HW/SW Co-design of a PCM/PWM converter: a System Level Approach based in the SpecC Methodology

Daniel G. P. Petrini Braz Izaias da Silva Junior

***Abstract*—[Original work from 2005; formatting revised in 2025, with no changes to the results.] We present a case study applying the SpecC methodology within a system-level hardware/software co-design flow to a PCM-to-PWM converter, the core of a Class-D audio amplifier. The converter was modeled and explored with SpecC methodology to derive an HW/SW partition. Using system-level estimates and fast functional simulation, we evaluated mappings that meet real-time constraints while reducing estimated cost of an all-hardware solution and avoiding the expense of a purely software implementation on a high-end processor. Despite the design's moderate complexity, the results underline the value of system-level co-design for early architectural insight, rapid validation, and actionable cost/performance trade-offs.**

***Index Terms*—co-design, System Level, SpecC, SystemC co-design, System Level, SpecC, SystemC**

I. Introduction

The recent requirements of the semiconductors industry has lead the design methodologies evolution in order to fill the design gap detected by the 1999 Roadmap [1]. Today design has very high integration density and complex functionalities to implement arising the necessity to use a higher level of abstraction, the so-called System Level. The latter has many approaches like: reuse methodologies [2], Intellectual Property cores, platform-based-design [3], specification and design languages for the system level and hardware/software co-design. In the latter, functional partitioning and mapping to hardware or software are analyzed to optimize objectives such as cost and time-to-market.

In this paper we present a case study of a design made in the System Level approach using the system level design language SpecC to obtain a hardware and software partitioning. In the next section we mention the design phases of co-design. In section III, co-design computer aided tools are analyzed and in section IV, SpecC is reviewed. Section V contains the description of the case study followed by its results and conclusion.

II. Related Work

In a hardware/software co-design environment, the project can be divided in phases [4]. These phases are specification, modeling, representation, estimation, partitioning, co-synthesis, co-simulation and eventually co-verification. This study focuses primarily on the estimation and partitioning phases of the design flow.

E-mail: dpetrini@alumni.usp.br

An estimator is a tool that, when applied to a candidate implementation, produces quantitative predictions for relevant metrics, thereby enabling comparative evaluation of design alternatives [5].

Partitioning may proceed in a software-oriented manner-starting from a 100% software implementation and progressively migrating selected functional blocks to hardware [6]-or in the reverse direction, beginning with a hardware-based solution and incrementally mapping functions to software [7]. The process first *divides the system* into groups or clusters at an appropriate level of granularity; these resulting blocks are then *mapped to processing elements* (the physical components). Selecting the specific components constitutes *architectural allocation*. From there, blocks are assigned to hardware or software according to an *objective (cost) function* that guides the trade-offs among design goals [6], [8].

Co-design methodologies and tools differ in how they produce a hardware/software partition-either automatically or manually (designer-driven). *Automatic partitioning* integrates algorithms within the tool that ingest the captured system model and propose an optimal split according to a defined cost function, optionally with limited designer guidance. Common algorithmic families include *constructive* methods [8], [9] and *interactive* approaches [7]. In *manual partitioning*-typically the outcome of design space exploration-functional logic blocks are grouped by affinity as expressed in a high-level design-language specification. The identification of these groups (often termed behaviors) is performed by the designer, guided by the architectural and logical characteristics of the system, as in [10].

III. Co-design Tools

The primary purpose of design tools is to automate and accelerate key phases of the development flow, enabling modeling, simulation, validation, and synthesis from a unified system representation.

A wide range of models of computation underpin these tools, including finite-state machines (FSMs), extended FSMs [11], data-flow graphs, statecharts, data-path FSMs [12], program FSMs (the basis of SpecC) [13] and Petri nets. This subject is extensively discussed in the Ptolemy project [14].

Within co-design, the specification should capture the system's full functionality. This specification serves as an executable model that encodes system properties and supports simulation. It forms the foundation for system-level design

exploration-extracting, analyzing, and experimenting with alternative architectural solutions.

Our survey of the area reveals two main classes of co-design tools. The first comprises toolchains originally oriented toward *board-level design*-i.e., printed-circuit boards integrating a CPU, memories, and an ASIC or FPGA-which later converged toward to a second approach that aims towards to the *system-on-chip (SoC)* paradigm, where these elements are integrated into a single device. Representative tools in the first group include Cosyma [6], Vulcan II [7], Ptolemy [14], PISH [15], Polis [16], and SpecSyn [17]. Tools emblematic of the second, SoC-focused group include CoWare [18], SpecC [13], SystemC [19], and, more recently, Spark, which raises the abstraction level by compiling high-level descriptions into HDL [20].

