A Hardware/Software Co-design Flow and IP Library Based on Simulink$^{\text{TM}}$

L.M.Reyneri, F.Cucinotta, A.Serra
Dipartimento di Elettronica
Politecnico di Torino, Italy
email:reyneri@polito.it

L.Lavagno
DIEGM
Università di Udine, Italy
email:lavagno@uniud.it

Abstract

This paper describes a design flow for data-dominated embedded systems. We use The Mathworks' Simulink$^{\text{TM}}$ environment for functional specification and algorithmic analysis. We developed a library of Simulink blocks, each parameterized by design choices such as implementation (software, analog or digital hardware, . . .) and numerical accuracy (resolution, S/N ratio). Each block is equipped with empirical models for cost (code size, chip area) and performance (timing, energy), based on surface fitting from actual measurements. We also developed an analysis toolbox that quickly evaluates algorithm and parameter choices performed by the designer, and presents the results for fast feedback. The chosen block netlist is then ready for implementation, by using a customization of The Mathworks' Real Time Workshop$^{\text{TM}}$ to generate a VHDL netlist for FPGA implementation, as well as embedded software for DSP implementation.

1. Introduction

More and more application domains, from consumer electronics, to embedded control, are benefiting from the electronic revolution. Embedded systems are thus rapidly becoming a driving factor of the explosive growth of the electronic industry. On the other hand, more and more designers need to be able to exploit digital and analog, off-the-shelf and application-specific electronics in their daily work, and cope both with their traditional application domain of expertise, as well as with hard to master Electronic Design Automation tools.

Timing, power dissipation and reliability are some of the main performance criteria that must be satisfied within tight time-to-market and cost constraints. This requires a drastic change in the design methodology in order to cope with the exponential growth of complexity of digital and analog electronics that can fit on a single chip.

We advocate a design methodology that on the one hand allows the system designer to concentrate on a high-level functional specification using *familiar tools tailored for his specific application area*. On the other hand our methodology provides a rapid feedback on the effects of algorithmic choices and of different architectural solutions, on the cost and performance of the system being designed. We claim that this can be achieved with a flow that *separates functionality from communication and architecture*. In this way, performance and cost can be automatically estimated based on a user-defined *mapping* from functional blocks (such as gains, integrators, FIR filters, IDCTs, and extended FSMs reacting to user inputs) to architectural entities and communication resources (such as processors, DSPs, ASICs, FPGAs, buses and memories) [1].

The designer must be offered the opportunity to trade off between *software* (for flexibility) and *hardware* (for performance and low power), without having to worry about details such as the hardware/software communication protocol implementation at the bit and cycle level. This enables quick exploration of the results of architectural decisions such as the number of processors, the hardware/software partitioning and numeric accuracy.

In this paper we propose an implementation of that methodology that relies on:

1. the well-known and widely used functional specification framework Simulink$^{\text{TM}}$ [4] from The Mathworks, enhanced to compute cost and performance functions such as total byte size or energy,
2. an in-house IP library:
   (a) *parameterized* by implementation choice (software, hardware, analog, digital, . . .), bit width, architecture (serial, parallel, . . .), pipelining level, and so on,
   (b) equipped with a cost and performance model for each block, estimating (based on a second order curve-fitting process from real implementation data) chip area, byte size, clock cycles, energy per operation, etc..
   (c) including key blocks such as all the standard Simulink blocks (gains, integrators, derivatives, generic functions, zero-pole transfer functions, etc.) augmented with a few Simulink toolboxes (like control, neural networks, fuzzy systems, etc.).

We chose Simulink to implement our design flow because most engineers graduating today have been trained extensively to use it. It is also one of the most widely used specification and modeling environments in the area of embedded control and system modeling.

Alternatively, our design flow and libraries can also be used as an accelerator for Simulink simulations. In this case, all blocks can be mapped onto an appropriate hardware/software platform (often including an FPGA), which can then run simulations at a much higher speed than a general-purpose PC would do. Obviously this idea is feasible only when simulation time is significantly higher than compilation time.

The paper is organized as follows. We first outline our functional and performance simulation framework. We then discuss the supported technologies and architectures. Section 4 discusses how performance models are evaluated, while section 5 describes a simple application example to illustrate the performance of our design flow and libraries.

