Seamless Hardware/Software Performance Co-Monitoring in a Codesign Simulation Environment with RTOS Support

L. Moss[1], M. de Nanclas[1], L. Filion[1], S. Fontaine[1], G. Bois[1] and M. Aboulhamid[2]

[1] Department of Computer Engineering, École Polytechnique de Montréal
[2] Department of Computer Science and Operational Research, Université de Montréal
{moss, denancla, filion, fontaine, bois}@grm.polymtl.ca, aboulham@iro.umontreal.ca

Abstract

Simulation monitoring tools are needed in hardware/software codesign for performance debugging, model validation and hardware/software partitioning purposes. Existing tools are either hardware- or software-centric and lack integrated and seamless co-monitoring. This paper presents a system-level co-monitoring tool that can monitor the computation and communication activities of SystemC user modules, as well as bus, memory and processor usage, on a variety of hardware/software embedded configurations that may include an RTOS. We also describe how performance metrics are generated during or after simulation and made accessible to users or external applications. Finally, experimental results show that such co-monitoring does not disturb the simulation's internal timing and only moderately increases the simulation's wall clock run time (by 11-22% for hardware/software partitioned architectures).

1. Introduction

Hardware/software codesign is an increasingly popular methodology used to design embedded systems at the system level by simultaneously developing the system's hardware and software user modules. This methodology needs to be supported by monitoring tools that are able to gather data and metrics on the performance of embedded systems simulated in a hardware/software codesign environment. Such monitoring tools can be used for performance debugging purposes and for design space exploration, either by human designers analyzing design performances or by automated hardware/software partitioning algorithms. These tools can also be used to validate high-level performance estimation methods or to feed data into a performance estimation model, as in [1].

Our SPACE platform, which has been thoroughly explained in [2] as a SystemC [3] virtual platform for the exploration of hardware/software embedded systems including real-time operating systems (RTOS), was instrumented in order to allow for performance co-monitoring. This platform supports the functional partitioning of a system specification into several SystemC modules and their mapping to hardware and software partitions. The instrumentation presented in this paper allows for the collection of metrics on the execution time of hardware user modules and on channel/bus, memory and processor usage. Also, a non-intrusive instrumentation of instruction set simulators (ISS) monitors RTOS context switches and calls to communication functions. This powers the collection of metrics on the execution time of software user modules and on communications between all user modules, no matter whether the communications are between hardware modules, between software modules or are crossing the hardware/software partition. This ability to seamlessly monitor SystemC user modules, both in software and hardware, is what we call *co-monitoring*.

Since the publication of [2], our virtual platform has seen several enhancements; one of these is the integration of a bus cycle accurate (BCA) protocol in order to suitably reflect the true behavior of a processor. Also, a collection of support IP blocks, such as timers, UARTs and memory controllers, is now provided. These components simulate platform behaviors that are similar to real implementations.

An important feature of our virtual platform, exploited in this paper, is that all communications keep the same application programming interface (API) at anytime and any location in the virtual platform, while seamless movement of user modules between hardware and software is allowed. Communications are based on fixed IDs attributed to every single user module, be it mapped to hardware or software. Hence, the code written to communicate to a module never changes, regardless of

978-3-9810801-2-4/DATE07 © 2007 EDAA

where the module is (hardware or software) or if it changes partition from one simulation to another. By keeping the same API, users can really explore their design with minimum effort. From a monitoring point of view, this ensures full transparency of communications and a simplification of the monitoring engine.

This paper's main contribution is a novel RTOS-aware co-monitoring architecture that seamlessly and non-intrusively monitors both hardware and software user modules' computations and communications. In other words, our co-monitoring yields the same kind of performance data for every user module independently of the hardware/software partitioning of the system, without disrupting the simulation's internal timing and without requiring any changes in user module source code when modules are moved between hardware and software partitions. Finally, an efficient and flexible mechanism to compute, store and access a history of performance metrics is described.

