System-Level Performance Analysis in SystemC[1]

H. Posadas*, F. Herrera*, P. Sánchez*, E. Villar* & F. Blasco**

*TEISA Dept., E.T.S.I. Industriales y Telecom. University of Cantabria
Avda. Los Castros s/n, 39005 Santander, Spain
{posadash, fherrera, sanchez, villar}@teisa.unican.es

**DS2 Robert Darwin 2, Parque Tecnológico, Paterna, Spain
francisco.blasco@ds2.es

Abstract

As both the ITRS and the Medea+ DA Roadmaps have highlighted, early performance estimation is an essential step in any SoC design methodology [1-2]. This paper presents a C++ library for timing estimation at system level. The library is based on a general and systematic methodology that takes as input the original SystemC source code without any modification and provides the estimation parameters by simply including the library within a usual simulation. As a consequence, the same models of computation used during system design are preserved and all simulation conditions are maintained. The method exploits the advantages of dynamic analysis, that is, easy management of unpredictable data-dependent conditions and computational efficiency compared with other alternatives (ISS or RT simulation, without the need for SW generation and compilation and HW synthesis). Results obtained on several examples show the accuracy of the method. In addition to the fundamental parameters needed for system-level design exploration, the proposed methodology allows the designer to include capture points at any place in the code. The user can process the corresponding captured events for unrestricted timing constraint verification.

1. Introduction

Performance analysis is becoming a very important and challenging task in embedded system design, where performance parameters (time, size, consumption, cost, etc.) can be as important as functional requirements [1-3]. Early and accurate performance estimation would avoid costly design process iterations.

Among the different performance figures, timing properties (execution times, delays, periods, etc.) are especially important in performance verification of multiprocessing, real-time embedded systems [4]. Deciding the most appropriate scheduling policy for each processor is critical to ensure the correct real-time behavior of the whole system [5]. Average interexecution time estimation is needed in rate analysis for embedded systems [6].

Performance verification has been proven particularly complex in heterogeneous, multiprocessing embedded systems under multi-rate dependencies and variable rate intervals [7-8]. All these techniques rely on accurate timing estimation figures.

Timing estimation is required for timed co-simulation [9-10]. Timed simulation is particularly important for design verification and evaluation at the system level once the architectural mapping has been decided.

Time execution estimation has been a traditional problem in real-time embedded SW engineering [11]. Estimation techniques can be divided in two main groups: static and dynamic techniques. Static techniques [12-14] are based on the formal analysis of a specification without executing it. They mainly pursue worst-case estimation time (WCET), assuming a static scheduling in order to avoid an exponential increase of complexity. Moreover, when an Object-Oriented language such as SystemC is used, polymorphism at execution time may make the code practically unpredictable at compilation time. On the other hand, dynamic techniques [15-17] provide feasible methods that keep into acount dynamic scheduling and give more accurate estimations tuned to the application.

SW execution time can be estimated from the assembler code [9], the compiler internal representation [10], a Virtual Processor Instruction Set [15] or the source code [13-14][16-17]. In all these cases, execution time estimation of the code and timed simulation are performed in separate steps.

A traditional problem in SW execution time estimation is the effect of cache memories. Static timing analysis for instruction caches has provided very good results [18]. These techniques can be extended easily to dynamic estimation based on instruction flow analysis. Nevertheless, some error percentage is unavoidable which may require providing confidence intervals [17].

[1] This work has been partially funded by the MEDEA A511 ToolIP and the Spanish MCYT-TIC-2002-00660 projects.

1530-1591/04 $20.00 (c) 2004 IEEE

Specification at system level has been identified as one of the most important ways to increase design productivity. Several system-level design languages (SLDL) have been proposed. Among them, those based on C/C++, i.e. SpecC [19] and SystemC [20] are gaining wider acceptance. Design flows based on these SLDLs need new estimation techniques in order to allow a fast and accurate design space exploration (DSE). At system level, system description becomes more decoupled from implementation details (basically functionality), although some implementation information has to be taken into account in order to obtain enough accuracy. For example, system-level DSE must consider the RTOS as part of the platform [21]. During system-level estimation of execution times of concurrent, heterogeneous embedded systems it is necessary to preserve the same model(s) of computation (MoC) of the system specification and the way each MoC is going to be implemented [22].