2. The Simulation and Performance Evaluation Framework

While defining a Simulink system, the designer selects functional blocks from a large library including low-level functions such as addition, multiplication and max/min, as well as high-level tasks such as FIR filters or arbitrary transfer functions, and IP libraries such as neural and fuzzy evaluation blocks. The library is built by using the standard Simulink model for each block, and adding to it a mask that contains both

1. the parameters to be set by the designer during algorithmic and implementation exploration (see section 3.1),
2. empirical models (derived as discussed in section 4) for the cost functions to be evaluated.

A *functional simulation* can then be carried out to validate the design, using the standard Simulink simulation engine which supports continuous time and sampled time simulations[1]. In this paper we focus on specifications that can be functionally represented wholly in the digital domain, and that can be implemented on a platform including a processor and programmable hardware (FPGAs), therefore we chose to use sampled time simulations, except for analog implementations.

Each algorithmic and architectural choice can thus be evaluated both in terms of functionality, using the standard tools provided by The Mathworks, and in terms of cost and performance, using our cost models for each block and for the entire system.

The *estimated* performance data are composed across multiple blocks and hierarchical entities, either by addition (for quantities such as energy or code size), or by critical cycle analysis (for system-level throughput), as indicated in section 4.1. Note that the dataflow process model [2] used by the Simulink sampled time simulator ensures scheduling-independent behavior and performance.

Hence the computational load (for software) and chip area (for hardware) imposed by one block are simply added to that of all other blocks mapped to the same resource (CPU, FPGA, ASIC, etc.). The results of the cost and performance analysis is shown to the designer by changing the text associated with a special block, which must be contained in every design to be analyzed using our methodology, and that executes the estimation computations.

The cost and performance models for each block are obtained:

- for hardware blocks, by curve fitting on a selected set of implementations for a representative combination of parameter values (as discussed in section 4),
- for software blocks, from worst case execution time or power estimation [3, 5].

In both cases, all the quantities of interest must be presented to the simulator as a *parameterized function* that can be evaluated by Simulink. The function can depend on all the design parameters that can customize the performance and cost of the block, such as, for example:

1. The implementation choices (software, hardware, digital, analog, bit-serial, parallel, . . .).
2. The bit parallelism or analog tolerance of the block (that is also used in functional simulation to determine the S/N ratio) and the data format (signed, unsigned, . . .).
3. The number of pipeline levels (usually 1, although complex blocks can have more levels; it can also be set to 0 to make blocks purely combinational).
4. The clock frequency and the kind of processor.

Once the implementation decisions have been made, the Simulink netlist with the architectural parameters is processed and split in one part for each implementation. The software part is then passed to The Mathworks' Real Time Workshop^{TM} [4], while the hardware part is processed by a *module generator* that

- selects and parameterizes the appropriate synthesizable VHDL block (taken from one of our libraries), and
- generates a VHDL netlist, including automatically generated interfaces between architectural components (e.g. between hardware and software blocks and between parallel and bit-serial blocks).

The design methodology in itself is not new, and has been the subject of a number of papers before. Moreover, there have been recent announcements (but not enough actual details) of commercial offerings of hardware and software implementations of Simulink libraries [6]. Our approach is different, however, because it provides the system-level designer who has limited experience in electronic design with *quick feedback* on the consequences of his choices. Alternatives based on detailed design and evaluation do not match well the fast design turnaround times of modern embedded systems, and require a *hardware and software design expertise* that may not be widely available.

The improved flexibility and ease of use is paid by a lesser accurate estimation of performance. Yet a such faster design flow allows to analyze many more architectures, select a handful of candidates, which can then be analyzed more accurately with the traditional approach (namely, post-layout simulation for hardware and cycle-accurate simulations for software).

3. Technologies and Architectures

Each Simulink block in our library can be implemented in one of several possible technologies: *software*, that is a piece of code to be run on the embedded CPU core; *digital*, that is a piece of hardware that can be either mapped onto an (embedded) FPGA, or on a standard-cell based VLSI block; *analog*, that is a piece of hardware mapped on either an analog full-custom, or an analog standard-cell block; *external*, which merely emulates the part of the system external to the chip/board (for instance, the system to be controlled), and therefore need not be converted to neither hardware nor software. External blocks are inserted only for the purpose of overall system-level simulations.

