System-Level Performance Estimation Strategy for Sw and Hw

A. Allara (1), C. Brandolese (2), W. Fornaciari (2,3), F. Salice (2,3), D. Sciuto (3)
(1) ITALTEL, Central Research Labs, CLTE, 20019 Castelletto di Settimo m.se (MI), Italy.
(2) CEFRIEL, via Fucini, 2 - 20133 Milano, Italy. salice@cefriel.it
(3) Politecnico di Milano, P.zza L. Da Vinci, 32 - 20133 Milano, Italy. {fornacia, sciuto}@elet.polimi.it

Abstract

The design of an embedded system is a process where the tuning of the architecture should take into account both the functionality and the timing performance while considering the heterogeneity of the hw and sw components. The goal of this paper is to present the new model developed during the SEED Esprit project, to estimate the software and hardware characteristics for cosimulation and profiling within the TOSCA codesign framework. The impact on the design space exploration of such an high-level cosimulation strategy has been tested by considering as a benchmark the reengineering of an industrial device.

1. Introduction

The TOSCA project [1] [2] aims at defining a complete methodology and the related CAD environment covering different aspects of embedded system codesign [7] including uncommitted system-level specification, design space exploration and cosynthesis.

A key point in such an activity is the possibility to take most of the decisions at the architectural level during the earlier stages of the design, in order to avoid as much as possible design loops including time-consuming synthesis activities. In fact, *time to market* pressure faces the designers with the necessity to adopt hw/sw partitioning or to reshape the system architecture in a short time to conform the project with the application requirements. A top level cosimulation is the ideal platform where the designer validates the system functionality and tradeoff alternatives. To speedup the the design process, TOSCA improves the simulation performance with acceptable detriment of accuracy by the use of a built-in cosimulator, based on estimations of the hw and sw execution times, starting from an internal OccamII model. The simulation engine is event driven and considers the intrinsic execution times of both hw and sw components according to an *a priori* analysis whose results are back-annotated onto the system-level description.

Under the designer's point of view, this last activity is the cornerstone to take synthesis decisions and normally it implies a high computational overhead: each sw and hw process need to be compiled, producing object code and synthesized hw respectively, before to evaluate and back-annotate the results on the system description.

Moreover, these evaluation steps have to be carried out each time the technology is modified (e.g., hw/sw partition, instruction set, technological libraries, ...) making onerous the design space exploration. To overcome these drawbacks it is valuable to estimate the performances by operating as much as possible at system level.

In literature some proposals are emerging to cope with the problem of estimating hw/sw performance. In [3], each process *i*, whose specification is captured in LOTOS, is characterized in time by performing a preliminary transformation in PCG (Process Communication Graph) followed by an appropriate expansion in blocks of data-dependent and/or deterministic operations; then, each new block is described through a CDFG. Successively, in the hw case, an ASAP algorithm is applied twice (non resource constrained and a single component of each type) on a pre-optimized graph providing the timing estimation (the target technology is an input). In the sw case, once the assembly code is produced, the response time of a CDFG can be estimated from a technology file (a description of the instruction set including the execution time for each instruction type).

In CoWare [4], system simulation is implemented by software: one UNIX process runs the instruction set simulator (armdbx) for the processes assigned to the sw partition, one UNIX process runs the VHDL debugger (vhdldbx) for the processes assigned to the hw partition while communication is implemented over UNIX IPC. After the assignment, all process instances assigned to the same simulation-processor are merged. As a consequence of this model, the simulation allows to verify only the functional behavior of the system. Since no timing estimation is performed at higher abstraction levels, no consideration about real-time behavior can be extracted.

In POLIS [5], each element of a network of CFSMs (Co-design Finite State Machines) describes a component of the system without specifying if it will belong to the hardware or software partition. Hw and sw modules are characterized by different delays even if each CFSM transition performs the same computation. System level hw-sw Co-simulation allows the designer to evaluate design choices that, at this level, include hw-sw partitioning, CPU selection, ... The timed co-simulation is based on software synthesis and performance estimation techniques. In the software synthesis each CFSM to be implemented via sw is mapped into a software structure (a procedure for each CFSM together with a simple Real-time Operating System) by means of a two-step process: first of all, each CFSM_sw is implemented and optimized in a high-level processor-independent representation, similar to a CDFG, then the CDFG is translated into C code. Successively, such a code is compiled and optimized in a specific instruction set. A timing estimator analyzes the program and reports code size and speed characteristics. The algorithm [6] uses a parametric formula, with parameters obtained from benchmark programs, to compute the delay of each node in the CDFG for different micro-controller architectures. The estimator allows the user to obtain accurate predictions of program execution times for any characterized target processor. This timing characterization is calculated by compiling the CDFG representation in C and by executing the resulting code on a host workstation where an instruction level simulator is running.