For the case study reported in Section V, we compared several of these tools using the following criteria: (a) compatibility of the modeling formalism with our specification (a C program), (b) support for design-space exploration of the hardware/software partition-preferably with automated assistance, (c) quality of documentation, and (d) availability. Based on this evaluation, we selected a tool grounded in the SpecC language and methodology: the *System-on-Chip Environment (SCE)*, alpha version, which was made available to us by its creators at the University of California, Irvine.

IV. SpecC Methodology

The SpecC methodology is a tool developed to support high levels of abstraction designs. It aims to refine an abstract design specification into a implementation model running in the target architecture. It is based in four models, namely a specification model, an architecture model, a communication model and finally an implementation model. In figure 1 we see these models and the three transformations, architecture Exploration, Communication Synthesis and backend, which comprises hardware and interface synthesis and software compiling, as in [13].

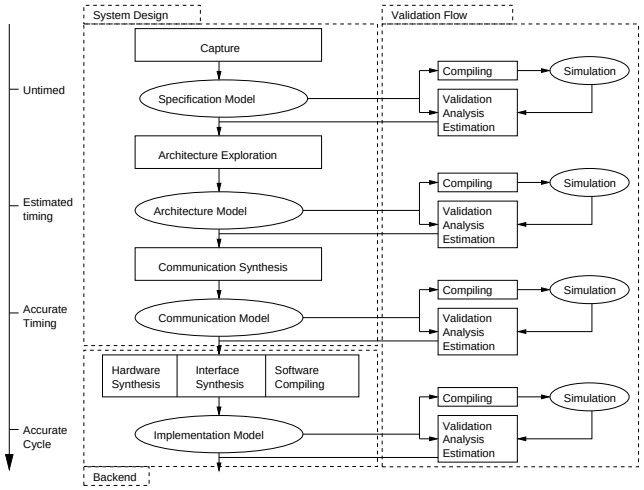

Figure 1. SpecC Models.

The first tree models are written in SpecC language. Each one is compiled and simulated until its validation. Estimations are obtained by operations counting during simulation together with information of the processing elements delays per operation.

The *Specification Model* is a functional description of the system. To create this model, the requirements must be analyzed and translated into the SpecC language. The system is then divided into blocks (behaviors) with similar functionality, thereby defining the granularity and revealing potential parallelism. Simulation at this stage is purely functional (zero-delay), with no timing considerations.

The *Architecture Model* is obtained after *architecture exploration* and *architecture refinement*. Architecture exploration evaluates design alternatives, using behavior estimates and electronic component characteristics to produce a manual hardware/software partition. Architecture refinement then applies transformations such as behavior and variable optimizations. The resulting model reflects the impact of the mapped computation on the chosen architecture and captures the desired hardware-software division.

Communication Model is reached after communication refinement, which adds system buses to the design and maps the abstract communication between components onto the busses. This refinements includes tasks such as communication channels partitioning, protocol insertion above the abstraction of channel and protocol in-lining. The result is a model with much more timing accuracy.

The *Implementation Model* is the model with the lowest level of abstraction. The SpecC code from the previous model, related to the hardware behaviors, should be translated to a language accepted by high-level synthesis as VHDL, because SpecC is not yet synthesizable. The software behaviors should be translated to standard C++ language to be compiled to the target processor. The interface is already allocated in the software or hardware behaviors, so should not have different handling.

SpecC language is a superset of C language [21]. The added constructions allow concurrency modeling, hierarchy, communication, synchronization, state transitions, exception handling and timing - all necessary to embedded systems design. it uses the SpecC Reference Compiler - *scrc*. In the figure 2 we can see some of its constructions.

```
behavior SubSistema ( in  int a,
                      out int b )
{
int x,y;
void main (void)
    {
     x = a;
     y = x * 5;
     b = y;
    }
};
```

(a)

```
void main (void)
{
par {
     f1.main();
     f2.main();
     f3.main();
    }
}
```

(c)

```
void main (void)
{
     f1.main();
     f2.main();
}
```

(b)

```
                        notify e;
                                    wait e;
                                    waitfor 2000;
```

(d)

Figure 2. SpecC Constructions: (a)Behavior declaring, (b)Sequential execution, (c)Parallel execution, (d)Synchronization elements.