In this paper, a library capable of automatically providing timing estimation figures from a system-level description written in SystemC, is presented. No change of the code is needed, since the library is integrated in a full system-level, single-source design methodology [23]. Thus, the model of computation used in system specification and during system design is preserved. Performance analysis is performed taking into account the characteristics of the system platform used, thus, ensuring accuracy. As it is done at the system specification level, results are obtained fast and early in the design process. The information provided can be as complete as required. It can contain all the instantaneous estimated parameters for each process as well as the global figures. The designers can include capture points whenever they want in order to get additional, specific timing information.

Performance estimation is carried out dynamically at the same time as post-mapping, timed simulation, thus avoiding any additional design step. Timed simulation results can be used for verification of the timing behavior of the system as a function of the architectural mapping decisions taken. From the state of the art outlined above, there is a lack of system-level timed simulation tools for HW/SW systems.

The structure of the paper is the following. In this section the motivation and related work have been presented. In section 2, a system-level estimation methodology based on the segmentation of the specification code is presented. The specification methodology is outlined and referenced. In section 3, the segment parameter estimation method is explained. In section 4, the global analysis methodology is presented. In section 5 experimental results are shown. Conclusions are derived in section 6.

2. Process segmentation

This library follows the SystemC, system-level specification methodology presented in [22]. It is a general-purpose specification methodology able to support different MoCs. The main goal of the specification methodology is ensuring orthogonality between computation and communication. To achieve this, no event object is supported inside processes. Thus, the notify or wait primitives are not allowed except for the "timing" wait(sc_time) and processes lack a sensitivity list. Processes can only interact among themselves and with the environment through predefined channels.

The estimation methodology works on process segments instead on basic blocks. Segments have proven to be a very appropriate piece of code for performance analysis [24].

Following the specification methodology, a process can be represented by a graph. Arcs correspond to plain code segments without any waiting statement or channel access. Based on the SystemC simulation semantics, each segment is a closed element: it is executed completely without any interaction with the rest of the processes or the simulation kernel. Nodes correspond to the entry and exit statements of the process, the timing wait statements and the channel accesses. This means that the rest of the system (or the environment) only interacts with the process at its nodes. Its initial and final statements identify each segment. A segment may contain several paths of execution, but the same initial and final nodes. If not, segments are considered to be different. This means that two segments may have the same starting or ending points but not both, although they may share pieces of code. This is shown in

Figure 1:

```
N0 void process() {
     do {             //code of segment S0-1
       …              //common code to S0-1 and S4-1
N1     ch1.read();
       …              //common code to S1-2 and S1-3
       if(condition){ //common code to S1-2 and S1-3
         …            //code of segment S1-2
N2       ch2.write();
         … }          //code of segment S2-3
       …              //common code to S1-3 and S2-3
N3     wait(delay1);
       …              //code of segment S3-4
N4     ch2.read();
     } while (true);  //code of segment S4-1
   }
```

Figure 1. Process segmentation.

In this figure, N0 is the starting node of the cyclic process, N1, N2 and N4 are nodes corresponding to channel accesses while N3 is a waiting statement node. 'Si-j' represents the segment between the starting node 'Ni' and the ending node 'Nj'. The corresponding process graph would be that of Figure 2. The process graph represents the internal structure

of a process and not the process structure of the system.

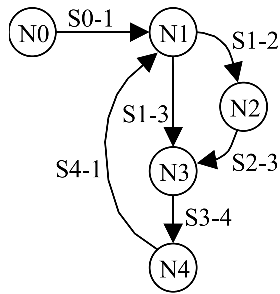


Figure 2. Process graph.

The library can dynamically recognize the processes but cannot directly recognize which segment is being executed. This is not necessary for obtaining both process and resource global estimations and event captures. Nevertheless, when required, the user can obtain an exact segment level report. To identify the segment, some marks are introduced into the code by a simple parser program. In the same way, a specific label is assigned to each channel access and during execution, segments are associated to the process to which they belong and the channel to which they access.