In summary, the timing estimation problem (when contemplated) is afforded either by translating the description at system-level into a finer grain, where each component of the system is accurately detailed and results are back-annotated, or by associating to the system level components some coarse grain information.

The former approach is characterized by a good accuracy in the estimations. Unfortunately, it is time-consuming and suffers of some drawbacks such as a strong sensitivity to the considered developing environment (e.g., the compiler) and architectural parameters, requires cooperation among different analysis tools and it is computed off-line with respect to the *what-if* analysis loop for hw/sw partitioning.

The latter approach, working at higher level, is fairly independent of the technology and fast, frequently in detriment of accuracy. It does not take significantly into account the differences among target architectures. Usually, such a class of estimation methodologies are computed on-line, within the inner loop of the hw vs sw analysis.

Our approach is a *meeting in the middle*, where the estimation is performed on-line at system-level, based on information coming from a performance model considering low level characteristics of code execution. The adopted timing estimation methodology consists of evaluating, run-time, the timing behavior of each process involved in the system level simulation. In such a way, is easier to take into account the unpredictable data dependent conditions such as branches, alternatives, … . The accuracy is pretty good, sufficient to take most of the decisions driving the hw/sw partitioning and behavioral analysis of the system. Furthermore, it is flexible since it considers the variability of the Instruction Sets (IS) and compiling environment and it is parametric in the clock period.

Due to space limitation only an overview of the theoretical model of hw and sw timing models will be discussed in this paper; more details on the general top level cosimulation strategy can be gathered from [8] [9] [1].

The paper is organized as follows. Sections two and three describe the models for the computation of the execution times of the Occam II system specification, for the sw and hw domains, respectively. Due to space limitation, details are reported only for some OccamII statements. Section four draws some conclusions and in particular discusses the potential benefit from the user point of view, obtained through the design space exploration activity performed during the SEED project.

2. The Software Estimation Model

In the TOSCA environment, a priori timing characterization of each OccamII sw-bound process γ is performed by estimating the process execution time as CPI_γ * Tck where Tck is the clock of the target processor and CPI_γ is represented by the following relation:

$$CPI_\gamma = CPI_{\gamma\min}{}^{\alpha} * CPI_{\gamma\max}{}^{(1-\alpha)} \qquad \text{with } \alpha\in[0,1]$$

Given the characterization of the processor and compiler, it is possible to compute, for each process γ, the value of $CPI_{\gamma\min}$ and $CPI_{\gamma\max}$. The best case, $CPI_{\gamma\min}$, is associated with a processor with unbound number of registers on which variables are pre-loaded; in such a case, the access to variables values requires no load/store steps. In the worst case, $CPI_{\gamma\min}$, the architecture is merely composed by 2 registers, thus requiring an intensive use of load/store operations. Note that the execution time of each process γ on a given executor falls in the range $[CPI_{\gamma\min}, CPI_{\gamma\max}]$. Since such a range depends not only on the intrinsic process but also on the CPI_{avg} of the target processor (for example, the MC68000 needs 4 CPI to move a word from memory to a data register), it is representative of a group of Compatible Instruction Sets (CIS). To make the estimation function as general as possible, CIS has been obtained by intersecting the relevant instructions sets and then by calculating the upper and lower bounds, that is:

$$\text{U-CIS} = \{CPI_{ave} \mid MAX(\cap_i \text{ IS of processor i})\}$$
$$\text{L-CIS} = \{CPI_{ave} \mid MIN(\cap_i \text{ IS of processor i})\}$$

The parameter α, introduced in the CPI_{avg} computation, allows to model different configurations of processor and compiler. For example, by increasing the data register count and by improving the compiler efficiency, CPI_{avg} tends towards $CPI_{\gamma\min}$. Furthermore, α depends on the executed code. By fixing processor and compiler, α can be estimated by compiling different code segments, evaluating α_i for each of them, and, finally, by computing the average, that is:

$$\alpha = \text{avg}\{\alpha_i | \min \sum_{\gamma \in code} \lceil (CPI_{real} - CPI_{\gamma min}{}^{\alpha i} * CPI_{\gamma max}{}^{(1-\alpha i)}) \rceil \; \forall \; code \in system\}$$

Such a value represents the characterization of the processor-compiler pair that must be considered by the software execution time estimator.

