System Design: Traditional Concepts and New Paradigms

Alberto Ferrari
PARADES EEIG,
Via San Pantaleo, 66, 00186 Rome, Italy
aferrari@parades.rm.cnr.it

Alberto Sangiovanni-Vincentelli
Department of EECS,
University of California at Berkeley,
PARADES EEIG,
Via San Pantaleo, 66, 00186 Rome, Italy
alberto@ic.eecs.berkeley.edu

Abstract

Recent advances in system design are presented. The shift towards flexible hardware architectures that can support a variety of applications via programmability and re-configurability is underlined. Essential to this process is the definition and use of platforms. We give an abstract definition of platform and show its use in system design drawing examples from the automotive system design field.

1. Introduction

System design is undergoing a series of radical transformations to meet performance, quality, safety, cost and time-to-market constraints introduced by the pervasive use of electronics in everyday objects. An essential component of the new system design paradigm is the *orthogonalization of concerns*, i.e., the separation of the various aspects of design to allow more effective exploration of alternative solutions. In particular, the pillar of the design methodology that we have proposed over the years is the separation between:

- function (what the system is supposed to do) and architecture (how it does it);
- communication and computation.

The mapping of function to architecture is an essential step from conception to implementation. When mapping the functionality of the system to an integrated circuit, the economics of chip design and manufacturing are essential to determine the quality and the cost of the system. Since the mask set and design cost for Deep Sub-Micron implementations is predicted to be overwhelming, it is important to find common architectures that can support a variety of applications [1]. To reduce design costs, re-use is a must. In particular, since system designers will use more and more frequently software to implement their products, there is a need for design methodologies that allow the substantial re-use of software. This implies that the basic architecture of the implementation is essentially "fixed", i.e., the principal components should remain the same within a certain degree of parameterization. For embedded systems, which we believe are going to be the dominant share of the electronics market, the "basic" architecture consists of programmable cores, I/O subsystem and memories. A family of architectures that allow substantial re-use of software is what we call a **hardware platform**. We believe that hardware platforms will take the lion's share of the IC market. However, the concept of hardware platform by itself is not enough to achieve the level of application software re-use we are looking for. To be useful, the hardware platform has to be abstracted at a level where the application software sees a high-level interface to the hardware that we call **Application Program Interface or API**. There is a software layer that is used to perform this abstraction. This layer wraps the different parts of the hardware platform: 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. This layer is called the **software platform.** The combination of the hardware and the software platforms is called the **system platform.**

In this paper, we first review the principles of system design, then offer a rigorous definition of platforms and show a methodology for their selection and use. We point to a forthcoming book for a detailed discussion of platform-based design using a less specific and formal view[1].

2. System Design Principles

The overall goal of electronic system design can be summarized as follows:

Minimize
- **production cost,**

- **development time and cost**

subject to constraints on performance and functionality of the system.

2.1. Production cost

Manufacturing cost depends mainly on the hardware components of the product. Minimizing production cost is the result of a balance between competing criteria. If we think of an integrated circuit implementation, then the size of the chip is an important factor in determining production cost. Minimizing the size of the chip implies tailoring the hardware architecture to the functionality of the product. However, the cost of a state-of-the-art fabrication facility continues to rise: it is estimated that a new 0.18µm high-volume manufacturing plant costs approximately $2-3B today. This increasing cost is prejudicing the manufacturers towards *parts that have guaranteed high-volume production form a single mask set* (or that are likely to have high volume production, if successful.) This translates to better response time and higher priorities at times when global manufacturing resources are in short supply.

In addition, the NRE costs associated with the design and tooling of complex chips are growing rapidly. The ITRS predicts that while manufacturing complex System-on-Chip designs will be practical, at least down to 50nm minimum feature sizes, *the production of practical masks and exposure systems* will likely be a major bottleneck for the development of such chips. That is, the *cost of masks will grow even more rapidly for these fine geometries, adding even more to the up-front NRE for a new design.* A single mask set and probe card cost for a state-of-the-art chip is over $1M for a complex part today, up from less than $100K a decade ago (*note: this does not include the design cost*). In addition, the cost of developing and implementing a comprehensive test for such complex designs will continue to represent an increasing fraction of a total design cost unless new approaches are developed.

As a consequence of this evolution of the Integrated Circuit world, if we determine a common "hardware" denominator (which we can call for now, **platform**) that could be shared across multiple applications, production volume increases and overall costs may eventually be (much) lower than in the case when the chip is customized for the application.