2. Related work

There are commercial tools, such as Mentor Graphics Seamless [4], Coware Platform Architect [5] and Xilinx Monitor [6], that offer cycle-accurate monitoring information on hardware user modules, bus usage, memory accesses and processors. However, they lack sufficient visibility into the system-level operations of software user modules. The software binary is either treated as a black box in which the software user modules are melded together, for instance when collecting statistics on cache usage, or metrics are collected at a too low level, for example in tracking when the processor enters or exits each function. The interpretation of the semantics of these function calls is then entirely left to the user. In particular, these tools do not automatically monitor RTOS context switches and communications made by software user modules, and the kind of performance data produced depends on the module's location in hardware or software.

There exists in the literature some ISS-based works that allow flexible monitoring of software user code without disturbing its execution or distorting its simulated run time. These tools typically count the number of times a given software region has been executed and monitor the time periods spent in it. For instance, FLATSIM [7] is a profiling tool that automatically tracks the entry and exit points of both functions and loops whereas the Comet Profiler [8] tracks the execution of software code blocks specifically labeled in user source code. While these two tools give an estimation of what the performance of the monitored software regions would be if they were instead implemented in hardware, they do not offer a cycle-accurate simulation of a partitioned hardware/software system and do not monitor memory accesses, bus transactions and end-to-end module communications inside and between partitions.

The advantage of our co-monitoring is that it combines the advantages of both approaches, simultaneously monitoring hardware user modules, buses and memory devices as well as software user modules, without requiring modifications in user code both when enabling monitoring and when re-partitioning the system.

3. Co-monitoring architecture

Our co-monitoring is based on the general framework illustrated in Figure 1. First, the virtual platform is instrumented in order to send event notifications to a central SystemC monitoring component. This component processes the events to assemble co-monitoring records and forwards these records to a log file or to a metrics generator outside of the SystemC simulation. During or after the simulation, the metrics generator computes performance metrics and statistics over these records and sends the results to a graphical user interface (GUI), or to another program using the performance analysis API.

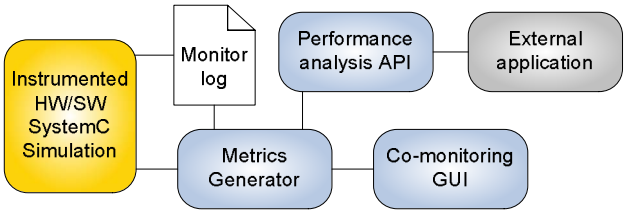

Figure 1: General Co-monitoring Framework

3.1. Co-monitoring records

Our co-monitoring framework is based on the following basic record types:

End-to-end transfer: This record type gives information about a completed end-to-end communication between a user module and another component, which may be a second user module. This information includes the identification of source and destination components, the start and stop time of the read operation at one end and of the counterpart write operation at the other end, and the data length of the transfer.

Bus transfer: This record type contains information about the transfer of a data packet on a bus, such as the identity of the bus, the time at which the packet arrives in the bus, the times at which it starts and ends to be transferred, the source and destination components as well as its data length. A single end-to-end transfer may generate more than one bus transfer, because the packet may need to travel on more than one bus to reach its destination (for instance, in multi-processor systems) and

because the communication protocol may require an acknowledgment to be sent.

Memory access: This collects information related to a read or a write access to a memory, particularly the identity of the accessing and accessed components as well as the access time, address and length.

User module computation: This represents a user module's computation phase and gives the computation's begin and end time. For software modules, such a record also contains the number of instructions, broken down by type, executed during this computation. Note that this is the only case where record contents differ according to the hardware or software location of user modules. The information on instruction frequencies by type can be used to estimate the degree of parallelism as well as the data- or control-orientation of user modules [9].

3.2. Platform instrumentation

Different approaches must be taken for the instrumentation of hardware and software components. The difference is due to the fact that hardware components can do arbitrarily complex operations without affecting the SystemC simulation clock time (though they do take non-zero wall clock time) whereas software instructions are executed in a cycle-accurate manner on an ISS. To preserve the simulation cycle-accurate timing, we must ensure that the simulation clock time is not distorted by the monitoring of software user modules. Figure 2 gives a general picture of the platform instrumentation.

In order to minimize the amount of code inserted into platform models, generic instrumentation macros were implemented. This also allows co-monitoring to be easily enabled or disabled at compile time. These macros are responsible for collecting data on significant events and passing that information to the central monitoring component, which generates and timestamps the monitoring records presented in the previous section.