The $CPI_{\gamma min}$ and $CPI_{\gamma max}$ of a single process are efficiently computed by considering the same characterization of the primitives composing it. For instance, the OccamII process

StateBitCondition := (bitval=1)AND(contONE<5)

is composed by the *assignment* primitives and a *logical expression* composed of two comparison operators.

Let us now give the flavor of the methodology, by evaluating $CPI_{\gamma min}$ and $CPI_{\gamma max}$ for some of the OccamII primitives. More details concerning the rest of OccamII statements used in TOSCA, can be found in [9].

2.1 Assignments

The assignment process, :=, modifies the variable content. The following table reports CPI_{min} and CPI_{max} for the three types of possible assignments:

1. *Var := Var;* in the best case a value stored in a register is transferred to another one while, in the worst case, a value stored in memory is copied to another location.
2. *Var := ConstantValue;* it represents the case where a pre-load constant is copied in a register (best case) or a number is moved to a specified location (worst case).
3. *Var := Exp;* it identifies the case where the expression result is loaded in a register or in a memory location. The best case CPI_{avg} depends on the considered CIS: for two or three operand instructions, the expression evaluation consists of *compute and save* operations (the worst case needs to explicitly save the result), for instructions with one operand both worst case and best case need to save the computational result.

#Op		CPI_{avg}s		
		Var := Var	*Var := ConstantValue*	*Var := Exp*
3 or 2	CPI_{min}	CPI_{moveRR}	CPI_{moveRR}	0
1	CPI_{min}	CPI_{moveRR}	CPI_{moveRR}	CPI_{moveRR}
	CPI_{max}	$CPI_{load} + CPI_{store}$	$CPI_{moveImmediateMem}$	CPI_{store}

Table 1. CPIavg for the OccamII assignments.

2.2 Algebraic Expressions

An algebraic expression is a combination of symbols, algebraic operators, constants, and parentheses; expressions follow the conventional rules of algebra. An algebraic expression can be easily represented by a DAG where each internal node (V_I) and each operand node (V_O) represent an algebraic operator and an operand, respectively. In the considered model, the set of internal nodes, V_I, has been decomposed into three subsets: V_{I0}, V_{I1} and,V_{I2}. The numerical index indicates the number of operands connected to the node; for example, by indicating with |V| the cardinality of V, in figure 1b we have $|V_{I0}|$ =2, $|V_{I1}|$=2 and, $|V_{I2}|$=3.

In the *best case* CPI_{min} depends on the instruction architecture. If a three operands instruction architecture is considered, since the target architecture has as many registers as necessary, the CPI_{min} is only related to both the number of algebraic operators and the overhead caused by arrays (the address register has to be computed).

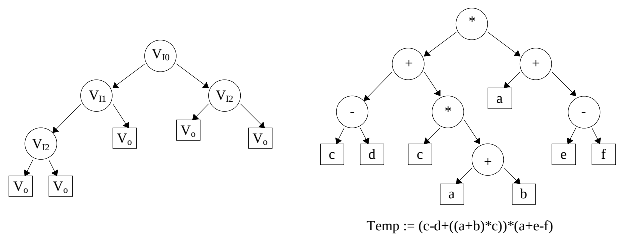

Figure 1. a) the considered model and b) an example of algebraic expression graph.

If the instruction set contemplates algebraic instructions with less than three operands, further overhead has to be considered. In the case of two operand instructions some variables have to be moved in temporary registers to save their value. In general, $|V_{I2}|$ variables have to be moved from their registers to temporary locations.

In the case of algebraic instructions with one operand, both partial results and some variables have to be moved, since the instruction itself implicitly specifies one register to store partial computations. In general, $|V_{I2}|$ variables and $|V_{I2}|$-1 partial results have to be moved.