In order to improve accuracy, the performance characteristics of the system platform have to be taken into account. During architectural mapping, each process is assigned to a platform resource. The architectural mapping decisions are annotated in the system specification simply by using pre-processor directives. Three kinds of resources are distinguished: Parallel resources (i.e. HW resources), sequential resources (i.e. SW resources) and components of the environment. The latter include any Virtual Component (re)used. For each kind of resource, different segment estimation methods are used. For VCs and test-bench components no performance analysis is done.

3. Segment Estimation Methods

The performance estimation method is based on the redefinition capabilities of C++. Each C++ object is characterized for each of the resources of the target platform by its execution time. This affects any kind of platform resource such as microprocessors, DSPs, a standard cell library, a programmable fabric, etc. The method for obtaining these parameters is independent of the performance analysis library, which is prepared to introduce them easily.

All the C++ objects, which contribute to the execution time of the resource to which this piece of code has been assigned, are redefined in order to calculate their time contribution when they are executed.

C operators are overloaded. The library automatically replaces ordinary variable types by a new class. So, for example, the "int" type used in C language is replaced by a "generic_int" type with a "#define" statement. The other operators are overloaded in the same way. Figure 3 shows an example of the method applied. Suppose segment Si-j with the following code:

Library parameters

C object	# cycles
=	$t_{=} = 2$
+	$t_{+} = 1$
<	$t_{<} = 3$
[]	$t_{[]} = 5$
if	$t_{if} = 2.4$
function call	$t_{fc} = 18$

Segment code	Delay calculation	
	time	
`ch1.read();`		
`if(i<0)`	time += $t_{if} + t_{<}$	(= 5.4)
`i=c+d;`	time += $t_{=} + t_{+}$	(= 8.4)
`datai=array[i];`	time += $t_{=} + t_{[]}$	(= 15.4)
`datao=func(datai);`	time += $t_{=} + t_{fc}$	(= 35.4)
…	time += …	(= 75.8)
`ch2.read();`		

Figure 3. Delay calculation.

Each time a C++ object is executed the delay calculation function operates. In the example, the code inside function 'func' is not shown but its contribution is 40'4 cycles. Final delay is 75'8 cycles.

In this way, the proposed methodology is completely transparent for the user who has only to instantiate the performance library. Even the performance parameters for each object on each platform resource should not be his/her responsibility and should be provided by the platform vendor.

In the case of sequential (SW) resources, two statements cannot be executed in parallel. Independently of the SystemC simulation semantics, the processor will execute statements one after the other. Therefore, total time is obtained by adding the partial times required to execute each operation in the code of the segment being estimated.

In the case of parallel (HW) resources, timing estimation cannot be obtained without considering the design constraints imposed during synthesis. As a consequence, the library has to provide the two extreme points, which delimit all possible solutions of interest in terms of the product of time and area as shown in Figure 4.

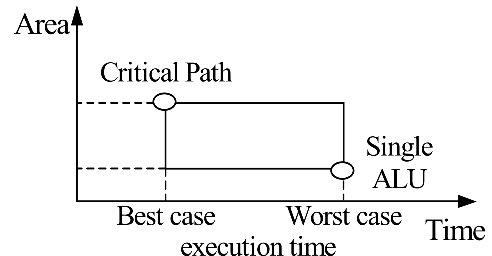


Figure 4. Implementation solutions.

The best execution time results in the fastest implementation of the functionality. To estimate this

value, the critical path of the sequence of operations in the segment is considered. For each operation, a multiple of the clock period is taken into account. This solution does not preclude a specific implementation for the operation (i.e. sequential, combinational in multi-cycle, etc.). To estimate the worst execution time, it is assumed that only one ALU is used and all the operations are executed sequentially. In this case, the execution time of the segment is obtained by the addition of the execution times for all the operations. However, the library time annotation method can only manage one value, not a range. The library calculates a weighted mean by using a constant value that determines the weight of each value. The following equation is used:

$$T = Tmin + (Tmax - Tmin)*k, \quad 0 \le k \le 1$$

Constant 'k' is determined by the user for each resource depending on the type of analysis he/she wants to perform or the relative priority that will be given to cost (k = 1) or performance (k = 0) during HW synthesis.

Once the estimated values for each segment have been obtained, they are used to carry out the performance analysis of the whole system.