Digital technologies can be designed in one of several *architectures*. A few examples are: synchronous parallel, synchronous bit-serial (leading MSB), synchronous bit-serial (trailing MSB), synchronous systolic, asynchronous parallel, adiabatic parallel, etc.

Different architectures can also be chosen for analog technologies, such as: single-ended voltage mode, differential voltage mode,

[1]The former is obviously more suited for simulations of the physical environment where the embedded system operates or to model analog components, while the latter is much more efficient for digital sampled hardware and software.

single-ended current mode, differential current mode, pulse width modulation, frequency modulation, etc.

Interfaces are handled by means of appropriate interface Simulink blocks (e.g. A/D, D/A converters, PWM and encoder drivers), which are inserted between hardware, software, and external blocks. This step is currently manual, but we are working towards its automation. For instance, an A/D converter block (that results in the instantiation of the appropriate hardware block and software driver) is inserted between an external and a digital block, or between an analog and a digital block, whereas a bus decoder is inserted between a software and a digital block.

An appropriate attribute (see section 3.1) is associated with each block to choose the desired technology for that block. To simplify the specification and design of complex blocks, attributes of hierarchical blocks are inherited along the hierarchy, unless otherwise indicated.

We are currently in the process of designing a family of implementations for each Simulink block (ideally we would need at least one implementation for each technology and for each architecture; currently we are planning to provide all implementation choices only for the most frequently used blocks). We are of course not planning to design a hardware implementation for blocks such as text and file I/O. Currently we implemented most Simulink blocks using the synchronous parallel digital architecture. A few blocks are also available in synchronous bit-serial architectures. Asynchronous and adiabatic architectures will be added in the near future. We also designed a few analog architectures for matrix-vector multiplications and neuro-fuzzy blocks [7], and in order to improve compilation efficiency for specific applications [8]. Software architectures need no library, as we use The Mathworks' Real Time Workshop to convert from Simulink to C code.

When invoked, an ad-hoc compiler reads from the libraries and generates, for each Simulink block, a description in a language which depends on the associated architecture. In particular: software blocks are described in ANSI-C, which can be compiled for most commercial μCs, DSPs and cores; digital hardware blocks are described in VHDL-93, which can be compiled by most silicon compilers for both ASICs and FPGAs; analog hardware blocks can be described in either AMS-VHDL or by means of the Cadence *Skill* language; external blocks need not be compiled, as they are not part of the system under design. Hybrid systems designed using more than one technology or architecture are automatically converted into one partial description for each technology used.

3.1. Design attributes and signal types

A new set of attributes has been added to the original ones of each Simulink block (e.g., gain for an amplifier, transfer function for a filter, number of points for an FFT). These define: *technology* and *architecture numeric accuracy* (data width, binary point position, truncation and overflow handling method), some performance trade-offs (processor and hardware clock speed, number of pipeline levels in hardware), type of signal (*scalar*, *vector* or *matrix*; vectors and matrices can be time-serial or parallel) and its encoding (*signed*, *unsigned*, or *sign+modulus*, plus some other useful encodings for low-power systems such as *redundant numbering systems*), and a few other characteristics of the block. Analog implementations are intrinsically continuous, although an appropriate attribute specifies the maximum allowed *S/N ratio* and *bandwidth.* Signals in software blocks can also be floating point (if supported by the chosen CPU).

The number of pipeline stages is usually either 0 (for purely combinational blocks, without flip-flops), or 1 (for blocks with one storage element at each output, for pipelining); it can sometimes be larger, in more complex blocks (such as neuro-fuzzy systems, or filters, or zero-pole transfer functions). The number of pipeline stages directly affects the trade-off between latency and max clock frequency, and indirectly also that between area and power consumption. Attributes are inserted by means of standard Simulink dialog boxes, as described in section 2.