In the *worst case*, because of the target architecture is constituted by only two data registers, the worst case CPI_{avg} does not depend on the considered CIS. Let us consider the algebraic expression $(b * c) + (d * e)$. The partial result corresponding to one partial computation (b*c or d*e) has to be stored to make data registers available for the other product; then, the stored value has to be re-loaded to compute the final result. In general $|V_{I0}|$ results are involved in load-store operations. Table 2 summarizes the possible cases.

2.3 Logical Expressions

A logical expression is a combination of symbols, logical operations (AND, OR and NOT), comparison operators (=, <>, <, >, >=, <=), constant, and parentheses; logical expressions produce boolean values. A logical expression can be represented by an OBDD (Ordered Binary Decision Diagram), representing a set of binary-valued decisions, culminating in an overall decision that can be TRUE or FALSE. Each Boolean variable in the OBDD corresponds to a set of nodes, either a test applied to two variables of the same type or a symbol (variable or constant whose value can be TRUE or FALSE); the nodes are ordered so that each level corresponds to a single variable. Hence, a logical expression can be seen as a network of local functions where the set of edges describes the logical operations while the set of nodes corresponds to the local functions

# Oper.		CPI_{avg}
3	CPI_{min}	$\sum^{\lvert V\rvert} CPI_{operator\_i}+OverHeadArray_{min}$
2	CPI_{min}	$\sum^{\lvert V\rvert} CPI_{operator\_i}+\lvert V_{i2}\rvert*CPI_{moveRR}+OverHeadArray_{min}$
1	CPI_{min}	$\sum^{\lvert V\rvert} CPI_{operator\_i}+(2\lvert V_{i2}\rvert-1)*CPI_{moveRR}+OverHeadArray_{min}$
	CPI_{max}	$\sum^{\lvert V\rvert} CPI_{operator\_i}+\sum^{op} CPI_{load}+\lvert V_{i0}\rvert*(CPI_{load}+CPI_{store})+OverHeadArray_{max}$ where $op = (2*\lvert V_{I2}\rvert+\lvert V_{I1}\rvert)$

Table 2. CPI_{avg} for the algebraic expressions.

It has to be underlined that the performance is related to the graph shortest path that, in turn, depends on the local functions ordering. To overcome the related computational effort in addressing all possible cases, since our goal is fast performance evaluation, two different approaches have been proposed. The first is based on two assumptions:

1. The shortest path is equal to 1 while the longest path is equal to the number of local functions. An analysis considering a set of examples has justified this first assumption: a typical logical expression is composed of a restricted number of local functions (less then 4) and, usually, the shortest path evaluates 1.
2. To preserve the functionality of a given expression, shortcuts have to be avoided; thus, all the local functions are pre-computed. This assumption allows a modular computation.

Therefore, if the number of symbols and constants involved is *op*, CPI_{min} and CPI_{max} are determined as:

	CPIs
CPI_{min}	$CPI_{cmp} + CPI_{min\_branch}$
CPI_{max}	$\lvert Longest\_path\rvert*(CPI_{cmp} + CPI_{max\_branch}) + \sum^{op} CPI_{load}$

Since hard real-time applications are considered, the second approach takes into account the worst case that is:

	CPIs
CPI_{min}	$\lvert Longest\_path\rvert*(CPI_{cmp} + CPI_{min\_branch})$
CPI_{max}	$\lvert Longest\_path\rvert*(CPI_{cmp} + CPI_{max\_branch}) + \sum^{op} CPI_{load}$

2.4 Tuning of the sw estimation model

Several tests have been performed during the SEED project to validate the sw estimation strategy. The first column of the following table reports a fragment of an OccamII specification computing a CRC code (of the ILC16 benchmark), after the necessary serialization of operations performed to fit a sw implementation (i.e. a single processor). The other columns report the values of CPI_{avg} computed according to the presented model. A value of α=0.7 has been chosen to minimize the cumulative error, computed on the set of processes composing the entire ILC16 system. The target microprocessor is the Motorola 68000. Figure 2 shows that, for the OccamII statements of table 3, the estimation sw model has a good correspondence with actual data.