The SCE provides a environment for modeling, synthesis and validation. It includes a graphical user interface and the

tools to facilitate the design flow and perform the aforementioned refinements of the SpecC methodology [22].

V. Case Study - The PCM/PWM converter

To assess the benefits of a system-level approach for a design of intermediate complexity, we conducted a case study in which an already implemented design was re-evaluated using a hardware/software co-design methodology. The target design is an audio PCM-to-PWM converter, originally written in Verilog HDL and implemented on an FPGA

Most audio power amplifiers operate in Class A or Class AB configurations. In these cases, the amplifier dissipates too much power, resulting in lower efficiency. By contrast, in digital (switching) amplification, power dissipation is reduced. This category, known as Class-D digital amplifiers, can achieve efficiencies of around 90%. This flow can be seen in figure 3.

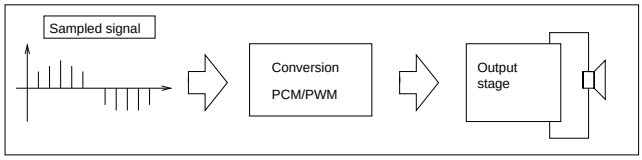

Figure 3. Class D digital amplification.

As the conversion itself is not the main focus of this study, so is the design methodology, we will briefly describe its working principle and its main blocks.

In the pulse-width modulation (PWM), the pulse amplitude is kept constant while the width is proportional to signal samples. In a rough approximation, this pulse width should have the same resolution of the samples, this means that if the signal samples is quantized in n levels, the pulse width should be quantized in n levels as well. For a CD audio signal we have 2^{16}quantization levels and the sampling frequency is 44,1 kHz. To transform the amplitude in pulse width we would need a $2^{16}x44100Hz = 2,89GHz$ resolution. Such frequency is not suitable for implementation in cost expectations of consumer electronics. To solve this problem, we need to add signal processing techniques to convert the PCM into PWM. The main goal is to decrease the operational frequency. Accordingly, we developed a technique that makes this feasible, comprising an algorithm divided into four stages: (a) an up-sampling block that increases the sampling rate by digital interpolation; (b) as PW modulation inserts delays causing harmonic distortion, the linearization block compensated this; (c) the noise shaper block which adjusts the trade-off between number of quantization levels and the new sampling rate; and (d) a wave form generation which performs the pulse creation according to output stage needs.

This stages happens sequentially as shown in figure 4. The frequency reached is about 45 MHz, in its original implementation, which is suitable for consumer electronics applications.

VI. SpecC Usage in case study

The algorithm development were done in Matlab environment and then translated to IBM-PC platform software in order to be validated. Once this algorithm was working in PC, it was converted manually to Verilog hardware description language and implemented in FPGA. Our goal is, from the same specification, which was the C program running in PC platform, develop the co-design case study using SpecC methodology. The implementation in FPGA reached the timing requirements but it ended up requiring a high number of programmable logic elements, so reached a high cost.

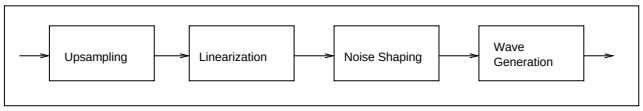

Figure 4. PCM-PWM conversion stages, which guide the subsequent behavior mapping: S0, S1, S2, and S3 (upsampling), LINE (linearization), and MOLD (noise shaping).

The first task was to convert the C program into a SpecC program. In order to validate this process, we developed a testbench that consists of a file containing about four seconds of audio. These samples were converted to PWM by the SpecC model and saved in a file. This file was then reconverted to audio by a command line tool which applied the opposite algorithm and the results were compared to initial audio file by subjective audition.

SCE provides a library of processing elements (PEs) along with associated statistics. During component allocation, the most suitable PEs are selected from this library. Because we used an alpha version, the library was neither complete nor fully accurate; however, it contained sufficient information for our purposes. Table I lists the component characteristics, including cost (in U.S. dollars). The sole exception is the hardware cost, which was derived from the original design's FPGA implementation rather than from the SCE library.

Element	Description	Category	Cost	F(MHz)
DSP	DSP 56600 Motorola	DSP	8	60
uP	Intel Pentium III	Microproc.	40	900
uC	Siemens 80166	Microcontrol.	1	8
HW	Standard RTL processor	FPGA	35	100

Table I
Characteristics of the components.

