTEREL, and their interconnection was described graphically with the Ptolemy user interface.

For incoming packets, the module `create_pack` receives a packet from the lower layer (in this case, the IP layer), and stores it in the shared memory. When it finishes, it sends the information about the starting address of the packet in memory, the number of bytes and the checksum header to a queue (`packet queue`). From this queue, the module `ip_check` retrieves a new packet, overwrites parts of the checksum header (which should not be used in the checksum computation) with 0s, and signals to the `checksum` process that a new packet can be checked for checksum consistency. The `checksum` process performs the core part of the checksum computation, accessing the packet in memory through the arbiter and accumulating the checksum for the packet body. When it is done, it sends the computed 16-bit checksum back to the `ip_check` process, which then compares the computed checksum with the incoming transmitted checksum, and flags an error if they do not match. The flow for outgoing packets is similar, but in the reverse direction, and there is no need for comparison of the final checksum.

2.3 Behavioral Model of the Memory Bus and Arbiter

In the original behavioral description that was used to validate the functionality of the processes, memory accesses were modeled by access to a global array, using a C function call from Esterel, *i.e.* the module `arbiter` shown in Figure 1 was not present. However, as we show in Section 3, using the same model for performance evaluation suffers from the drawback of ignoring effects such as shared memory access conflicts, block access mode (DMA), *etc.* Hence, we described a behavioral model of the shared bus and memory arbiter (shown as module `arbiter` in Figure 1) to model the effect of the controller (arbiter) of the shared memory bus. The `arbiter` module is the only module that can access the shared memory: it receives requests from the processes `create_pack`, `ip_check` and `checksum`, and is responsible for deciding which module is given access to the memory. The functional model of the arbiter is such that the access priority scheme can be easily changed or parametrized. For example, we may specify that in the case of simultaneous requests, the arbiter should give higher priority to `checksum` and lower priority to `create_pack`.

In our system, the primary functions of the arbiter are: (i) to avoid multiple components simultaneously driving the bus in an attempt to access memory using a simple request-grant protocol, (ii) to resolve simultaneous access attempts based on priorities that can be specified by the designer, (iii) to allow components to request dedicated access of the memory bus for a certain number of bus cycles (block access mode or DMA mode). We have created a behavioral model of the arbiter and shared memory bus in Esterel that is called `arbiter` in Figure 1. The `arbiter` process has a dedicated interface to each of the processes that require to access memory, that can be similar to, or an abstraction of, the shared memory bus interface. In addition, each process that accesses memory is enhanced to include an arbiter interface. For example, the signals that interface the `arbiter` process to the `checksum` process are shown in Figure 2. The interface consists of a memory access

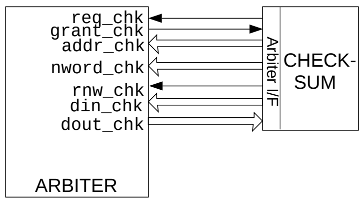

Figure 2. The interface of the arbiter model

request signal req_chk on which the `checksum` process generates an event to indicate that it would like to access memory. The starting address is placed on signal $addr\_chk$, and a block size signal $nword\_chk$ is used (in DMA or block access mode) to convey the number of bus cycles of dedicated bus access requested. The arbiter generates and event on the signal $grant\_chk$ to indicate that the request has been granted. In addition, there are data in, data out, and read/write signals to the memory.

A part of the Esterel specification of the `arbiter` process is shown in Figure 3. Signals req_create, req_ip, and req_check represent the requests for access to the memory bus from the `create_pack`, `ip_check`, and `checksum` processes, respectively. Note that the behavior of the arbiter is described as an infinite loop which immediately encloses a set of nested $if - then - else$ statements that test for the presence of events on the various memory access request signals. The code within this set of $if - then - else$ statements represents the actions to be taken for processing a memory access request from the corresponding module. Figure 3 only shows the code for processing a memory access request from the `checksum` process, the parts for han-

```
....
loop
    if (?req_create=1) then
         ....
         ....  % grant access to create_pack
    elsif (?req_ip=1) then
         ....
         ....  % grant access to create_pack
    elsif (?req_chk=1) then
      i:=?addr_chk;
      emit grant_chk;
      repeat ?nword_chk times
        if (?rnw_chk=false) then
          await din_chk;        % memory write
          emit addr(i); emit din(?din_chk);
          emit rnw(?false);
        else
          emit addr(i);         % memory read
          emit din(?din_chk);
          emit rnw(?true);
          await din_mem;
          emit dout_chk(?din_mem);
        end if;
        i:=i+1;
      end repeat;
      emit res_chk;
    end if;
end loop;
....
```