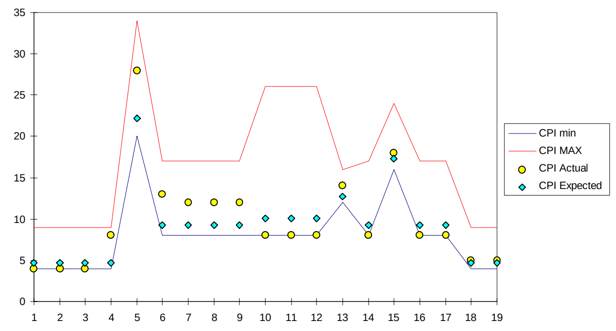

Figure 2. Comparison between actual and estimated CPI_{avg} for the OccamII code of table 3.

3. The Hardware Estimation Model

Similarly to the software case, the variability of hw resources and implementation strategies have been taken into account by modeling the execution time of hw-bound OccamII processes, through the following relation, where Tck_{HW} depends on the target technology:

$$CPI\_eq_{\gamma} = CPI\_eq_{\gamma min}{}^{\beta} * CPI\_eq_{\gamma max}{}^{(1-\beta)} \qquad \text{with } \beta \in [0,1]$$

Again, $CPI\_eq_{\gamma min}$ and $CPI\_eq_{\gamma max}$ represent the range containing the execution time of each process γ. The best case, $CPI\_eq_{\gamma min}$, represents an ASAP scheduling with no bound on functional resources. For the sake of completeness, note that such a value may slightly differ from the global minimum, since it does not takes into account the effect of possible inter-process optimizations. The worst case, $CPI\_eq_{\gamma min}$, corresponds to the presence of a single functional resource per type, i.e. a purely serial computation of the process. The parameter β depends on both the number of resources and the scheduling policy. The proper value has to be determined by benchmarking the hw compiling environment, through the analysis of different code segments, each producing a local β_i. Hence, the final value β used to represent a given implementation technology/architecture and scheduling policy, is:

$$\beta = \text{avg}\{\beta_i \mid \min \sum_{\beta \in \text{code}} \left(CPI_{real} - CPI_{\gamma min}{}^{\beta i} * CPI_{\gamma max}{}^{(1-\beta i)} \right) \forall \text{ code} \in \text{system}\}$$

OCCAM	CPI_{min}	CPI_{max}	CPI_{act}
MyData := Data	4	9	4
MyCRC_LOW := CRCLow	4	9	4
MyCRC_HI := CRCHi	4	9	4
i := 0	4	9	8
While (i < lenData)	20	34	28
i := i+1	8	17	13
tempData := BIT(MyData BITAND bytemask)	8	17	12
tempCRC_LOW := BIT(MyCRC_LOW BITAND bytemask)	8	17	12
tempCRC_HI := BIT(MyCRC_HI BITAND bytemask)	8	17	12
MyData := MyData >> 1	8	26	8
MyCRC_HI := MyCRC_HI >> 1	8	26	8
MyCRC_LOW := MyCRC_LOW >> 1	8	26	8
IF (tempCRC_HI)	12	16	14
MyCRC_LOW := MyCRC_LOW + remainder	8	17	8
IF (tempData >< tempCRC_LOW)	16	24	18
MyCRC_LOW := MyCRC_LOW >< generator_LSB	8	17	8
MyCRC_HI := MyCRC_HI >< generator_MSB	8	17	8
CRCLow := MyCRC_LOW	4	9	5
CRCHi := MyCRC_HI	4	9	5

Table 3. Value of CPI_{avg} for a serialized OccamII program computing a CRC code.

3.1 Equivalent CPI

In order to simplify the comparison among hw and sw solutions, the concept of *equivalent* CPI (CPI_eq) has been introduced. The execution time of each operation is mainly a function of the architecture (e.g., an adder can be implemented as ripple carry, carry look-ahead, ...) and the number of bits *n* of the involved operators, so that the execution time to carry out each operation is $T_{op}=f(\Delta, n, \textit{architecture})$, where Δ is the elementary unit of delay. By assuming that the clock period of the hw partition is $Tck_{hw}=K*\Delta$, the corresponding number of clock cycles is:

$$Nck= T_{op}/Tck_{hw} = \lceil f(\Delta, n, \text{architecture}) / K*\Delta \rceil$$