Of course, the choice of the platform has to be based on production volume but we cannot just forget the size of the implementation since a platform that can support the functionality and performance required for a "high-end" product may end up being too expensive for other lower complexity products. Today the choice of a platform is more an art than a science. We believe that a system design methodology must assist designers in this difficult choice with metrics and with early assessments of the capability of a given platform to meet design constraints (see for a comprehensive discussion of metrics to be adopted for platform-based design).

2.2. Development Cost & Time

As the complexity of the products under design increases, the development efforts increase exponentially. Hence to keep these efforts in check a design methodology that favors **re-use** and early error detection is essential. In addition, development time must be shorter and shorter to meet time-to-market requirements. Short development times often imply changes in specifications while the product is being designed. Hence**, flexibility**, i.e., the capability of the platform to adapt to different functionalities without significant changes, is a very important criterion to measure the quality of a platform. Note that flexibility and re-use are related: the more flexible a platform is, the more re-usable. In addition to the ability to cover different behaviors, another important quality of a platform is the range of **performance**. The flexibility/performance trade-off is very complex and it is strictly related to the ability of the platform to match a set of applications. For example, a hardware block implementing a digital filter with hardwired taps is not flexible since it cannot perform any other operation. The same function of digital filtering can be achieved using a software-programmable Digital Signal Processor (DSP) that is clearly much more flexible since it can perform a large set of operations. Limiting, for sake of simplicity, the set of behavior to the class of finite impulse response (FIR) digital filters, the length of the filter response or number of taps characterize the filtering function, the maximum sampling frequency its performance. The filtering function implemented as a hardware block is limited to a fixed number of taps, while the filtering function implemented on a DSP can have a variable number of taps, only limited by the available memory of the DSP. On the other hand, the performance range for the hardware implementation of the digital filter is fairly large, while for the DSP implementation is lower and decreases with the number of taps.

2.2.1. Flexibility

The most flexible platforms include **software programmable components**. In this case, flexibility is achieved using a "machine" able to perform a set of instructions specified by the instruction set architecture (ISA). The execution of the sequence of instructions, or instruction stream stored in memory, realizes the desired behavior (Processor unit, micro-code unit). Powerful 32-bit microprocessors are very flexible since they can perform an almost unlimited number of functions with good performance, small 8-bit microprocessors have the same functional flexibility but their performance is clearly much more limited.

An intermediate point between software programmable components and hardware blocks in the space of flexible objects consists of **re-configurable** parts. Re-configurable components are characterized by hardware that can change the logic function it implements by changing the interconnection pattern among basic building blocks and the function implemented by the basic building blocks themselves. For some re-configurable components this change can be done only once at design time (for example, FPGAs with anti-fuses and EPROMs), for some, it can be done many times even at run-time (FPGAs with RAM-like cells, EEPROMs, Flash EEPROMs). There are re-configurable components where the basic building blocks have large granularity and re-configuration is achieved by changing appropriate parameters in memory. An example is the run-time re-configurability of microprocessor-based systems where peripherals can be connected with different patterns according to the application. Re-configurable components have been used for years not only for fast-prototyping but also for final products in the initial stages of their market introduction. Because of their superior performance in terms of speed and power consumption with respect to instruction-set machines, re-configurable components are a valid choice for several systems. Research has been carried out to identify novel re-configurable architectures that are more flexible and more performing than the present choices [3].

2.2.2. Design Re-use

Design re-use can be achieved by sharing components among different products (*re-use in space*) and across different product generations (*re-use in time).*

Re-use-in-time has to bridge the technology gap between two (or even more) generations. Re-use-in-space is based on commonality among different products. It depends on the capability of designers to partition their designs so that similarities among different products are factored as common terms. Both re-use and early error detection imply that the design activity must be defined rigorously so that all phases are clearly identified and appropriate checks are enforced. To be effective, a design methodology that addresses complex systems has to start at high levels of abstraction. In several system and IC companies, designers are used to working at levels of abstraction that are too close to implementation so that sharing design components and verifying before prototypes are built is nearly impossible. Design methodologies that address this problem emerged recently and design tools and environments can be put in place to help supporting this design methodology (see the Polis and VCC Felix design systems and the methodology they support) [2][6].

Design re-use is most effective in reducing NRE costs and development time when the components to be shared are close to final implementation. For hardware components, re-use at the implementation level means re-use of physical components avoiding new mask design and production. On the other hand, it is not always possible or desirable to share designs at this level since minimal variations in specification may result in different, albeit similar, implementations.

The ultimate goal is to create a library of functions and of hardware and software implementations that can be used for all new designs of a company. It is important to have a multi-level library since it is often the case that the lower levels that are closer to the physical implementation change due to the advances in technology while the higher levels tend to be stable across product versions.

