Benefits and Challenges for Platform-Based Design

Alberto Sangiovanni-Vincentelli[††] Luca Carloni[†] Fernando De Bernardinis[†§] Marco Sgroi[*]

[†]Department of EECS
University of California, Berkeley

[§]Dip. di Ingegneria dell'Informazione
Università di Pisa, Italy

[*] DoCoMo Euro-Labs
Munich, Germany

ABSTRACT

Platforms have become an important concept in the design of electronic systems. We present here the motivations behind the interest shown and the challenges that we have to face to make the Platform-based Design method a standard. As a generic term, platforms have meant different things to different people. The main challenges are to distill the essence of the method, to formalize it and to provide a framework to support its use in areas that go beyond the original domain of application.

Categories and Subject Descriptors

B.7 [**Integrated Circuits**]: Design Aids

General Terms

Design, Performance, Standardization.

1. INTRODUCTION

Platform-Based Design [9, 7] has emerged as an important design style as the electronic industry has to face serious difficulties due to three major factors:

- *Disaggregation (or "horizontalization") of the electronic industry* has begun about a decade ago and has affected the structure of the electronics industry favoring the move from a vertically-oriented business model into a horizontally-oriented one. In the past, electronic system companies used to maintain full control of the product development cycle from product definition to final manufacturing. Today, the identification of a new market opportunity, the definition of the detailed system specifications, the development and assembly of the components, and the manufacturing of the final product are tasks performed more and more frequently by distinct organizations. In fact, the complexity of electronic designs and the number of technologies that must be mastered to bring to market winning products have forced electronic companies to focus on their core competence. In this scenario, the integration of the design chain becomes a serious problem at the *hand-off points* from one company to another.

- The pressure for reducing *time-to-market* of electronics products in the presence of exponentially increasing complexity has forced designers to adopt methods that favor component re-use at all levels of abstraction. Furthermore, each organization that contributes a component to the final product naturally strives for flexibility in their design approach that allows to make continuous adjustments and accommodate last-minute engineering changes.

- The dramatic increase in *Non-Recurring Engineering (NRE) costs* due to mask making at the Integrated Circuit (IC) implementation level (a set of masks for the 90 nanometer technology node costs more than two millions US dollars), development of production plants (a new fab costs more than two Billions US dollars), and design (a new generation microprocessor design requires more than 500 designers with all the associated costs in tools and infrastructure!)

has created on one hand the necessity of correct-the-first-time designs and on the other, the push for consolidation of efforts in manufacturing [1].

The combination of these factors has caused several system companies to reduce substantially their ASIC design efforts. Traditional paradigms in electronic system and IC design have to be revisited and re-adjusted or altogether abandoned. Along the same line of reasoning, IC manufacturers are moving towards developing parts that have guaranteed high-volume production form a single mask set (or that are likely to have high-volume production, if successful) thus moving differentiation and optimization to reconfigurability and programmability.

Platform-based design has emerged over the years as a way of coping with the problems listed above. The term "platform" has been used in several domains: from service providers to system companies, from tier 1 suppliers to IC companies. In particular, IC companies have been very active lately to espouse platforms. The TI OMAP platform for cellular phones, the Philips Viper and Nexperia platforms for consumer electronics, the Intel Centrino platform

[1]The cost of fabs have changed the landscape of IC manufacturing in a substantial way forcing companies to team up for developing new technology nodes (see, for example, the recent agreement among Motorola, Philips and ST Microelectronics and the creation of Renesas in Japan).

DAC 2004, June 7–11, 2004, San Diego, California, USA.

for laptops, are but a few examples. Recently, Intel has been characterized by its CEO Ottellini as a "platform company".

As is often the case for fairly radical new approaches, the methodology emerged as a sequence of empirical rules and concepts but we have reached a point where a rigorous design process was needed together with supporting EDA environments and tools. The research carried out in the MARCO GSRC (see [6, 7, 9]) has identified the basic tenets of the methodology. Platform-based design

- lies the foundation for developing economically feasible design flows because it is a structured methodology that *theoretically limits the space of exploration, yet still achieves superior results in the fixed time constraints of the design*;
- provides a formal mechanism for identifying the most critical hand-off points in the design chain: the hand-off point between system companies and IC design companies and the one between IC design companies (or divisions) and IC manufacturing companies (or divisions) represent the *articulation points* of the overall design process;
- eliminates costly design iterations because it fosters *design re-use* at all abstraction levels thus enabling the design of an electronic product by assembling and configuring platform components in a rapid and reliable fashion;
- provides an intellectual framework for the complete electronic design process.