3.2. Modularity of digital blocks

Most digital hardware blocks are internally composed of a few interacting blocks, both for simulation and for hardware implementation[2]:

1. the *protocol cell*, which handles all and only the protocol signals independent of block function; it interacts with the actual functional block by means of a few signals which enable the pipeline storage elements (if present) and reset accumulators and counters (when present). There is one protocol cell for each architecture;

2. one or more *data conversion cells* which convert numeric resolution, data format and handle truncation and overflow of input and output data (when they do not match each other). Also the function of this cell is independent of block functionality, and it is parameterized by the numeric accuracy attributes (see section 3.1) of input and output ports;

3. the *main functional cell*, which implements the actual block functionality (e.g. gain, integrator, filter, . . .). This cell can be in turn made of a few simpler interacting cells, when this simplifies design (design reuse).

The main functional cell does not have to handle protocol signals nor data conversions, thus easing the design of a new block.

The correctness of data transfers is guaranteed by the protocol cell, which removes that burden from the functional cell. In a similar way, data conversion cells take care of adapting resolution and data format among different cells.

In addition, the protocol cells are such that several functions (namely, all those which do not require having an internal $\frac{1}{z}$ delay) can process either scalar or time-serial vectors or time-serial matrices in a transparent manner. As a consequence, the very same block automatically adapts to the number of elements in the scalar/vector/matrix. Only some performance figure (namely, energy per cycle, see section 4) are affected by that number.

4. Simulation and Performance Models

Simulink blocks are simulated in time-discrete mode, namely once for each Simulink time step.

The data exchange paradigm underlying the proposed design flow for synchronous (or bit-serial or systolic) hardware and for software requires the availability of two independent clocks:

1. a *system clock*, that is (on the boards that we used) both the global clock for sequential blocks in hardware, and the CPU clock for software;

2. one *timing clock* to give appropriate *functional* timing to the blocks. This clock is the implementation counterpart of the discrete time simulation clock of Simulink (several timing clocks are used in multi-rate Simulink systems). The timing clock is generated either by an appropriate hardware *timing block*

[2]Implementation modularity is essential in order to keep the library manageable, and we noticed that the overhead is mostly eliminated by the logic synthesis step.

(added to the Simulink netlist as appropriate), or by some other external sub-system.

Asynchronous hardware blocks only require the availability of the timing clock, while analog blocks might require no clock (when used in continuous time mode).

Thanks to the chosen dataflow paradigm, each block is triggered every time a new data item (token) is present at each input, and each new data at the output of a block automatically triggers all the driven blocks. Blocks can be either combinational or sequential. In the former case, data are output within the same clock cycle, while in the latter case, they are sampled at the next rising edge of system clock.

Simulink blocks are chained; each chain originates either from a *signal source*, or from an *external interface*, or from sampling or storage blocks (e.g. *unit delay*, or *discrete integrator* in *forward Euler* mode), namely any cell which synchronizes with the timing clock.

Loops of blocks that do not include either a sampling, or a storage, or an external block, must be strictly avoided. This is because the dataflow paradigm would create a deadlock condition in presence of a loop, unless sampling or storage blocks break it. In practice, Simulink itself does not allow such a situation, and generates an error message before starting any simulation. This constraint makes the computation of some system performance figures (e.g., latency and throughput) much easier.

Currently we deal with the following performance figures. For hardware: *latency* (for synchronous implementations), which gives block delay in terms of clock cycles; *throughput*, which gives the fractional number of data processed per clock cycle; *max clock frequency*, which gives the max clock frequency of each block; *energy dissipation* per timing cycle; *area*, which gives system size, either in terms of Si surface (for ASICs) or number of cells (for FPGAs); and, for software: *code size*, which gives the size of code in bytes; *clock cycles*, which gives number of CPU clock cycles to compute one Simulink time-step.

Each block is associated with a *performance model* which provides the system with the performance figures of that block as a function of block attributes. Our design flow then combines the individual figures as described in section 4.1 and delivers to the user the overall performance figures, which can be used to evaluate to quality of the chosen architectures and to compare that against other implementation choices.

Currently the structure of performance models of each block reflects the common internal structure of the blocks themselves (see section 3.2). Partial models are generated for the protocol cell, the data conversion cells and the main functional unit. They are subsequently merged together to provide the complete block model.