A. Specification Model

Use of the SpecC tool begins with constructing the *Specification Model*. Our first step was a straightforward translation from C to SpecC, wrapping the original C code in SpecC constructs to establish the testbench. At this stage, the entire code resides in a single SpecC behavior (behavior *Principal* in Fig. 5). Although the design can already be simulated and validated in the SCE environment, further refinement is required to divide this behavior into more partitions.

The next step was to separate convenient groups of functionalities into individual behaviors to facilitate analysis. The computations were grouped into three behaviors corresponding to the up-sampling, linearization, and noise-shaping phases. In the SpecC code, the original C segments implementing these functions were replaced by calls to the newly defined

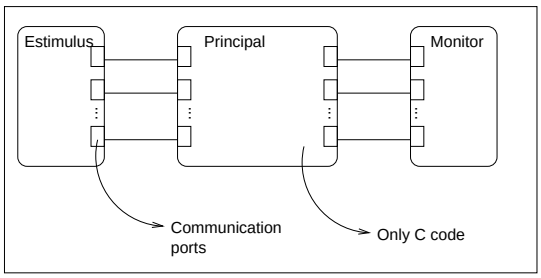

Figure 5. Connectivity between behaviors in the testbench for phase # 1 towards specification model.

behaviors, yielding code that closely resembles a C program with standard function calls. The control structure, four nested 'for' loops, was left unchanged at this point.

The third and final step was more challenging, as it required a modeling style compatible with SCE's automatic tooling for subsequent design phases. To achieve this, the program structure was refactored into a program finite-state machine (program FSM), with each state representing a distinct program-FSM stage. Figure 6 shows the resulting system, which contains only leaf behaviors within the top-level behavior *Principal*. It is worth noting that this re-modeling of the algorithm's control flow took longer than anticipated to reach the specification model.

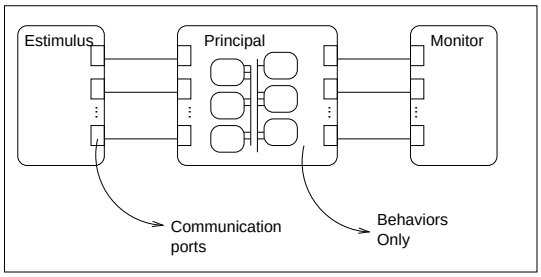

Figure 6. Connectivity of top-level behavior Principal, now only containing leaf-behaviors inside.

The final specification model was composed by six state-behaviors that runs on a fsm fashion. Our intention was to use the minimum number of behaviors because for each added behavior there is a overhead of communication, due to the use of communication ports.

B. Architecture Model

The first step toward this model is *architecture exploration*, a key advantage of our system-level approach, as it quickly reveals how the system should be organized.

Architecture exploration begins by gathering information about the computational cost of the system across different processing elements. In SpecC, estimates are derived from the number of operations counted during simulation. Given this count, we compute execution times for each target platform by applying *weighted operation costs*. For example, on a DSP, a multiply-accumulate (MAC) may require a single cycle, whereas on a general-purpose processor it may take multiple cycles. Combining the weighted operation counts with each component's operating frequency yields the estimated execution time for the design, as shown in Table II.

Element	Type	Cycles	Δt Execution(s)	Goal
DSP	SW	272.935.511	4,54	
uP	SW	935.152.757	1,04	X
uC	SW	935.152.757	116,89	
HW	HW	250.224.089	2,50	X

Table II
PRINCIPAL BEHAVIOR ESTIMATION IN ALL COMPONENTS.

The objective is met when the execution time is shorter than the input file's playback duration, that is, when real-time performance is achieved. We can see it graphically in figure 7. SCE profiler gives other kind of estimates, such as amount of communication, in bytes, between behaviors, variable counting and others (not shown here).

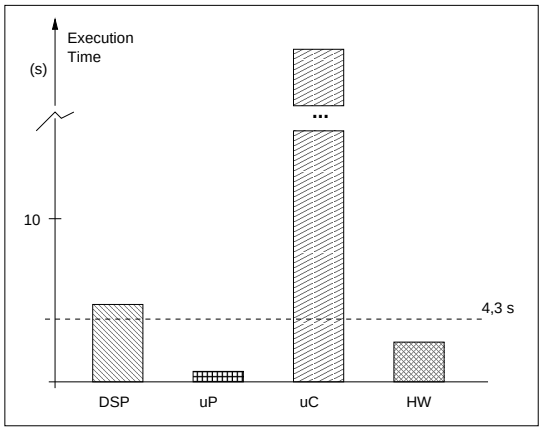

Figure 7. Execution time for different target candidates and the threshold time.