By calling $\Phi=Tck_{hw}/Tck$, where Tck is the clock cycle of the target processor, the following expression of the execution time can be obtained:

$$\begin{aligned} Execution\_Time_{op} &= (N_{ck}*Tck_{hw}/Tck)*Tck = \\ &= (\lceil (f(\Delta, n;m, \text{architecture})/K*\Delta) \rceil *\Phi)*Tck \end{aligned}$$

By comparing the above expression with the one determined for the software, it is straightforward to derive the following equivalence:

$$CPI\_eq = (\lceil (f(\Delta, n, \text{architecture})/K*\Delta)\rceil *\Phi) \equiv CPI$$

CPI_eq_{min} and CPI_eq_{max} of each process γ, representing the best and worst case of scheduling, are obtained by a modular composition of a set of primitives, each being an expression depending on a set of parameters.

The set of primitives includes the whole OccamII language, For space reasons, the paper reports only a subset of such analysis. More details can be found in [9].

3.2 Hardware assignments

As stated before, an assignment process, :=, modifies a variable. Three different types can be identified:

1. *Var := Var* : a value stored in a register is transferred in a different one;
2. *Var := ConstantValue* : a constant value is copied in a register;
3. *Var := Exp* : the expression result is loaded in a register. Since *expression computation* and *result storing* can be occur simultaneously, the CPI_eq corresponding to this process is completely dominated by the expression computation.

CPI_eq	
Var:=Var	1
Var := ConstantValue	1
Var := Exp	0

3.3 Hardware Algebraic Expressions

In the best case, execution time corresponds to a scheduling without resources limitation, i.e. the intrinsic process parallelism is maximized. In such a case, a good approximation of best performance can be achieved by considering all the operators of the same type and by considering the related maximum CPI.

The worst case refers to an architecture where the computation is fully serial. By considering the DAG representation introduced before and calling $|V_i|$ the cardinality of the subset of V constituted by an operator of type *i*, we obtain:

	CPI_eq
CPI_eq_{min}	$\max\{CPI\_eq_i \mid \lceil \log_2 (\lvert V_i\rvert+1)\rceil * CPI\_eq_{Vi}\}$
CPI_eq_{max}	$\sum_{\forall i} \sum^{\lvert Vi\rvert} CPI\_eq_{Vi}$

3.4 Hardware Algebraic Operators

In order to produce a characterization as close as possible to the user needs, the most representative architectures of algebraic operators have been analyzed [10]. The following table contains the models adopted to evaluate the performance of their implementations.

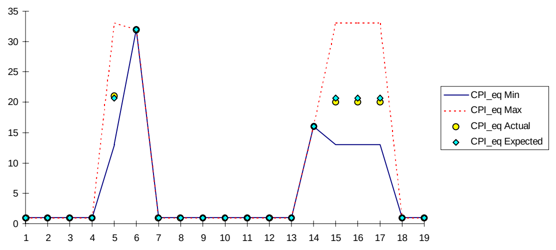

Figure 3. CPIeq for the hardware implementation of the OccamII specification of figure 2.

Op	Architecture	f(Δ, n, architecture)
+	Ripple carry	2*n*Δ

+	CCLA(n/p*p)
	p = BCLA dimension
		if (n=1) 11*Δ *else*
		$(10 + \text{logp}(1 + n * (\frac{n}{4} - 1))(\frac{n}{4} + 1) * 11 * \Delta$

*	Baugh Wooley	*if (n=1)* 2*Δ *else* 4*n*Δ
*	Bisection	*if (n=1)* 2*Δ *else* 4*n*Δ
/	Restoring (Dean)	$(3*n^2+1) * \Delta$
/	Non Restoring (Guild)	$3*(n+1)^2 * \Delta$
/	Non Restoring with 2-level CLA and Carry-save (Cappa - Hamacher)	(11*n+12) * Δ
/	Non restoring with 1-level CLA and Carry save (Cappa - Hamacher)	(9*n+10) * Δ

Table 4. Timing performance of the main Hw operator implementations.

The accordance between the estimations and the actual data for an hardware implementation is shown in figure 3, reporting CPIeq against the OccamII lines of the same example of figure 2.

4. Concluding remarks

The paper presented an overview of the hw/sw estimation model currently used by the TOSCA top-level cosimulator. This model has been validated and tuned by modeling both small benchmarks and the ILC16 component, commercialized by Italtel, used as test vehicle for the SEED Esprit project [1].