Each partial model is a function of different parameters (for instance, the model of data conversion blocks is a function of numeric accuracy parameters, while the model of functional blocks is a function of number of pipeline levels and internal data width). The individual partial models are generated by 2^{nd} order polynomial fitting of empirical data obtained after synthesis of a number of different cases with different data widths, pipeline levels, etc. The procedure of model generation is partially automated, and must be executed once for each block in each supported technology.

4.1. Computing overall performance

The goal is to perform system simulation utilizing the computational power of Matlab$^{\text{TM}}$ while preserving, as much as possible, the look and feel of the simulation environment. Three groups of parameters are added to every standard block:

1. Software implementation parameters;
2. Hardware implementation parameters;
3. Models of performance;

In addition, a flag is associated with each block to indicate the type of implementation used. With these flag and parameters, the block contains all the information about its implementation and its local performance. This technique is used to extend any Simulink block into an *extended block* for performance analysis.

This extension is completely transparent to the Simulink simulation. The computation of performance is started by the *initialization block* which must always be placed inside the Simulink netlist by the designer. Once the block is inserted, it will also display the estimated system performance.

Before launching a simulation, the designer has to set the intended type of simulation inside the parameter mask of the initialization block. Two types of simulation are possible: either a *functional simulation* (standard Simulink simulation) or a *performance simulation*.

Performance estimation is executed by an algorithm written in Matlab, and executed when the Start command is selected. Matlab allows expressions that are executed when the block diagram or a block is encountered during netlist traversal. These expressions, called ”call-back” routines, are associated with block parameters. By using one of these parameters, a function, ‘InitFcn’, is associated to the Initialization Block to evaluate the performance parameters. By using this function the pre-simulation is split into two parts: first the whole Simulink netlist is loaded into the Matlab Workspace. The result of this first parsing step is a directed graph. For each block (graph node) the local parameters of performance are evaluated using their associated models, as discussed in Section 4.

In a second step, all the local data are merged to produce the overall performance figures. The following figures are computed for the hardware partition:

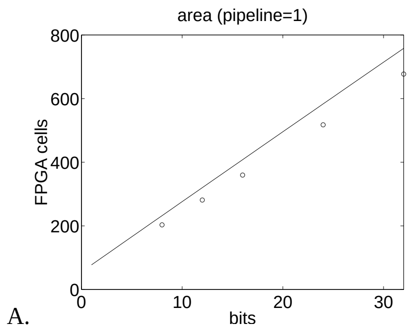
1. area: it is obtained by summing up all the individual estimated areas of each block (for ASIC implementation) or the estimated number of logic cells (for FPGA implementation);
2. energy consumption: it is also obtained by summing up all energy consumptions;
3. latency: it is obtained by finding the longest path, in terms of number of pipeline registers, between any input and output (including $\frac{1}{z}$ sampling blocks, that are split into an input and an output);
4. max clock frequency: it is obtained by taking the minimum among the maximum clock frequencies of each single block[3].

Similarly, the following figures are computed for the software partition:

1. code size: it is obtained by summing up the size of all software cells;
2. cycle time: it is obtained by summing up the cycle time of all software cells (based on the properties of the single-rate static scheduler used by Real Time Workshop);

The computed figures are displayed within the initialization block.

[3]Unfortunately, timing estimation at this level is too rough to allow one to estimate the clock frequency when there are blocks without pipeline registers. In this case a full synthesis run is required.



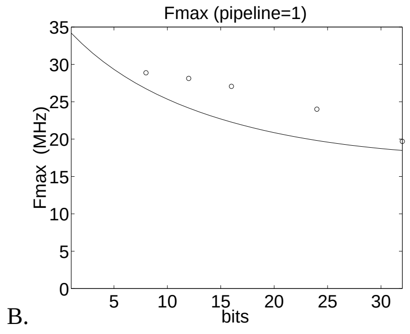


Figure 1. Plots of estimated (continuous line) and actual (circles) performance versus numerical accuracy.