This paper presents the foundations of this discipline and outlines a variety of domains where the PBD principles can be applied. In particular, in Section 2 we define the main principles of PBD. Our goal is to provide a precise reference that may be used as the basis for reaching a common understanding in the electronic system and circuit design community. Then, we present the platforms that define the articulation points between system definition and implementation (Section 3). As a demonstration of applicability of the Platform-Based Design paradigm to all levels of design. in the following sections, we show that platforms can be applied to very high levels of abstraction such as communication networks (Section 4), where platforms provide connectivity and services, as well as to low levels such as analog parts (Section 5), where performance is the main focus.

2. PLATFORM-BASED DESIGN

The basic tenets of platform-based design are the identification of design as a *meeting-in-the-middle process*, where successive refinements of specifications meet with abstractions of potential implementations, and the identification of precisely defined layers where the refinement and abstraction processes take place. Each layer supports a design stage providing an opaque abstraction of lower layers that allows accurate performance estimations. This information is incorporated in appropriate parameters that annotate design choices at the present layer of abstraction. These layers of abstraction are called *platforms*.

A platform is a *library of components together with their composition rules*. A design at each level of abstraction is a *platform instance*, i.e., a legal composition of a set of library elements. The library not only contains *computational* blocks that carry out the appropriate computation but also *communication* components that are used to interconnect the functional components. Each element of the library has a characterization in terms of performance parameters together with the functionality it can support. For every platform level, there is a set of methods used to map the upper layers of abstraction into the platform and a set of methods used to estimate performances of lower level abstractions. The meeting-in-the-middle process is the combination of two efforts:

- **top-down**: map an instance of the top platform with constraints into an instance of the lower platform with appropriate constraints resulting from an appropriate propagation involving budgeting wherever needed;
- **bottom-up**: build a platform by defining its components and their performance abstraction (e.g., number of literals for technology independent optimization, and area and propagation delay for a cell in a standard cell library).

Establishing the number, location, abstraction and components of intermediate platforms is the essence of platform-based design. The trade-offs involved in the selection of the number and characteristics of platforms relate to the size of the design space to be explored and the accuracy of the estimation of the characteristics of the solution adopted. Naturally, the larger the step across platforms, the more difficult is predicting performance, optimizing at the higher levels of abstraction, and providing a tight lower bound. In fact, the design space for this approach may actually be smaller than the one obtained with smaller steps because it becomes harder to explore meaningful design alternatives and the restriction on search impedes complete design space exploration. Ultimately, predictions/abstractions may be so inaccurate that design optimizations are misguided and the lower bounds are incorrect.

3. SYSTEM PLATFORM STACK

The articulation point between system definition and implementation is a critical one for design quality and time. Indeed, the very notion of platform-based design originated at this point (see [3, 4, 6, 7]). In [6, 7, 9], we have discovered that at this level there are two distinct platforms forming a *system platform stack*: a *(micro-)architecture platform* and an API platform. The API platform allows system designers to use the *services* that a (micro-)architecture offers them. In the world of Personal Computers, this concept is well known and is essential to the development of application software on different hardware that share some commonalities allowing the definition of a unique API.

3.1 (Micro-) Architecture Platforms

Integrated circuits used for embedded systems will most likely be developed as an instance of a particular *(micro-) architecture platform*. That is, rather than being assembled from a collection of independently developed blocks of silicon functionalities, they will be derived from a specific *family of micro-architectures*, possibly oriented toward a particular class of problems, that can be extended or reduced by the system developer. The elements of this family are a sort of "hardware denominator" that could be shared across multiple applications. Every element of the family can be obtained quickly through the personalization of an appropriate

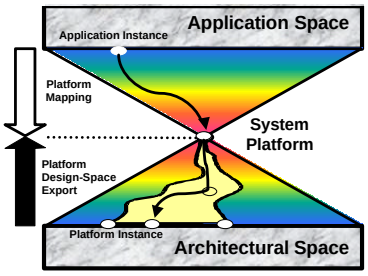

Figure 1: System Platform Stack.

set of parameters controlling the micro-architecture. Often the family may have additional constraints on the components of the library that can or should be used. Depending on the implementation platform that is chosen, each element of the family may still need to go through the standard manufacturing process including mask making. This approach then conjugates the need of saving design time with the optimization of the element of the family for the application at hand.