The instrumentation of hardware components is straightforward, since the relevant macros can be directly inserted into their respective platform models. Thus, bus models and memory models are instrumented by their designer whereas this is done automatically for hardware user modules through the instrumentation of our virtual platform's API functions. This instrumentation also takes advantage of timing annotations that are present in user modules, whether they were inserted manually, through a behavioral synthesis tool or via estimation techniques.

As explained above, software user modules cannot directly call monitoring macros, since doing so would mean executing additional software instructions and disrupting the simulation's timing. Each ISS is instead given an ISS monitoring component which calls the macros on behalf of the software user modules. Because the ISS monitor is itself a SystemC component, it can fulfill its role without distorting the simulation clock time.

In our virtual platform, all user modules in all applications call up to four API functions for their communication operations, which are to read from or write to a device or another user module. These API calls are independent of both the target processor and the embedded RTOS used. Software modules start and stop their computations when their context is switched in and out by the RTOS scheduler. The functions used for context switching depend only on the RTOS, not on the processor on which it runs or on the user model. This limits the effort required for implementing the software monitoring environment, since there is little variation in the functions to look for.

The ISS monitor tracks the ISS program counter to detect when the execution flow reaches an entry or exit point in one of these functions. The ISS monitor then takes action depending on the event that occurred. For instance, if a communication function has just been entered, the ISS monitor retrieves from the stack and registers the parameters passed to the function and then transfers the information about the started communication to the central monitor. If a context switch has been requested, the ISS monitor will similarly retrieve the IDs of the user modules being switched in and out, and will thus keep track of the user module currently being executed by the ISS. If the end of a communication function has been reached, the ISS monitor uses our virtual platform's communication model to determine which communication has just been terminated. In every case, the central monitor uses the information extracted by the ISS monitor to produce appropriate co-monitoring records.

The co-monitoring instrumentation is always automatically adjusted for the simulation of any new system partitioning, each time a user module is moved from hardware to software (or vice-versa), meaning that a hardware user module becomes a software user module (or vice-versa) after re-partitioning.

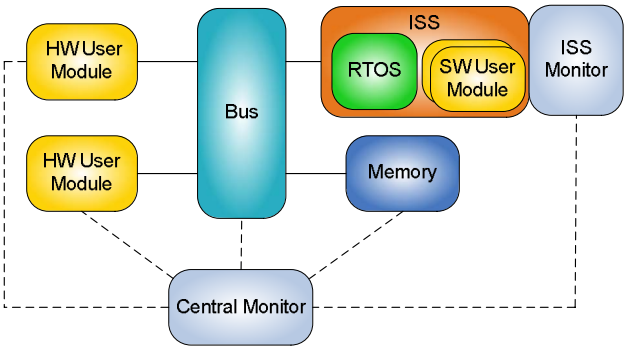

Figure 2: Platform instrumentation. Dashed lines represent monitoring components' lines of sight

3.3. Metrics generation

The co-monitoring records produced by the instrumented SystemC simulation are passed on to a metrics generator either synchronously through socket communication or asynchronously through the file system. In addition to being platform-independent and language-independent, this interprocess communication method enables both local and remote analysis.

The metrics generator processes the records to generate metrics about bus, memory and processor usage, channel transfer rates, read and write communications between user modules (minimum, maximum, average and standard deviation of end-to-end communication time as well as the amount of data transferred), and execution times for each user module. The generation of each of these metrics can be turned on and off and a performance analysis API makes them available to external programs, for example a hardware/software partitioning tool.

These metrics can be computed over the full length of the simulation or subdivided into a series of time-based analysis intervals whose length is adjustable at runtime. When using analysis intervals, each metric is computed for each interval using all records comprised within the interval. Figure 3 gives an example of a metric, end-to-end communication times between two given modules, computed with analysis intervals of 50 microseconds.

To avoid large memory space to be taken by the records, especially for long simulations, records are pre-processed into atomic intervals so that each record needs not be kept in memory, except when detailed Gantt charts are required. Metrics and statistics (maximums, minimums, averages, sums and standard deviations) are computed for each atomic interval, whose time span is fixed before the simulation starts. This method allows the re-generation of metrics and statistics for different interval lengths at runtime, by combining an integral number of atomic intervals into a larger analysis interval. This can be used for starting with a coarse-grained plot and progressively decreasing the analysis interval length in order to zero in on interesting points.