Figure 3. Esterel model of the `arbiter` process

dling requests from other modules are similar. The priorities given by the arbiter to requests from the various processes are determined by the order in which the request signals are tested in the nested $if-then-else$ statements. For example, the code shown in Figure 3 gives highest priority to requests from `create_pack`, since the signal req_create is tested for an event first. Thus, changing the memory access priorities of the processes can be achieved by simply re-ordering the testing of the access request signals in the behavioral arbiter model.

We would like to reiterate that the behavioral arbiter model shown above is not part of the system specification - it was added to model the effects of the shared memory bus and memory arbiter during behavioral level performance simulation. However, during the performance simulation, it is treated just like any other module. The implementation of the `arbiter` process is specified to be HW, because it allows us finer control of its timing properties. The number of memory access cycles, and processing time taken by the arbiter, can be easily modeled using $await\ tick$ statements appropriately inserted in the behavioral model.

3 Performance Simulation and Experimental Results

In the POLIS environment, the system specification, which may consist of a PTOLEMY netlist that describes the interconnection of the functional components or modules and an Esterel specification that describes the functionality of each module, is translated into a network of co-design finite state machines (CFSMs), which are extended FSMs with asynchronous buffered communication. Performance analysis is carried out using the heterogeneous simulation environment offered by PTOLEMY [13]. The performance simulation is based on a C model of each CFSM that is automatically generated, using the hardware/software partitioning specified by the user, the scheduling policy for the RTOS specified by the user, and a timing model for the target processor that is derived during a characterization step [12]. We simulated the TCP/IP subsystem with network traffic that was captured using a profiling tool from an existing software implementation of the TCP/IP protocol.

We performed several experiments to demonstrate the value added by our behavioral model of the arbiter and shared memory bus during system-level design, some of which we present here.

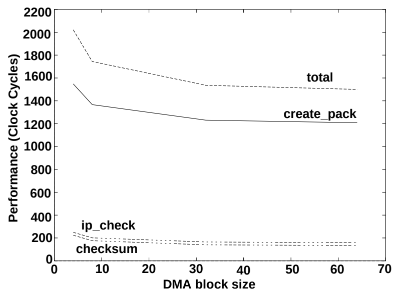

Figure 4. Variation of computation times with DMA block size

In the first experiment we performed an analysis of the variation of the processing times for each module as well as the complete per-packet processing time for the entire system for various sizes of the DMA block size used for memory access. For this experiment, the `create_pack` process was mapped to software running on a MIPS R3000 processor, and the `checksum` and `ip_chk` processes were mapped to hardware. Figure 4 shows the variation of (average) per-packet processing times for the three processes for a test bench consisting of three packets of length 512, 64, and 448 bytes, for block sizes of 4, 8, 32, and 64 bytes. The following conclusions can be drawn from Figure 4:

- As expected, the processing times for all the modules as well as the total processing time decrease with increasing DMA block size, since the handshaking overhead required to obtain memory access is amortized over a

Table 1. Processing times without memory conflicts

packet #	create_pack	ip_check	checksum	total
1	513	1101	1088	2702
2	65	149	136	350
3	449	965	952	2366

larger number of data transfers. The decrease is significant at lower DMA block sizes.

- In addition, the sensitivity of the performance of the software module (`create_pack`) to DMA block size is higher, since the time required for handshaking with the arbiter is much higher for the software module than for the hardware modules.

Note that it would have been impossible to perform the above analysis in the absence of the behavioral model of the shared memory bus and arbiter, since the reported processing times would be constant for various values of block size.