The flexibility, or the capability of supporting different applications, of a *platform instance* is guaranteed by programmable components. Programmability will ultimately be of various forms. One is software programmability to indicate the presence of a microprocessor, DSP or any other software programmable component. Another is hardware programmability to indicate the presence of reconfigurable logic blocks such as FPGAs, whereby logic function can be changed by software tools without requiring a custom set of masks. Some of the new architecture and/or implementation platforms being offered on the market mix the two into a single chip.

3.2 API Platform

The concept of architecture platform by itself is not enough to achieve the level of application software re-use we require. The architecture platform has to be abstracted at a level where the application software "sees" a high-level interface to the hardware that we call Application Programm Interface (API) or Programmers Model. A software layer is used to perform this abstraction. This layer wraps the essential parts of the architecture platform, which are the programmable cores and the memory subsystem via a Real Time Operating System (RTOS), the I/O subsystem via the Device Drivers, and the network connection via the network communication subsystem.

In our framework, the API or Programmers Model is a unique abstract representation of the architecture platform via the software layer. With an API so defined, the application software can be re-used for every platform instance. Indeed the Programmers Model (API) is a platform itself that we can call the API platform. Of course, the higher the abstraction level at which a platform is defined, the more instances it contains. For example, to share source code, we need to have the same operating system but not necessarily the same instruction set, while to share binary code, we need to add the architectural constraints that force to use the same ISA, thus greatly restricting the range of architectural choices.

3.3 System Platform

The basic idea of system platform is captured in Figure 1. The vertex of the two cones represents the combination of the API or Programmers' Model and the architecture platform. A system designer maps its application into the abstract representation that "includes" a family of architectures that can be chosen to optimize cost, efficiency, energy consumption and flexibility. The mapping of the application into the actual architecture in the family specified by the Programmers' Model or API can be carried out, at least in part, automatically if a set of appropriate software tools (e.g., software synthesis, RTOS synthesis, device-driver synthesis) is available. It is clear that the synthesis tools have to be aware of the architecture features as well as of the API. This set of tools makes use of the software layer to go from the API platform to the architecture platform. Note that the system platform effectively decouples the application development process (the upper triangle) from the architecture implementation process (the lower triangle). Note also that, once we use the abstract definition of "API" as described above, we may obtain extreme cases such as traditional PC platforms on one side and full hardware implementation on the other. Of course, the programmer model for a full custom hardware solution is trivial since there is a one-to-one map between functions to be implemented and physical blocks that implement them. In this latter case, platform-based design amount to adding to traditional design methodologies some higher level of abstractions.

4. NETWORK PLATFORMS

In distributed systems, the design of the protocols that support the communication among system components is difficult due to the tight constraints on performance and cost. To make the communication design problem more manageable, designers usually decompose the communication function into distinct protocol layers, and design each layer separately. At the same time, meeting the tight performance constraints of the present embedded system applications often requires to optimize also across layers. The definition of an optimal layered architecture, the design of the correct functionality for each protocol layer, and the design space exploration for the choice of the physical implementation must be supported by tools and methodologies that allow to evaluate the performance and guarantee the satisfaction of the constraints after each step. Below we formalize the concept of Network Platform and outline a methodology for selecting, composing and refining Network Platforms [10].

A *Network Platform (NP)* is a library of resources that can be selected and composed together to form a Network Platform Instance (NPI) and support the interaction among a group of interacting components. The structure of an NPI is defined abstracting computation resources as nodes and communication resources as links. Ports interface nodes with links or with the environment of the NPI. The behaviors and the performances of an NPI are defined in terms of the type and the quality of the communication services it offers. We formalize the behaviors of an NPI using the Tagged Signal Model [8]. NPI components are modeled as processes and events model the instances of the send and receive actions of the processes. An event is associated with a message which has a type and a value and with tags that

specify attributes of the corresponding action instance (e.g. when it occurs in time). The set of behaviors of an NPI is defined by the intersection of the behaviors of the component processes.