3.4. Metrics presentation

Graphical user interfaces for performance metrics are described here in order to illustrate the possibilities offered by co-monitoring and how it can be integrated into a larger hardware/software codesign framework.

A GUI, fed by the metrics generator, provides several ways to visualize collected data, primarily geared toward performance analysis. Time-based plots show statistics on data transiting on buses, on end-to-end transfers between components, on memory accesses and on processor usage. Heatmap views also illustrate how much communication occurs between every pair of components and how much different memory regions are accessed per analysis interval. Gantt charts can also be used to have a more detailed view of the activities of buses, memory devices and hardware and software user modules. These graphs make it possible to pinpoint bottlenecks and heavily used segments of memory, to find deadlocks, to look for long blocking requests, and more. Figures 3 and 4 are screenshots from this GUI and respectively depict a time-based plot and a heatmap view.

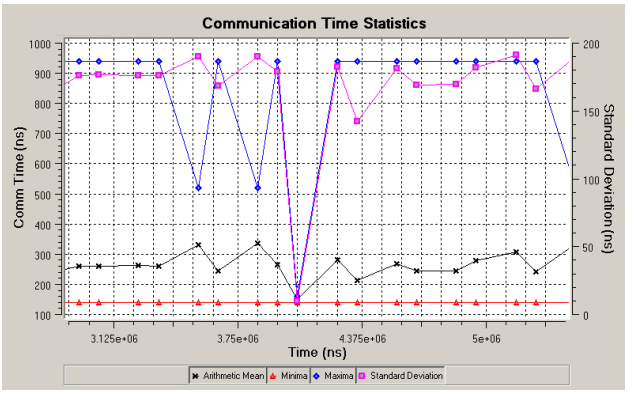

Figure 3: Statistical graph of end-to-end communication times between two user modules

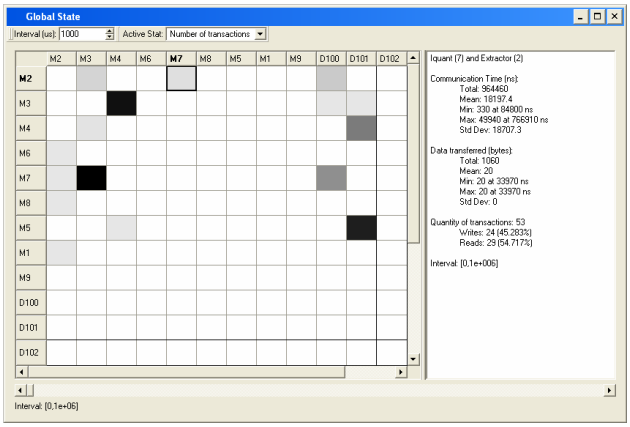

Figure 4: Heatmap of the number of transactions between each pair of components in a simulation

4. Experimental results

The SPACE virtual platform [2] features three levels of abstraction, two of which are untimed and are used for validating system specifications and untimed functional partitionings. The third one offers timed functional (TF) and cycle-accurate simulations of partitioned systems. Performance co-monitoring was implemented in the latter level of abstraction for three different kinds of architectures. The first one is an all-HW TF architecture in which user modules are all put into the HW partition and communicate through a virtual TF FIFO channel. The second one is an all-HW architecture in which

communications travel on a BCA model of the IBM CoreConnect™ OPB bus [10]. The last one, which exploits the full power of hardware/software co-monitoring, uses a BCA OPB bus, a Xilinx MicroBlaze [11] ISS and a µC/OS-II RTOS kernel [12] to provide a cycle-accurate simulation of bus communications and of software execution. Results are presented here on the co-monitoring of a digital RF filter and of a JPEG picture processor. All our simulations were run on a Windows XP workstation with a 2.8 GHz Pentium IV processor, a 7200 RPM hard drive and 1 GB of RAM.