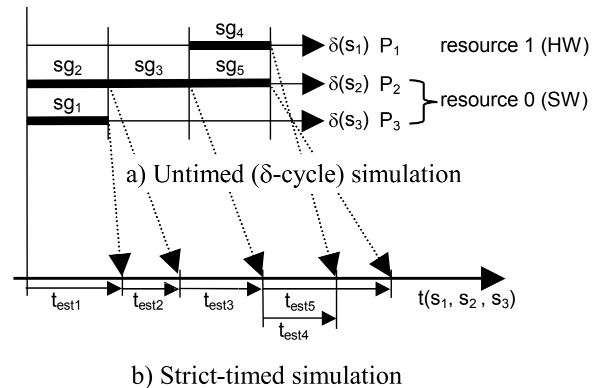
4. Global Analysis

In order to perform the simulation of the whole system while taking into account the estimated execution times for each segment, these segments are executed and the corresponding process slept for the segment estimated time. This method emulates the platform behavior.

In this way, the simulation is transferred from an untimed (delta cycle-based) execution to a strict-timed execution. Segments do not start following the order established by the delta cycle semantics of the SystemC system-level specification, but emulating their final behavior on the platform. Consequently, the models of computation used at the specification and the implementation level are preserved. By adding the performance estimation library, without modifying either the SystemC code or the SystemC simulation kernel, the simulation becomes strict-timed and all the events are assigned to a physical point in time (horizontal axis). This is shown in Figure 5 where signals s_1, s_2 and s_3 are generated by three (concurrent) SystemC processes P_1, P_2 and P_3 respectively. Part a) corresponds to the untimed simulation in δ-cycles. Part b) shows the strict-timed simulation with the delays for each segment estimated by the library taking into account the resources to which each process has been allocated.

Back-annotation of timing data is done automatically at the end of each segment. Channels and waiting statements have been extended to include the functions required to do this task.

The method used to decide the time when a process is resumed depends on the type of resource to which the process has been assigned. In the case of parallel processes, it is the maximum value between the ending time of the previous segment and the time of the event that has awakened the process. This is the case of signal s_1 in Figure 5. As can be seen, the execution time of segment sg_4 runs in parallel to the execution time of segment sg_5 as this segment corresponds to a process mapped to a different resource.



b) Strict-timed simulation

Figure 5. Delay annotation.

For sequential processes the decision is more complex. In SystemC each process has an independent thread running in parallel. However, over the platform, it is not possible to execute more than one process on a single microprocessor. The process needs to wait until the resource is empty. When a new segment is awakened, it reads the time of the event that has awakened it and the time when the resource is expected to be empty. If they are greater than the current simulation time, the process executes one wait to make all times equal. This process has to be repeated until the resource is empty because another process can take up the resource while it is waiting. Only when times are equal, the segment can run. This is shown in Figure 5 with signals s_2 and s_3. Although they are generated by different processes, as these processes have been assigned to the same sequential resource, the execution times of segments sg_1 and sg_2 are scheduled sequentially despite having been executed in the same δ-cycle.

The RTOS execution time is taken into account during process communication and synchronization. The RTOS will be executed each time a thread is stopped, that is, when a channel or a waiting statement is reached. Thus, the RTOS timing is estimated assigning an execution time to those channels and waiting statements executed by processes mapped to SW resources.

As mentioned above, annotation functions have been introduced into channel code. This means that to include new channels, they have to be adapted by inserting in a pair of functions provided by the library.

Library estimations are reported in two different ways. Total execution times for processes and resources are generated automatically. All instantaneous segment values of execution time parameters can be provided if required.

In addition to the fundamental parameters described above, the user can obtain additional information he/she may need to take the most appropriate design decisions. The user can insert capture points anywhere inside the code and a list of events corresponding to the concrete times when the capture points were executed is generated. The format of these lists is prepared for post-processing using mathematical tools (i.e. Matlab). Capture points can be conditional to a certain assertion. It is also possible to associate values of internal signals of the system to these time values. This possibility is very useful to verify timing constraints and to analyze response times, throughputs, input and output rates, etc.

5. Experimental Results

To assess the accuracy of the library developed, several small (sequential) benchmarks were used. For SW processes we have compared library results with simulations obtained from an ISS. This ISS was an OpenRISC architectural simulator modified to supply cycle accurate estimations. Library weights were obtained analyzing assembler code from several functions specifically developed for this purpose and taking into account microprocessor architectural characteristics.