A Network Platform Instance is defined as a tuple $NPI = (L, N, P, S)$, where: 1) $L = \{L_1, L_2, ...L_{Nl}\}$ is a set of directed links, 2) $N = \{N_1, N_2, ...N_{Nn}\}$ is a set of nodes, 3) $P = \{P_1, P_2, ...P_{Np}\}$ is a set of ports. A port P_i is a triple (N_i, L_i, d), where $N_i \in N$ is a node, $L_i \in L \cup Env$ is a link or the NPI environment and $d = in$ if it is an input port, $d = out$ if it is an output port. The ports that interface the NPI with the environment define the sets $P^{in} = \{(N_i, Env, in)\} \subseteq P, P^{out} = \{(N_i, Env, out)\} \subseteq P$, 4) $S = \bigcap_{Nn+Nl} R_i$ is the set of behaviors, where R_i indicates the set of behaviors of a resource that can be a link in L or a node in N.

The basic services provided by an NPI are called *Communication Services (CS)*. A CS consists of a sequence of message exchanges through the NPI from its input to its output ports. A CS can be accessed by NPI users through the invocation of send and receive *primitives* whose instances are modeled as events. An *NPI Application Programming Interface (API)* consists of the set of methods that are invoked by the NPI users to access the CS. For the definition of an NPI API it is essential to specify not only the service primitives but also the type of CS they provide access to (e.g. reliable send, out-of-order delivery). Formally, a Communication Service (CS) is a tuple $(\overline{P}^{in}, \overline{P}^{out}, M, E, h, g, <^t)$, where $\overline{P}^{in} \subseteq P^{in}$ is a non-empty set of NPI input ports, $\overline{P}^{out} \subseteq P^{out}$ is a non-empty set of NPI output ports, M is a non-empty set of messages, E is a non-empty set of events, h is a mapping $h : E \rightarrow (\overline{P}^{in} \cup \overline{P}^{out})$ that associates each event with a port, g is a mapping $g : E \rightarrow M$ associating each event with a message, $<^t$ is a total order on the events in E. A CS is defined in terms of the number of ports, that determine, for example, if it is a unicast, multicast or broadcast CS, the set M of messages representing the exchanged information, the set E including the events that are associated with the messages in M and model the instances of the send and receive methods invocations. The CS concept is useful to express the correlation among events, and explicit, for example, if two events are from the same source or are associated with the same message.

NPIs can be classified according to the number, the type, the quality and the cost of the CS they offer. Rather than in terms of event sequences, a CS is more conveniently described using *QoS parameters* like error rate, latency, throughput, jitter, and *cost parameters* like consumed power and manufacturing cost of the NPI components. QoS parameters can be simply defined using annotation functions that associate individual events with quantities, such as the time when an event occurs and the power consumed by an action. Hence, one can compare the values of pairs of input and output events associated with the same message to quantify the error rate, or compare the timestamp of events observed at the same port to compute the jitter. The number of CS that an NPI can offer is large, so the concept of Class of Communication Services (CCS) is useful to simplify the description of an NPI. CCS define a new abstraction (and therefore a platform) that groups together CS of similar type and quality. For each NPI supporting multiple CS, there are several ways to group them into CCS. It is task of the NPI designer to identify the CCS and provide the proper abstractions to facilitate the use of the NPI.

4.1 Design of Network Platforms

The design methodology derives an NPI implementation by successive refinement from the specification of the behaviors of the interacting components and the declaration of the constraints that an NPI implementation must satisfy. The most abstract NPI is defined by a set of end-to-end direct logical links connecting pairs of interacting components. Communication refinement of an NPI defines at each step a more detailed NPI$'$ by replacing one or multiple links in the original NPI with a set of components or NPIs. During this process another NPI can be used as a resource to build other NPIs. A correct refinement procedure generates an NPI$'$ that provides CS equivalent to those offered by the original NPI with respect to the constraints defined at the upper level. A typical communication refinement step requires to define both the structure of the refined NPI$'$, i.e. its components and topology, and the behavior of these components, i.e. the protocols deployed at each node. One or more NP components (or predefined NPIs) are selected from a library and composed to create CS of better quality. Two types of compositions are possible. One type consists of choosing a NPI and extending it with a protocol layer to create CS at a higher level of abstraction (vertical composition). The other type is based on the concatenation of NPIs using an intermediate component called adapter (or gateway) that maps sequences of events between the ports being connected (horizontal composition).