4.1. Digital RF filter

The digital RF filter example has been thoroughly presented in [2] and is shown in Figure 5. In short, the Producer generates a periodic flow of data that is filtered and then stored in memory by the Mux. The Controller periodically adjusts the memory address used by the Mux, requests analysis of the data and changes the Filter's coefficients according to the results of the analysis.

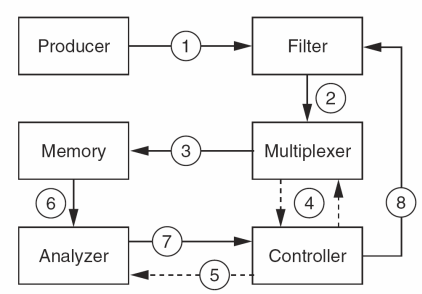

Figure 5: RF filter. Dashed lines represent control signals, solid lines are data transfers

Architecture	Comm. channel	HW partition	SW partition
HW-TF	TF	All	None
HW-BCA	OPB	All	None
PART1	OPB	Producer, Filter, Mux, Analyzer	Controller
PART2	OPB	Producer, Filter, Mux, Controller	Analyzer
PART3	OPB	Producer, Filter, Mux	Controller, Analyzer

Table 1: Five architectures of the RF filter

We simulated five different architectures for the RF filter, three of which were partitioned between hardware and software, as shown in Table 1. Each architecture was simulated with and without co-monitoring in order to measure how much overhead is added when instrumenting the SystemC simulation and outputting the monitoring records to a log file. Table 2 compares the SystemC and wall clock times taken by each simulation.

Our tests confirm that the simulation's SystemC clock time stays exactly the same whether monitoring is enabled or not. This means that our co-monitoring does not disturb the timing of system-level simulations. On the other hand, co-monitoring does increase the wall clock run time of the simulation, which means that the user has to wait longer before the simulation ends.

	Wall clock run time (seconds)		SystemC clock time (seconds)
Architecture	Plain	Monitored	Plain = Monitored
HW-TF	16	23	0.18976762
HW-BCA	24	32	0.36211836
PART1	98	115	1.02003820
PART2	198	220	2.13268552
PART3	284	316	3.01680632

Table 2: Results for five architectures of the RF filter example with and without co-monitoring

Communication path (as shown in Figure 5)	Communication volume (kB)
(1) Producer→Filter	256.000
(2) Filter→Mux	256.000
(3) Mux→Memory	256.000
(4) Controller↔Mux	0.570
(5) Controller→Analyzer	0.285
(6) Memory→Analyzer	580.277
(7) Analyzer→Controller	0.277
(8) Controller→Filter	1.688
Instruction and data memory→ISS	96579.404

Table 3: Communication volumes in the RF filter, with the Analyzer and Controller in SW (PART3)

For all-HW architectures, the absolute size of this increase is small (7-8 seconds) whereas its relative size is significant (33-44% of the non-monitored wall clock run time), which reflects the fact that all-HW SystemC simulations are very fast. For partitioned architectures, the absolute overhead is larger (17-32 seconds) whereas the relative overhead is modest (11-17%). The larger absolute overhead in partitioned architectures is due to the operations of the ISS monitor and to the monitoring of every data and instruction memory access made by the ISS (see last line of Table 3), whereas the smaller relative overhead reflects the fact that an ISS simulation is inherently much more time-consuming than an all-HW SystemC simulation.

Also, Table 3 shows that co-monitoring is able to measure the communication volume between a pair of components, whether they are both in hardware (paths 1, 2 and 3 in Figure 5), both in software (paths 5 and 7) or are split between HW and SW (paths 4, 6 and 8).

4.2. JPEG picture processor

The JPEG picture processor example has been detailed in [2] as an edge detection application for JPEG images. Figure 6 illustrates the components and communication paths for this example.

Like in the preceding example, the picture processor was simulated with and without co-monitoring for five architectures, as shown in Table 4. Each simulation was run for 9 JPEG images with a dimension of 128x128. Table 5 summarizes our findings. Again, the wall clock overhead is smaller in absolute terms (8-11s) and larger in relative terms (33-42%) for all-HW architectures, and the reverse is true for partitioned architectures (96-200 seconds or 14-22% of the wall clock run time).