Design re-use is desirable both at hardware and software level:

- By fixing the hardware architecture, the customization effort is entirely at the software level. This solution has obvious advantages in terms of design cycles since hardware development and production cycles are indeed longer than their software counterpart.
- Basic software such as RTOS and device drivers can be easily shared across multiple applications *if they are written following the appropriate methodology.* This is true in particular for device drivers as will be discussed in details later.
- Application software can be re-used if a set of conditions are satisfied:
    - Each function is decomposed in parts with the goal of identifying components that are common across different products. In the best case, the entire functionality can be shared. More often, sub-components will be shared. In this case, the decomposition process is essential to maximize design re-use. The trade-off here is the granularity of the components versus the sharing potential. The smaller the parts are, the easier is to share them but the smaller is the gain. In addition, sharing does not come for free. Encapsulation is necessary for both hardware and software to prevent undesired side effects. Encapsulation techniques have been extensively used in large software designs (object

orientation) but only recently have caught the attention of embedded system designers [7].

- "High-level" languages (e.g., C, C++, Java) are easily re-targetable to different processors. However, standard compilers and interpreters for these languages are not efficient enough for the tight requirements that system designers must satisfy. Hence they often exploit the micro-architecture of the processor to save execution time and memory, thus making the re-usability of their software across different microprocessors almost impossible. If super-optimized object code generation with the execution time versus memory occupation trade-off made visible to the designers were available, then re-usability would be finally easy to achieve. Our research groups and others have attacked this very topic to allow embedded system designers to focus on the high-level aspects of their problem (see the code generation techniques of Polis[6], software synthesis of data flow graph [8] and super-compilation for VLIW architecture [9]).

3. Design Methodology

Once the context of the design methodology is set, we can move on to define some important concepts precisely that form the foundation of our approach.

3.1. Function

A system implements a set of functions. A function is an abstract view of the behavior of the system. It is the input/output characterization of the system with respect to its environment. It has no notion of implementation associated to it. For example, *"when the engine of a car starts (input), the display of the number of revolutions per minute of the engine (output)"* is a function, while *"when the engine starts, the display in digital form of the number of revolutions per minute on the LCD panel"* is not a function. In this case, we already decided that the display device is an LCD and that the format of the data is digital. Similarly, "*when the driver moves the direction indicator (input), the display of a sign that the direction indicator is used until it is returned in its base position*" is a function, while "*when the driver moves the direction indicator, the emission of an intermittent sound until it is returned to its base position*" is not a function.

The notion of function depends very much on the level of abstraction at which the design is entered. For example, the decision whether to use sound or some other visual indication about the direction indicator may not be a free parameter of the design. Consequently, the second description of the example is indeed a function since the specification is in terms of sound. However, even in this case, it is important to realize that there is a higher level of abstraction where the decision about the type of signal is made. This may uncover new designs that were not even considered because of the entry level of the design. Our point is that no design decision should ever be made implicitly and that capturing the design at higher levels of abstraction yields better designs in the end.

The functions to be included in a product may be left to the decision of the designer or may be imposed by the customer. If there are design decisions involved, then the decisions are grouped in a design phase called *function* (or sometimes feature) *design*. The decisions may be limited or range quite widely.

3.2. Architecture

An architecture is a set of components, either abstract or with a physical dimension, that is used to implement a function. For example, an LCD, a physical component of an architecture, can be used to display the number of revolutions per minute of an automotive engine. In this case, the component has a concrete, physical representation. In other cases it may have a more abstract representation. In general, a component is an element with specified interfaces and explicit context dependency. The architecture determines the final hardware implementation and hence it is strictly related to the concept of platform.

The most important architecture for the majority of embedded designs consists of microprocessors, peripherals, dedicated logic blocks and memories. For some products, the architecture is completely or in part fixed. In the case of automotive body electronics, the actual placement of the electronic components inside the body of the car and their interconnections is kept mostly fixed, while the single components, i.e., the processors, may vary to a certain extent. A fixed architecture simplifies the design problem a great deal but limits design optimality. The trade-off is not easy to achieve.

We call an ***architecture platform***, a fixed set of components with some degrees of variability in the performance or other parameters of one or more of its components.