5. ANALOG PLATFORMS

An analog platform consists of performance models $\mathcal{P}(\zeta)$, behavioral models $\mu(in, out, \zeta)$ and interconnection models $\iota(in, out, \zeta_A, \zeta_B)$, where $\mu(in, out, \zeta)$ is a parameterized executable model that introduces at the functional level a number of non-idealities due to the actual circuit implementation and ζ is a vector of parameters controlling the actual behavior of the model (e.g. specific gain and noise values). Even though behavioral models introduce a number of non-idealities in system simulation, their actual scope is quite limited since ζ can assume any value. In this sense, a behavioral model is a functional model that does not expose any architectural effect, which however are very important when exploring different tradeoffs at system level. $\mathcal{P}(\zeta)$ are provided by analog platforms to constrain μ to feasible behaviors according to the selected architecture/topology. Performance models ($\mathcal{P}(\cdot)$) are represented as a relation on ζ. $\iota(in, out, \zeta_A, \zeta_B)$ are specialized behavioral models that take into account interface issues in the composition between block A and block B. Behavioral models have no intrinsic loading notion. This may have a serious impact on circuit performance since circuit composition may significantly alter individual circuit performance. Unfortunately, it is not possible to give general guidelines on how to model analog communication. The required effort obviously depends on the mutual dependence of the connected components. For linear systems, the specification of input and output impedances is sufficient to model the interconnection. However, for complex interfaces, loading effects between blocks A and B may be explicitly included in the behavioral models μ_A and μ_B so that the composition $\mu_A \circ \iota \circ \mu_B$ is correct. One possible approach is to model all the composition ef-

fects in the output port of the driving block (e.g. A) so that all performance figures of A (ζ_A) embed the loading effect of B. Consequently, the performance model has to include some variables that parameterize the output load as well as other possible parameters affecting ι. For example, given an amplifier with {*gain, bandwidth, power*}, and characterizing its output load with the area of the next input stage MOS transistors, its performance model introduces a dependence of gain, bandwidth and power on the next stage MOST area. Therefore, the behavioral model μ_A does not have to explicitly model the loading of B, since its effects are accounted for by the relation existing on ζ_A. This modeling scheme is very general, and can model very complex interactions.

Analog Platforms can be generated at multiple levels of abstraction. Platforms can be hierarchically organized into platforms stacks. At each level of the stack, an optimization process can be used to map constraints from one level to the next. Platform stacks provide a unifying framework to model both the system abstraction hierarchy and the system refinement process. Essential to analog platform is that performance models constrain behavioral optimizations/explorations to the feasibility region of the current platform level, so that the next level constraints is feasible and exploration can proceed.

5.1 Performance Models

Performance models are at the heart of analog platforms. Because of the nature of analog designs, performance models have to characterize continuous variations in performance as a function of continuous variations of design parameters. For accuracy, $\mathcal{P}$s are derived bottom-up from general performance evaluation schemes, such as simulation for circuit platforms. Differently from common approaches that represent performances through a regression on circuit configuration parameters, e.g., [5], relations on performance parameters are directly modeled by means of characteristic functions. Hiding architecture parameters (regression variables) allows design flows to proceed top-down and enables different architectures to be directly compared based on the effects they introduce. Given a behavioral model $\mu(in, out, \zeta)$, a performance model $\mathcal{P}$ constrains μ to feasible values of ζ ($\mathcal{P}(\zeta) = 1$). Performance models are defined by:

1. *Input space* $\mathcal{I}$ - Given a circuit $\mathcal{C}$ and m parameters controlling its configuration (a vector κ), $\mathcal{I}_\mathcal{C} \subseteq \mathbb{R}^m$ is the set of κ over which we want to characterize $\mathcal{C}$.
2. *Output space* $\mathcal{O}$ - Given a circuit $\mathcal{C}$ and n performance figures (a vector ζ) characterizing its behavioral model, $\mathcal{O}_\mathcal{C} \subseteq \mathbb{R}^n$ is the set of ζ that are achievable by $\mathcal{C}$.
3. *Evaluation function* ϕ - Given a circuit $\mathcal{C}$, $\mathcal{I}_\mathcal{C}$ and $\mathcal{O}_\mathcal{C}$, $\phi_\mathcal{C} : \mathcal{I} \rightarrow \mathcal{O}$ allows translating a parameter m-tuple set into a performance n-tuple set.
4. *Performance relation* $\mathcal{P}$ - Given a circuit $\mathcal{C}$, $\mathcal{I}_\mathcal{C}$, $\mathcal{O}_\mathcal{C}$ and $\phi_\mathcal{C}$, we define the performance relation of $\mathcal{C}$ given $\mathcal{I}_\mathcal{C}$ and $\phi_\mathcal{C}$ to be $\mathcal{P}_\mathcal{C}$ on $\mathbb{R}^n$ that holds only for points $o \in \mathcal{O}_\mathcal{C}$. With a little abuse in notation, we will denote both the performance relation characteristic function $\chi_\mathcal{P}(x) : \mathbb{R}^n \rightarrow \{0, 1\}$ and the relation itself with $\mathcal{P}_\mathcal{C}(\zeta)$.