From the estimated execution times, only the Pentium processor (uP) and the fully hardware-based implementation satisfy the real-time requirement. The Pentium option, being a purely software solution that departs from the co-design premise, is also cost-prohibitive and is therefore discarded. Although a full-hardware solution is expensive, we plan to use it only partially, so we continue its analysis. We do the same for the DSP, which nearly met the target and offers an attractive cost profile. The microcontroller (uC), another 100% software option, is eliminated due to inadequate performance for this type of system.

To advance the analysis, we introduce the implementation cost of each behavior, introduced in Figure 4, on each selected processing element (PE). Using the *code size* metric available in SCE (beta) for each PE, we compute the per-behavior cost proportional to its code size and to the PEs total cost. For the *HW* and *DSP* PEs, the base costs are taken from Table I; for the DSP, we add one cost unit for memory, yielding a fixed cost of 9,00. Our intent is to employ both the DSP and HW components. The DSP's cost is fixed, independent of the number of mapped behaviors, since it is entirely software. By contrast, the hardware cost varies with the number of mapped behaviors, the less the number of behavior, the smaller the FPGA needed. Table III reports per-behavior execution times for the two processing elements, FPGA (hardware) and DSP

(software), along with the corresponding proportional costs for the FPGA.

	S0	S1	S2	S3	LINE	MOLD	Total
Δt HW	2,2	183,3	502,0	988,1	77,4	749,1	2502,5s
($) HW	0,01	8,17	10,73	10,65	3,60	1,60	$35,00
Δt DSP	5,4	305,6	836,7	1646,4	188,1	1566,7	4548,9s

Table III
EXECUTION TIME (MS) AND COST (US$). NOTE THAT ALL BEHAVIORS MAPPED TO SOFTWARE (DSP) SUM TO MORE THAN 4.5 S, THEREBY FAILING TO MEET THE 4.3 S REAL-TIME REQUIREMENT.

An analysis of possibilities, or *architecture mapping*, is carried out with the results in the Table III. We analyze the trade-off between required timings, costs and available resources. Our goal is to maximize the software usage because that reflects in cost reduction in this design.

	Mapped in HW	Δt DSP	Δt HW	Δt total	Total cost
1	S3	2902,5	988,1	3890,6	19,65
2	S1, S2, S3	1760,2	1673,4	3433,6	38,55
3	LINE,MOLD	2794,1	826,5	3620,6	14,20
4	MOLD	2982,0	749,1	3731,0	10,60

Table IV
EXECUTION TIME (MS) AND COST ANALYSIS VARYING THE SET OF BEHAVIORS MAPPED IN HW.

From Table IV, *Option 1* maps the heaviest behavior (*S3*) to hardware, meeting the timing constraint at a reasonable cost. However, *S1*, *S2*, and *S3* share the same FIR filter, so placing all three on a single PE is attractive; *Option 2* evaluates this and attains very aggressive timing, albeit at high cost. *Option 3* maps *LINE* and *MOLD* to hardware due to their affinity (shared communication buffers). *Option 4* maps only *MOLD* to hardware and yields the lowest implementation cost. Comparing *Options 3 and 4*, adding *LINE* (Option 3) improves timing by only *3.05%* but increases cost by *25.53%*. Since both total times (3620.6 ms and 3731.0 ms) are below the 4300 ms threshold, *cost* is the decisive metric, favoring *Option 4*.

These were the most relevant execution times and costs to consider. Because the system must run sequentially, options that assume non-sequential behaviors are unsuitable. *State S0* was mapped to the DSP due to its small size and proximity to *S1*. The PWM waveform generator, the final block in Fig. 4, is treated separately and can be mapped either to hardware or to software. Implemented in hardware, it offers high performance with minimal area/cost, as it consists only of a counter, a register, and a comparator. Alternatively, on a software PE it can be realized via an internal peripheral (e.g., the *DSP56003/005* include five on-chip PWMs). For these reasons, it was excluded from the exploration calculations.