The next experiment we performed was to evaluate the effect of memory conflicts due to the shared memory bus on the performance of the individual processes as well as the overall system performance. The performance estimates without and with memory conflicts are presented in Tables 1 and 2, for a sequence of three packets (512, 64 and 448 bytes long) that are part of a longer stream. The performance estimates without memory conflicts were obtained by not including the arbiter process, and modeling memory as an array shared between the `create_pack`, `ip_check` and `checksum` processes. Access to the shared array is performed using a C function call annotated with a fixed delay to represent the access time of the memory.

Table 2. Processing times with memory conflicts

packet #	create_pack	ip_check	checksum	total
1	513	1617	1538	3688
2	65	218	192	475
3	449	1418	1346	3213

The results indicate that:

- The performance of the `create_pack` process was not affected by the presence of memory conflicts. This is because the memory arbiter gives highest priority to requests from `create_pack` when simultaneous or pending requests are present.

- The per-packet performance estimates of the `ip_check` and `checksum` processes are in error (underestimates) by 46.9% and 41.4%, respectively if memory conflicts are ignored, and the total performance of the system is underestimated by 36.39%

It is clear from the above results that the effects of memory conflicts due to the use of shared memory and the DMA block size need to be considered while estimating the performance of HW/SW systems.

4 Conclusions and Future Work

We presented a case study to study the effects of shared memory buses and arbiters during system-level performance analysis. Using the case study of a part of a TCP/IP network interface system, we have proposed a methodology to model the shared memory bus and arbiter at the behavioral level. We presented experimental results to demonstrate that ignoring these effects leads to a large error in system-level performance estimates, and that the effects of some design tradeoffs cannot be evaluated without modeling memory effects accurately. We are currently working on automatically generating the models required to incorporate the effects of the shared memory bus and memory arbiter during performance analysis of HW/SW systems.

Acknowledgements: The authors would like to thank Leslie French and Toshio Misawa of NEC C&C Research Labs for providing the software implementation of the TCP/IP system, and for useful technical discussions.

References

[1] K. Suzuki and A. Sangiovanni-Vincentelli, "Efficient software performance estimation methods for hardware/software codesign," in *Proc. Design Automation Conf.*, pp. 605–610, June 1996.

[2] B. Tabbara and L. Lavagno and A. Sangiovanni-Vincentelli, "Fast Hardware-Software Co-simulation Using Software Synthesis and Estimation," in *Proc. Int. High Level Design Validation and Test Wkshp.*, pp. 149–156, Nov. 1997.

[3] S. Bhattacharya, S. Dey, and F. Brglez, "Performance analysis and optimization of schedules for conditional and loop-intensive specifications," in *Proc. Design Automation Conf.*, pp. 491–496, June 1994.

[4] M. Rahmouni and A. Jerraya, "Formulation and evaluation of scheduling techniques for control flow graphs," in *Proc. European Design Automation Conf.*, Sept. 1995.

[5] S. Dey and S. Bommu, "Performance analysis of a system of communication processes," in *Proc. Int. Conf. Computer-Aided Design*, pp. 590–597, Nov. 1997.

[6] S. Malik, M. Martonosi, and Y. T. S. Li, "Static Timing Analysis of Embedded Software," in *Proc. Design Automation Conf.*, pp. 147–152, June 1997.

[7] R. Ernst and W. Ye, "Embedded program timing analysis based on path clustering and architecture classification," in *Proc. Int. Conf. Computer-Aided Design*, pp. 598–604, Nov. 1997.

[8] J. Rowson, "Hardware/Software Co-Simulation," in *Proc. Design Automation Conf.*, pp. 439–440, June 1994.

[9] "Mentor Graphics Seamless CVE Home Page (http://www.mentorg.com/seamless/).".

[10] S. Yoo and K. Choi, "Synchronization Overhead Reduction ini Timed Cosimulation," in *Proc. Int. High Level Design Validation and Test Wkshp.*, pp. 157–164, Nov. 1997.

[11] J. Rowson, "Interface Based Design," in *Proc. Design Automation Conf.*, pp. 178–183, June 1997.

[12] F. Balarin, M. Chiodo, H. Hsieh, A. Jureska, L. Lavagno, C.Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara, *Hardware-software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, Norwell, MA., 1997.

[13] J. Buck, S. Ha, E. Lee, and D. Masserchmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *International Journal on Computer Simulation, Special Issue on Simulation Software Management*, Jan. 1990.