We rely on statistical sampling of $\mathcal{I}$ to generate a set of performance vectors ζ and on statistical learning theory to generate an approximation to $\mathcal{P}_\mathcal{C}$. The bottom-up characterization process is potentially expensive in terms of number of simulations required. However, by choosing the right

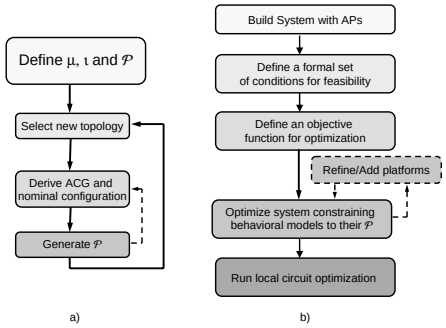

Figure 2: Design flow with analog platforms. Bottom-up (a) and top-down (b) design phases are shown.

level of granularity for platform generation, using ad hoc heuristics to limit $\mathcal{I}$ (see Sec. 5.2) and eventually exploiting approximate simulation methods such as Model Order Reduction schemes, the cost of characterization can be made affordable.

The problem of approximating $\mathcal{P}$ from performance samples can be stated as a classification problem. We adopt an approximation method based on Support Vector Machines (SVM) to represent $\mathcal{P}$ as described in [2]. SVMs provide a favorable approximation scheme since they lead to compact models with smooth approximation and good generalization properties. In particular, we use SVMs with a Gaussian RBF kernel that provide a basis function for the approximation of the form:

$$f(x) = sign(\sum_i \alpha_i e^{-\gamma|x-x_i|^2} - \rho) \quad (1)$$

Vectors x_i are performance vector obtained through simulation, γ is a kernel parameter that controls approximation features. The coefficients α_i, ρ are weight and bias parameters that can be efficiently computed solving a convex problem. x_is are a subset of the simulated ζ also selected through the convex problem. Since only vectors useful for classification are retained, compact performance models can be obtained.

5.2 Analog Constraint Graphs

The *bottom-up* characterization of Analog platforms is usually achieved using a simulator as the mapping function $\phi(\cdot)$. Thus, it is possible to characterize accurately complex blocks. The overall characterization cost is proportional to the simulation time for each circuit. The number of simulations is exponentially dependent on the number of dimensions of the configuration space $\mathcal{I}$. However, even if $\mathcal{I} \subseteq \mathbb{R}^n$, the n dimensions of $\mathcal{I}$ are usually strongly correlated in "functional" circuits. For example, matching requirements, device stacking, operating region enforcement introduce a set of constraints on κ that effectively reduces the size of $\mathcal{I}$. We can therefore define an effective $\mathcal{I}_{eff}$ as the subset of $\mathcal{I}$ for which the following constraints hold

$$\begin{cases} f_i(\kappa) = 0 & \text{for } i = 1 \dots n \\ g_j(\kappa) < 0 & \text{for } j = 1 \dots m \end{cases} \quad (2)$$

These constraints can be used to bias properly sampling so that approximations of $\mathcal{P}$ can be obtained efficiently. The

constraints in 2 can be easily supplied by the platform designer using simple analytical equations and inequalities. Since constraint equations are usually analytic approximations of the underlying circuit relations, some attention has to paid on the *conservativeness* of constraints. A set of constraints is conservative when $\mathbb{R}^n \setminus \mathcal{I}_{eff}$ corresponds only to incorrect circuits, even though some incorrect circuit may be generated for some $\kappa \in \mathcal{I}_{eff}$. A most useful way to represent 2 is through bipartite non directed graphs. An Analog Constraint Graph (ACG) is defined as a (Ξ, Ψ, Υ) where Ξ is the set of design variables, Ψ is the set of constraints on ξs and $\Upsilon \subseteq \Xi \times \Psi \cup \Psi \times \Xi$ is the set of edges that link design variable ξ_i to constraint ψ_j. Bipartite graphs are a natural way to operate on systems of equations, as in the DONALD workbench [11]. Statistical sampling can leverage the graph representation of 2 to obtain an operative way of drawing samples in $\mathbb{R}^n$. This can be achieved obtaining a schedule of the ACG that leads to very efficient code to generate random configurations. Exploiting ACGs, the characterization cost can be cut by orders of magnitude in terms of number of simulations at virtually no expense in terms of (useful) performances. For some LNA and mixer platforms that we generated, the reduction was respectively on the order of 10^4 and 10^3, leading to characterization times on the order of a couple of days.