According to the analysis in paragraphs above, mapping *MOLD* to hardware satisfies the timing constraints while reducing cost relative to an all-hardware implementation. Figure 8 shows execution time and per-behavior cost for three alternatives: a *100% software* approach on the DSP, a *100% hardware* solution, and a *co-design* configuration that meets the computational requirements at an attractive cost. Thus, we manually defined the system's hardware/software partition.

Once concluded the design space exploration, SCE tool performs automatically the architecture refinement transformation optimizing variables (that serve as communication between behavior mapped in different PEs) and scheduling of behaviors. Thus, we reach architecture model and the system is validated by simulation as in figure 1.

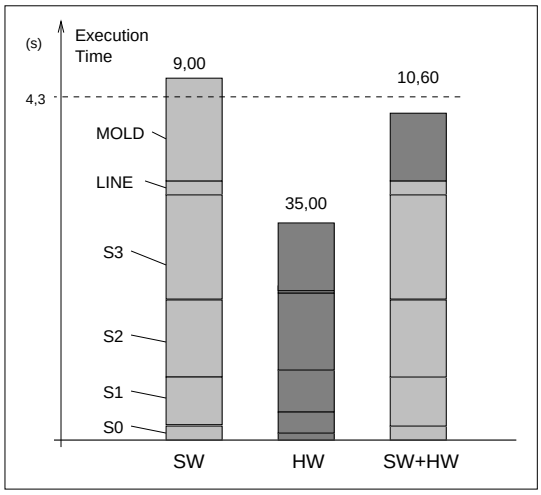

Figure 8. Co-design approach result. Components cost above the bars.

C. Communication and Implementation models

In the communication model, a bus is chosen and its timing is considered in timing analysis. The *communication refinement* automatic transformation, inserts handshake protocol and other details to reflect the communication between PEs and bus. We then validate the system via simulation, which takes longer than before due to the added detail.

The resulting communication model serves as input to the implementation (backend) phase. At this stage, SpecC modules designated as software must be translated into C for the target DSP, while hardware behaviors should be converted into HDL and synthesized. Because the SCE version used did not fully support these features, we did not perform this step in our case study.

VII. RESULTS

For our case study, the SpecC/SCE methodology offered several advantages: (a) the use of estimates to inform architectural decisions; (b) automatic refinement steps that shorten development time; (c) fast, system-level simulationâĂŤless precise but much quicker, enabling earlier validation; and (d) feasible manual exploration thanks to a small number of system behaviors. We also observed limitations: (a) absence of automatic partitioning algorithms; (b) a limited set of exploration metrics; and (c) a reduced processing-element library. It is important to note that the evaluated version was a work in progress, and these gaps may be addressed in future releases.

We also defined comparison criteria between the original RTL-based design flow and the system-level co-design flow used in the case study. Focusing on the most relevant factors for low cost and time-to-market, the summary appears in Table V.

Criterion	Co-design	RTL
Remodeling effort	High to control flow/ low to signal processing	High to both control flow and signal processing
Simulation time	Specif. Model: 1min, Comm.Model: 30 min	High: some hours
Implementation cost	Low cost DSP with reduced area for HW ("small FPGA")	High, only HW
Reuse	Not used, but SpecC supports	Yes, some functions were reused
Lines of Code	800 lines SpecC/C - 500 added	Verilog ~ 10k lines
Refinement	Low effort, Automatic	High effort, manual convertion
HW/SW Partitioning	Manual with only 6 behaviors	No partitioning, only HW

Table V
COMPARISON BETWEEN HARDWARE/SOFTWARE CO-DESIGN AND RTL.

The conversion of C code to SpecC code to reach specification model and the architecture exploration were the phases that dispensed most time of the work. The effort to remodel the code were reasonable, even dealing in the same abstraction level (including the fact that the experience with SpecC grew during the study), but as mentioned in the table above, the computation internal to behaviors were not modified in the C to SpecC conversion.

In the design space exploration phase, we used three criteria, execution time, monetary cost and designer experience. The first two were provided by the project environment and the last is experience from past designs. In our case study the first two were enough to the result.