The number of CPU cycles over the target platform estimated by the library and provided by the ISS is shown in the left part of Table 1.

	Target platform estimation time (μs)			Host simulation time (milliseconds)		
Benchmark	Library Estimation	ISS	Error (%)	Library exec. time	Overload w.r.t. SystemC	Gain w.r.t. ISS
FIR	28131	29288	4	325.2	51.6	142
Compress	167175	168744	0.9	1.61	27.28	236
Quick sort	5197	5202	0.1	6.77	67.7	237
Bubble	28947	28121	2.9	2.58	122.8	178
Fibonacci	730461	741590	1.5	38.98	10.9	278
Array	19142	18602	2.9	1.09	72.6	247

Table 1. SW estimation results for sequential benchmarks.

The right-hand part shows the execution times when including the performance library, the overload factor with respect to the original SystemC specification and the gain factor (time reduction) with respect to ISS execution.

For HW resources, the real execution times under resource-constrained and time-constrained scheduling have been obtained by using the Concentric behavioral synthesis tool from Synopsys. Two examples were selected, a FIR filter and the Euler algorithm.

Benchmarks	Real exec. time (ns)	Estimated exec. time (ns)	Error (%)
FIR (WC)	342	360	5.3
FIR (BC)	945	908	3.9
Euler (WC)	90	88	2.2
Euler (BC)	225	220	2.2

Table 2. HW estimation results.

In order to get results from a concurrent, sufficiently complex case study, an ETSI standard, the EN 301 245 vocoder for GSM applications, has been used. The sequential code has been divided in the 5 concurrent processes shown in Table 3:

	Target platform estimation time (ms)			Host simulation time (milliseconds)		
Benchmark	Library Estimation	ISS	Error (%)	Library exec. time	Overload w.r.t. SystemC	Gain over ISS
LSP estim.	371.4	365.2	1.7	95.75	36.8	187
LPC int.	322.1	330.3	2.5	90.25	34.7	182
ACB sear.	463.9	443.9	4.5	91.04	35.7	224
ICB sear.	662.7	642.9	3.1	150.3	30.2	235
Post Proc.	2	2.1	5	0.62	10	379

Table 3. SW estimation results for Vocoder.

The pre-processing function of the vocoder was mapped to HW. Results are shown in Table 4:

Benchmarks	Real exec. time (ns)	Estimated exec. time (ns)	Error (%)
Post. Proc. (WC)	4.875	4.475	8.2
Post. Proc. (BC)	1.975	1.900	3.8

Table 4. HW estimation results for Vocoder.

As can be seen, the error is maintained below 4.5% in SW and 8.2% in HW with a gain in simulation speed w.r.t. ISS more than 142 times. The simulation time is higher than the untimed simulation but it is kept lower than 73 times except in one small example.

6. Conclusions

In this paper a system-level performance analysis library for SystemC has been presented. The library can be introduced directly in the original code with minimal or no changes at all. It allows the designer to make an early estimation of the timing performance of the system. A novel, time estimation method exploiting the overloading capabilities of C++ has been used.

The library has an intrinsic value as an analysis tool, providing useful data for the co-design process based on SystemC. Timing constraints on a certain platform can be estimated and verified. The library automatically provides the fundamental performance parameters of the design required to take the most appropriate design decisions. In addition to these

parameters, the designer is able to extract from the SystemC code additional events to perform the specific timing analyses required, such as response times, throughputs, input and output times, etc. Thus the library can be used for timing constraint verification. As the SystemC simulation semantics is adapted by including the impact of the architectural mapping decisions, an additional value of the library comes from its application as a strict-timed (co-) simulation tool.

This method maintains the global behavior of the description although the execution order of processes can change as a result of the architectural mapping decisions. If results are different from the original system-level specification, it means that the description is not deterministic (potentially wrong). This represents an additional way to detect errors that may remain hidden in an ordinary simulation. Thus, the library becomes a powerful verification tool.

Based on the mean execution times and periods of the different processes, rate analysis and scheduling for soft, real-time embedded systems can be performed. The instantaneous execution times for the segments in the different processes can be used for performance verification and scheduling of hard, real-time systems. In both cases, the interaction between the different platform resources independently of their type is taken into account. The RTOS overload is evaluated.