5.3 AP Design Flow

Analog design with APs consists of two separate phases. Platform characterization is the bottom-up platform generation process that, starting from a given set of candidate architectures, builds the models $(\mu, \iota, \mathcal{P})$. Each platform can be successively refined to improve accuracy locally during the design space exploration process. This process is shown in Fig. 2a. The second phase consists of casting the design problem as an optimization problem on platforms that implements constraint propagation through a platform stack until a bottom level platform instance is defined (Fig. 2b). Since several circuit topologies (lower platform levels) can be merged at higher abstraction levels for a given functionality, constraint propagation intrinsically performs topology selection. In fact, performance models constraints force the optimization process to pick an optimal (feasible) point that may correspond to architecture (platform) A rather than B, thus selecting platform B for further refinements. In conclusion, analog platforms allow raising the level of abstraction of system level analog design, and the apparently increased effort in developing platform models is largely balanced by the exploration and optimization possibilities that are achievable exploiting APs.

6. CONCLUDING REMARKS

Platform-based design (PBD) is an all-encompassing intellectual framework in which scientific research, design tool development, and design practices can be embedded and justified. In our definition, a platform is simply an abstraction layer that hides the details of the several possible implementation refinements of the underlying layer. The main benefit of platform-based design is that it allows designers to tradeoff various components of manufacturing, NRE and design costs while sacrificing as little as possible potential design performance. The main challenges in adopting this methodology are all related to the lack of precise definitions and characterization of platforms and of the associated design flow in the industry today. This in turns causes difficulties in moving designers from commonly used methodologies such as the ASIC flow to this paradigm and in developing the appropriate tools to support it. We argued in this paper that the value of PBD can be multiplied by providing an appropriate set of tools and a general framework where platforms can be *formally* defined in terms of rigorous semantics, manipulated by appropriate synthesis and optimization tools and verified. Examples of platforms have been given using the concepts that we have developed. We conclude by mentioning that the Metropolis design environment [1], a federation of integrated analysis, verification, and synthesis tools supported by a rigorous mathematical theory of metamodels and agents, has been designed to provide a general open-domain PBD framework.

7. REFERENCES

[1] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: an integrated electronic system design environment. *IEEE Computer*, 36:45–52, April 2003.

[2] F. De Bernardinis, M.I. Jordan, and A.L. Sangiovanni Vincentelli. Support vector machines for analog circuit performance representation. In *Proc. of the Design Automation Conf.*, June 2003.

[3] F. Balarin et al. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, Boston/Dordrecht/London, 1997.

[4] H. Chang et al. *Surviving the SOC Revolution: A Guide to Platform Based Design,* . Kluwer Academic Publishers, Boston/Dordrecht/London, 1999.

[5] H. Liu et al. Remembrance of circuits past: Macromodeling by data mining in large analog design spaces. In *Proceedings of DAC*, 2002.

[6] A. Ferrari and A. L. Sangiovanni-Vincentelli. System Design: Traditional Concepts and New Paradigms. In *Proc. Intl. Conf. on Computer Design*, pages 1–12, October 1999.

[7] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12), December 2000.

[8] E. A. Lee and A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

[9] A. Sangiovanni-Vincentelli. Defining Platform-Based Design. In *www.eedesign.com/story/OEG20020204S0062*, February 2002.

[10] A. L. Sangiovanni-Vincentelli and M. Sgroi. Service-based Model and Methodology for Network Platforms. Technical Report available at www-cad.eecs.berkeley.edu/šgroi/tech_reports, University of California, Berkeley, CA 94720, June 2002.

[11] K. Swings and W. Sansen. Donald: a workbench for interactive design space exploration and sizing of analog circuits. In *Proceedings of the European Conference on Design Automation. EDAC*, pages 475–479, 1991.