VIII. CONCLUSIONS

Although we did not produce a final implementation, the evidence summarized in Table V indicates that the adopted approach was satisfactory, yielding meaningful cost optimization. The PCM/PWM converter presents moderate computational complexity, making it amenable to both the co-design flow used here and a conventional RTL flow. Nonetheless, as system complexity continues to grow, system-level co-design becomes increasingly recommended. The signal-processing nature of the algorithm aligned well with SpecC, whose C-based foundation supports broad applicability. For problems specified directly in high-level languages, or easily translated to them, system-level methodologies such as SpecC are effective for producing implementation proposals with actionable estimates, while accelerating development cycles through rapid simulation and early validation.

ACKNOWLEDGEMENTS: We would like to thank Rainer Dömer from University of California, Irvine.

REFERENCES

[1] "International technology roadmap for semiconductors." Available in: <http://public.itrs.net>. Accessed in: 15 of march of 2004.

[2] M. Keating and P. Bricaud, *Reuse Methodology Manual - For System-On-a-Chip Designs*. Kluwer Academic Publishers, second ed., 1999.

[3] A. S. Vincentelli and G. Martin, "Platform-based design and software design methodology for embedded systems," *IEEE Design & Test of Computers*, pp. 23–33, November-December 2001.

[4] K. Kalavade and E. A. Lee, "A hardware-software codesign methodology for DSP applications," *IEEE Design & Test of Computers*, vol. 10, pp. 16–28, Setembro 1993.

[5] D. Gajski and F. Vahid, "Specification and design of embedded software / hardware systems," *IEEE Design & Test of Computers*, vol. 12, Spring 1995.

[6] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis of microcontrollers," *IEEE Design & Test of Computers 10*, pp. 64–75, Dezembro 1993.

[7] R. K. Gupta, J. C. N. Coelho, and G. D. Micheli, "Synthesis and simulation of digital systems containing interacting hardware and software components," in *Proceedings, 29th Design Automation Conference* (I. C. S. Press, ed.), 1992.

[8] C. A. Valderrama, M. E. D. Lima, S. Cavalcante, and E. Barros, *Hardware / Software co-design: projetando hardware e software concorrentemente*. IME-USP, 2000.

[9] M. Baleani, F. Gennari, Y. Jiang, Y. Patel, R. Brayton, and A. Sangiovanni-Vincentelli, "HW/SW partitioning and code generation of embedded control application on a reconfigurable architecture plataform," in *Workshop on Hardware/Software Co-design*, 2002.

[10] D. Gajski, L. Cai, and M. .Olivarez, "Introduction do system level architecture exploration using the SpecC methodology," *ISCAS*, 2000.

[11] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vicentelli, "A formal methodology for hardware/software co-design of embedded systems," *IEEE Micro*, pp. 26–36, Agosto 1994.

[12] D. Gajski, N. D. Dutt, C. H. Wu, and Y. L. Lin, *HIgh Level Synthesis: Introduction to Chip and System Design*. Boston, MA: Kluwer Academic, 1991.

[13] Gerstlauer, Domer, Peng, and Gajski, *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, Maio 2001.

[14] P. group, "Overview of the Ptolemy Project," tech. rep., University of California, Berkeley, CA, USA, 2001.

[15] CIN-UFPE, "Projeto Integrado de Software e Hardware." DisponÃ■vel em <http://www.cin.ufpe.br/ pish/>. Acesso em 10 de Novembro de 2003.

[16] M. Chiodo, P. Giusto, A. Jurecska, H. C. Hsieh, A. Sangiovanni-Vincentelli, and L. Lavagno, "Hardware-software codesign of embedded systems," *IEEE Micro*, vol. 14, pp. 26–36, Agosto 1994.

[17] D. Gajski, F. Vahid, S. Narayan, and J. Gong, "Specsyn: An environment supporting the specify-explorerefine paradigm for hardware/software system design," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 6, no. 1, pp. 84–100, 1998.

[18] CoWare, "CoWare." Available at: <http://www.coware.com>. Accessed in: 15 of march of 2004.

[19] "Welcome to SystemC." Available at: <http://www.systemc.org>. Accessed in: 15 of march of 2004.

[20] "Spark: High Level Synthesis using Parallelizing Compiler Techniques," 2003. Available at: <http://www.cecs.uci.edu/ spark/>. Accessed at 1st of November of 2003.

[21] D. G. Rainer Domer, Andreas Gerstlauer, *SpecC Language Reference Manual*. SpecC Technology Open Consortium, Dezembro 2002. Version 2.0.

[22] S. Abdi, J. Peng, H. Yu, and D. Shin, *System-on-Chip Environment*. University of California, version 2.2.0 beta ed., Julho 2003.