Resolution /pipeline (bits/levels)	estimated logic cells ALTERA	estimated energy (pJ/cycle)	estimated latency (clock cy.)	estimated max freq. (MHz)	actual logic cells ALTERA	actual energy (pJ/cycle)	actual latency (clock cy.)	actual max freq. (MHz)
8/0	136	1,564	1	8.2	136	1,566	1	13.2
12/0	196	2,249	1	7.4	195	2,245	1	11.8
16/0	255	2,932	1	6.7	254	2,923	1	10.5
24/0	374	4,297	1	5.8	372	4,275	1	8.4
8/1	232	2,668	4	26.7	203	2,337	4	28.9
12/1	320	3,680	4	24.2	281	3,235	4	28.1
16/1	408	4,692	4	22.3	359	4,137	4	27.1
24/1	583	6,708	4	19.8	518	5,954	4	24.0
RMS error (%)	9.36	9.36	-	25.8				

Table 1. Estimated and actual performance.

5. An Application Example: Control of a Brushless Motor

One of the first real-world applications of our design flow and libraries has been the closed-loop control of a brushless motor. Although being a rather simple problem, it has several commercial applications, therefore it is a reasonable test bench.

Our task was to control a MAXON brushless motor (three-phases, 12 V, 40 W) with a 500 points per revolution incremental encoder. It was driven by a three-phase n-MOS bridge, which had to be controlled by a PID controller, for constant (programmable) speed.

Figure 2 shows the Simulink model used to model (blocks `Gain3` and `Zero-Pole`) and to control (all other blocks) the motor. The motor model was intentionally very simple, yet accurate enough for the given application.

Blocks `Integrator`, `Derivative`, `Gain`, `Gain1`, `Gain2`, `Sum` and `Saturation` implement a closed-loop PID control with programmable PID gains.

The system has been simulated in Simulink, with a continuous time simulation model (with the Runge-Kutta integration method), with a 2 kHz sampling frequency, while in the *hardware* implementation a time-discrete integrator and derivative methods have been used.

The motor model blocks have been flagged as *external_brushless* and *external_encoder* respectively, while the motor controller blocks (namely, the PID) have been flagged as *hardware synchronous parallel*. The `Step` block (namely, the reference speed) has been flagged as *software*, assuming that it will be implemented on a DSP.

The Simulink model was then compiled, and appropriate interfaces (see section 3) were inserted between *hardware* and *external* blocks (namely, the logic of the brushless motor driver). An encoder driver was inserted as an interface between *external* and *hardware* blocks.

The transient behavior of the system was both simulated and measured on the real system. Matching was good; the only minor difference was due to the approximate model of the motor.

5.1. Accuracy of Performance Estimation

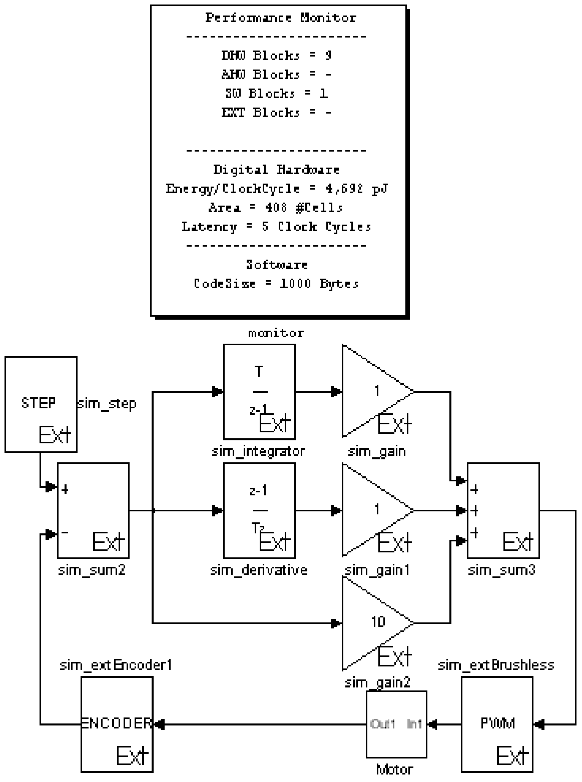
Table 1 shows the estimated and the actual performance of the proposed example, for different levels of numeric accuracy in the digital hardware portion (compiled on an ALTERA 10K40 FPGA with 5 V power supply). *Estimated performance* figures have been obtained as described in section 4.1, while *actual performance* figures have been obtained after complete compilation, synthesis, placement and routing. The last row gives the mean square error of the estimate, which is reasonably good when considering that the proposed performance models can be computed at least three orders of magnitude faster than the whole synthesis process (FPGA placement and routing is especially slow).