Accuracy of high-level cosimulation has been significantly improved while ensuring remarkable performance during cosimulation. By using a SPARCstation 20 running at 85MHz, we observed an average *simulation ratio* around 16-18. This means that, to simulate a system characterized by 24K events/s, the simulator is able to process 1.5K events per CPU second. The current implementation delivers a throughput already good enough to enable its effective use for developing real-size designs. In particular, such a type of analysis seems to be two orders of magnitude faster than the low-level strategy presented in [2], based on the use of VHDL models for both the hw and the sw.

The ILC16 component is a data-link controller for sixteen asynchronous/synchronous data streams based on the HDLC protocol. The original design allocates the management of the channels to a RISC CPU core cell, while the HDLC protocol processing is hw-bound; both sections are embedded in the same chip. The system has been reverse engineered by using the TOSCA environment and validated through the high-level simulator, the specification is composed of more than 4K OccamII lines of code.

The high-level TOSCA co-simulator has been the basis for design space exploration, the various components of the ILC16 system have been initially allocated to either hw or sw domains and then simulated to verify the functionality and the fulfilment of the timing constraints. Different scenarios have been considered before committing to the final hw/sw implementation, ranging from the fully hw to fully sw solution. Apart from the above considerations on the functional/timing properties, the system specifications have been modified through a set of transformations according to the computation of some metrics evaluating the quality of the system, but this aspect is out of the scope of this paper. Different alternatives in terms of microprocessor clock frequency have been also taken into account.

The total manpower (excluding the synthesis stage) has been allocated on the different design stages as presented in Table 5, depending on the level of confidence of the designer with the co-design environment.

Activity	weeks	new designer	trained designer
Occam Language (tool learning)	4	13%	
Exploration Manager (tool learning)	1	3%	
Design Specification	8	26%	31%
Functional Debug	16	52%	62%
Design Space Exploration	2	6%	8%
Total weeks/m	31		

Table 5. Breakdown of the manpower for the development of the ILC16 device.

These results are important to get a feedback on the effectiveness of the proposed methodology for design entry and design space exploration.

Currently, most of the effort is devoted to define high-level estimation strategies for the hw and sw power consumption, and in the improving of the existing algorithms for hw/sw partitioning.

5. References

[1] SEED ESPRIT-ESD project n.22133, www *site* http://www.cefriel.it/eda/projects/seed/mainmenu.htm.

[2] W.Fornaciari, F.Salice, D.Sciuto, *A two-level Cosimulation Environment*, IEEE Computer, pp. 109-111, June 1997,.

[3] C.Carraras et al., *A Co-Design Methodology Based On Formal Specification And High-Level Estimation*, Proc. of IEEE Codes/CASHE'96, Pittsburgh, Pennsylvania, 1996.

[4] I.Bolsen et al., *Hardware/Software Co-Design of Digital Telecommunication Systems*, Proc. of the IEEE, Vol. 85, No. 3, pp. 391- 418, March 1997.

[5] M.Chiodo et al., *Hardware-Software Codesign of Embedded Systems*, IEEE Micro, Vol. 14, No. 4, pp. 26-36, Aug. 1994.

[6] K.Suzuki, A. Sangiovanni-Vincentelli, *Efficient Software Performance Estimation Methods for Hardware/Software Codesign*, Proc. of DAC '96, Las Vegas, US, June 1996.

[7] G. De Micheli, M.G. Sami editors, *Hardware/Software Co-Design*,: NATO ASI Series, Series E: Applied Sciences - vol.310, Kluwer Academic Pub., The Netherlands, 1996.

[8] A.Allara, S.Filipponi, W.Fornaciari, F. Salice, D. Sciuto, *A Flexible Model for Evaluating the Behavior of Hardware/Software Systems* , Proc. of IEEE Codes/CASHE'97, Braunschweig, Germany, March 1997.

[9] W.Fornaciari, F.Salice, D.Sciuto, *Hw and Sw estimation strategies for the SEED project*, Tech. Report n. 97011, CEFRIEL, Milano, Italy, 1997.

[10] K.Hwang, *Computer Arithmetic, Principles, Architectures and Design*, John Wiley & Sons, 1979.