SystemC has proven to be a powerful language for untimed, system-level simulation of complex systems, dynamic estimation of execution times during simulation and timed, system-level simulation taking into account the decisions taken during architectural mapping.

References

[1] "International Technology Roadmap for semiconductors: 2001 Edition, http://public.itrs.net.

[2] "The MEDEA+ Design Automation Roadmap", 2002, www.medea.org/webpublic/publ_relation_eda.htm.

[3] A. SanGiovanni-Vicentelli, G.Martín, "Platform-based design and software design methodology for embedded systems", IEEE Design & Test of Computers, November-December 2001.

[4] K. Richter, M. Jersak, R. Ernst, "A formal approach to MpSoC performance verification", Computer, April 2003.

[5] F. Balarin, L. Lavagno, P. Murthy, A. SanGiovanni-Vicentelli, "Scheduling for embedded real-time systems", IEEE Design & Test of Computers, January-March 1998.

[6] A. Mathur, A. Dasdan, R. Gupta, "Rate analysis for embedded systems", ACM Trans. on Design Automation of Electronic Systems, V.3, N.3, July 1998.

[7] S. Chakraborty, S. Künzli, L. Thiele, "A general framework for analyzing system properties in platform-based embedded system designs", Proc. of DATE, IEEE, 2003.

[8] M. Jersak, R. Ernst, "Enabling scheduling analysis of heterogeneous systems with multi-rate data dependencies and rate intervals", Proc. of DAC, IEEE, 2003.

[9] S. Yoo, G. Nicolescu, L. Gauthier, A. Jerraya, "Automatic generation of fast timed simulation models for operating systems in SoC design", Proc. of DATE, IEEE, 2002.

[10] J.Y. Lee, I. Park, "Timed Compiled-Code Simulation of Embedded Software for Performance Analysis of SoC Design", Proc. of DAC, IEEE, 2002.

[11] P. Puschner, C. Koza, "Calculating the maximum execution time of real-time programs", The Journal of Real-Time Systems, N. 1, 1989.

[12] S. Malik, M. Martonosi, Y.-T.S. Li, "Static Timing Analysis of Embedded Software", Proc. of DAC, IEEE, 1997.

[13] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. SanGiovanni-Vicentelli, E. Sentovich, K. Suzuki and B. Tabbara, "Hardware-Software Codesign of Embedded Systems: The POLIS Approach", Kluwer, 1997.

[14] A. Hergenhan, W. Rosenstiel, "Static Timing Analysis of Embedded Software on Advanced Processor Architectures", Proc. of DATE, IEEE, 2000.

[15] P. Giusto, G. Martin, E. Harcourt, "Reliable Estimation of Execution Time of Embedded Software", Proc. of DATE, IEEE, 2001.

[16] C. Brandolese, W. Fornaciari, F. Salice, D. Sciuto, "Source-level execution time estimation of C programs", Proc. of the Int. Symposium on HW/SW CoDesign, CoDes, 2001.

[17] P. Bjuréus, A. Jantsch, "Performance analysis with confidence intervals for embedded software processes", Proc. of the Int. Symposium on System Synthesis, ISSS, 2001.

[18] F. Mueller, "Timing analysis for instruction caches", The International Journal of Time-Critical Computing Systems, N.18, Kluwer, 2000.

[19] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao, "SpecC: Specification Language and Design Methodology", Kluwer, 2000.

[20] T. Grotker, S. Liao, G. Martín, S. Swan, "System Design with SystemC", Kluwer, 2002.

[21] A. Gerstlauer, H. Yu, D.D. Gajski, "RTOS Modeling for System Level Design", Proc. DATE, IEEE, 2003.

[22] F. Herrera, P. Sánchez, E. Villar, "Modeling of CSP, KPN and SR Systems with SystemC", Proc. of FDL, ECSI, 2003.

[23] F. Herrera, H. Posadas, P. Sánchez, E.Villar, "Systematic Embedded Software Generation from SystemC". Proc. of DATE, IEEE, 2003.

[24] F. Wolf, "Behavioral Intervals in Embedded Software", Kluwer, 2002.