Thanks to the high speed of the proposed performance estimation, it is very easy to obtain graphs such as those shown in figure 1, which plot overall performance as a function of some tradeoff parameter (like numeric accuracy, in the proposed example).

Table 2 gives the performance of the proposed design flow and libraries for use as a Simulink accelerator, when running on a proprietary hardware/software platform (with a TMS320C30 DSP and an ALTERA 10K40 FPGA), developed by another group at our University [8]. The performance of the original Simulink model and of the corresponding hardware/software implementation generated by our system are compared. The “simulation speedup” column indicates how faster the accelerated hardware emulation is than the Simulink

Parameter	Simulink	HW	Improvement
Speed-up	0.5	1,000	2,000
Sample freq. (max)	1 kHz	2 MHz	2,000
Size (8 bit)	-	1.6% of ALTERA 10K40	-
Size (16 bit)	-	3.0% of ALTERA 10K40	-
Power (8 bit) @ 2 MHz	100 W	3.13 mW	590,000
Power (16 bit) @ 2 MHz	100 W	8.55 mW	195,000
Compil. time	N/A	150 s	-

Table 2. Performance of our design flow and libraries compared against Simulink running on a Pentium II 333MHz, when used as an accelerator of Simulink simulations.



```
Performance Monitor
----------------------
DHW Blocks = 9
AHW Blocks = -
SW Blocks = 1
EXT Blocks = -

----------------------
Digital Hardware
Energy/ClockCycle = 4,692 pJ
Area = 408 #Cells
Latency = 5 Clock Cycles
----------------------
Software
CodeSize = 1000 Bytes
```

Figure 2. Simulink diagram for modeling and control of a brushless motor with the performance monitor block.

simulation. Even in this simple case three orders of magnitude of performance improvement can be achieved. This technique can thus be very advantageous for long simulations, in which the long FPGA compilation times are compensated by the simulation speedup.

Our design flow and libraries have also been successfully used in other applications, like neuro-fuzzy implementations, as discussed in [8].

6. Conclusion

In this paper a new Simulink-based tool for hardware/software codesign and architecture selection of high performance, low-cost, data-dominated, embedded systems has been proposed. Different technologies and architectures can be chosen and compared, for each block of the design library, in order to achieve an optimal power consumption, area and speed trade-off.

We showed the effectiveness, performance, and flexibility of our tool and library, by using a simple example. It showed that the proposed architecture selection method and the performance models provide good estimates of overall system performance very quickly, and that a speed improvement of over three orders of magnitude can be achieved with respect to Simulink simulations.

Acknowledgements

We gratefully acknowledge the support of MURST 40% research project “Co-design methods for low-power integrated microelectronic systems”. We also thank Luca Cerrato, Marcello Chiaberge, Dante Del Corso, Francesco Gregoretti, Luca Mari, and Claudio Sansoè for their help with this research work.

7. REFERENCES

[1] F. Balarin, E. Sentovich, M. Chiodo, P. Giusto, H. Hsieh, B. Tabbara, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuki, and A. Sangiovanni-Vincentelli. *Hardware-Software Co-design of Embedded Systems – The POLIS approach*. Kluwer Academic Publishers, 1997.

[2] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, May 1995.

[3] Y.T.S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the Design Automation Conference*, June 1995.

[4] The Mathworks, 2000. See http://www.mathworks.com/products/.

[5] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *IEEE Transactions on VLSI Systems*, 2(4):437–445, December 1994.

[6] Xilinx. Simulink system generator, 2000. See http://www.xilinx.com/ipcenter/.

[7] M. Chiaberge, E. Miranda Sologuren, L.M. Reyneri, “A Pulse Stream System for Low Power Neuro-Fuzzy Computation”, in *IEEE Trans. on Circuits and Systems - I*, Vol. 42, no. 11, November 1995, IEEE Inc., New York (NY), pp. 946-954.

[8] L.M. Reyneri, M. Chiaberge, L. Lavagno, B. Pino, E. Miranda, “Simulink-Based HW/SW Codesign of Embedded Neuro-Fuzzy Systems”, in *Int’l Journal of Neural Systems*, Vol. 10, no. 3, June 2000, pp. 211-226.