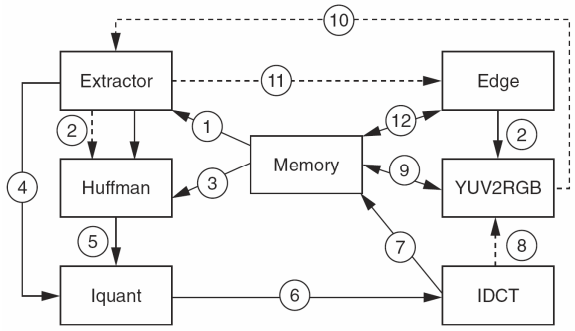

Figure 6: JPEG picture processor example

Architecture	Comm. channel	HW partition	SW partition
HW-TF	TF	All	None
HW-BCA	OPB	All	None
PART1	OPB	HUFF, IDCT, YUV2RGB, Edge, Iquant	Extractor
PART2	OPB	HUFF, IDCT, Extractor, Edge, Iquant	YUV2RGB
PART3	OPB	HUFF, IDCT, Edge, Iquant	Extractor, YUV2RGB

Table 4: Five picture processor architectures

	Wall clock run time (seconds)		SystemC clock time (seconds)
Architecture	Plain	Monitored	Plain = Monitored
HW-TF	19	27	0.15102666
HW-BCA	33	44	0.36296708
PART1	494	604	4.69517290
PART2	683	779	7.27851472
PART3	1169	1369	11.84522752

Table 5: Results for five picture processor architectures with and without co-monitoring

5. Conclusions

This paper explained how to get, with relatively little overhead, a rich set of performance data from an embedded system simulated in a hardware/software codesign environment. In particular, it demonstrated how the activities of hardware and software user modules can be seamlessly and non-intrusively monitored inside and across partition boundaries and how performance metrics and statistics could be efficiently extracted from monitoring records obtained from the instrumented model.

Future work remains ahead in the refinement of this co-monitoring tool. Monitoring for software user modules at a higher level of abstraction, in which the software is not run on an ISS but rather inside a host process for faster simulation, is still under development. Also in the works is a hardware/software partitioning application that would make use of co-monitoring to drive the partitioning algorithm. Finally, the technique used for co-monitoring is currently employed to implement a non-intrusive ISS-based RTOS monitoring tool.

References

[1] K. Ueda, K. Sakanushi, Y. Takeuchi, and M. Imai, "Architecture-level performance estimation method based on system-level profiling," *IEE Proceedings-Computers and Digital Techniques*, vol. 152, no. 1, 2005, pp. 12-19.

[2] J. Chevalier, M. de Nanclas, L. Filion, O. Benny, M. Rondonneau, G. Bois, and M. Aboulhamid, "A SystemC Refinement Methodology for Embedded Software," *IEEE Design & Test of Computers*, vol. 23, no. 2, 2006, pp. 148-158.

[3] "SystemC," Open SystemC Initiative, www.systemc.org

[4] "Seamless," Mentor Graphics Corp, www.mentor.com

[5] "CoWare Platform Architect," CoWare Inc., www.coware.com

[6] A. Donlin and T. Lenart, "Performance Analysis and Visualization of SystemC Models," presented at *5th North American SystemC User Group Meeting*, San Jose, 2006.

[7] D. C. Suresh, W. A. Najjar, F. Vahid, J. R. Villarreal, and G. Stitt, "Profiling tools for hardware/software partitioning of embedded applications," *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, 2003, pp. 189-198.

[8] M. Finc and A. Zemva, "Profiling soft-core processor applications for hardware/software partitioning," *Journal of Systems Architecture*, vol. 51, no. 5, 2005, pp. 315-29.

[9] M. Holzer and M. Rupp, "Static Code Analysis of Functional Descriptions in SystemC," *Third IEEE International Workshop on Electronic Design, Test and Applications (DELTA'06)*, 2006, pp. 243-248.

[10] "On-Chip Peripheral Bus," IBM, www.ibm.com

[11] "MicroBlaze Soft Processor Core," Xilinx Inc., www.xilinx.com

[12] "µC/OS-II," Micrium Inc., www.micrium.com