3.3. Mapping

The essential design step that allows moving down the levels of the design flow is the mapping process, where the functions to be implemented are assigned (mapped) to the components of the architecture. For example, the computations needed to display a set of signals may all be mapped to the same processor or to two different components of the architecture (e.g., a microprocessor and a DSP). The mapping process determines the performance and the cost of the design. To measure exactly the performance of the design and its cost in terms of used resources, it is often necessary to complete the design, leading to a number of time consuming design cycles. This is a motivation for using a more rigorous design methodology. When the mapping step is carried out, our choice is dictated by *estimates* of the performance of the implementation of that function (or part of it) onto the architecture component. Estimates can be provided either by the manufacturers of the components (e.g., IC manufacturers) or by system designers. Designers use their experience and some analysis to develop estimation models that can be easily evaluated to allow for fast design exploration and yet are accurate enough to choose a good architecture. Given the importance of this step in any application domain, automated tools and environments should support effectively the mapping of functions to architectures [2][6].

The mapping process is best carried out interactively in the design environment. The output of the process is either:

- a mapped architecture iteratively refined towards the final implementation with a set of constraints on each mapped component (derived from the top-level design constraints) or
    - a set of diagnostics to the architecture and function selection phase in case the estimation process signals that design constraints may not be met with the present architecture and function set. In this case, if possible, an alternative architecture is selected. Otherwise, we have to work in the function space by either reducing the number of functions to be supported or their demands in terms of performance.

3.4. Link to Implementation

We enter this phase once the mapped architecture has been estimated as capable of meeting the design constraints. We now have the problem of implementing the components of the architecture. This requires the development of an appropriate hardware block or of the software needed to make the programmable components perform the appropriate computations. This step brings the design to the final implementation stage. The hardware block may be found in an existing library or may need a special purpose implementation as dedicated logic. In this case, it may be further decomposed into sub-blocks until either we find what we need in a library or we decide to implement it by "custom" design. The software components may exist already in an appropriate library or may need further decomposition into a set of sub-components, thus exposing what we call the *fractal nature of design, i.e.*, the design problem repeats itself at every level of the design hierarchy into a sequence of nested function-architecture-mapping processes.

4. Hardware Platforms

Product features can be used as a metric for measuring the complexity of the functionality of a product. Figure 1 shows six different products in the same product family ranked according to product features and lined up in time of introduction in the market place. We can group these products into two classes: high-end (A,B,C) and low-end (D,E,F), according to the number of features and their overall complexity. If we use the same architecture for both classes, then the hardware part of the design is fully re-used while software may need to be partially customized. If indeed the high-end class strictly contains the features (functions) of the other, then software design can also be in large part re-used. Thus even if unit cost were higher than needed for the low-class products, design costs and NREs may push towards this solution. We have witnessed this move towards reducing product variants in a number of occasions both in system and IC companies. The common architecture can then be identified as a platform. We feel that, given its great importance, a definition of hardware platform is needed since different interpretations have been used of this commonly used term.

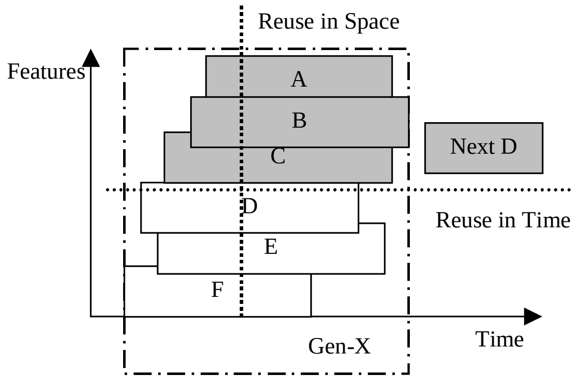

Figure 1: Product Space

As we have discussed earlier, a hardware platform is a rather warm and fuzzy concept that is related to a common architecture of sort. We believe that it is possible to generalize the basic idea to encompass not only a fully specified architecture but it is also *a family of architectures that share some common feature.* Hence we prefer to identify a platform with the set of constraints that can be used to test whether a given architecture belongs to the family.

A ***Hardware Platform*** is *a family of architectures that satisfy a set of architectural constraints* that are imposed to allow the re-use of hardware and software components.

The "strength" of hardware and software architectural constraints defines the level and the degree of re-use. The stronger the architectural constraints the more component re-use can be obtained. On the other hand, stronger constraints imply also fewer architectures to choose from and, consequently, less application-dependent potential optimization.

The Personal Computer (PC) [5] platform is a good example of tight hardware and software constraints that enable design re-use (both in space and in time). The essential constraints that determine the PC hardware platforms are:

- the x86 instruction set architecture (ISA) that makes it possible to re-use the operating system and the software application at the binary level[1];
- a fully specified set of busses (ISA, USB, PCI) that make it possible to use the same expansion boards or IC's for different products[2];
- legacy support for the ISA interrupt controller that handles the basic interaction between software and hardware.
- a full specification of a set of I/O devices, such as keyboard, mouse, audio and video devices.

All PCs satisfy this set of constraints. The degrees of freedom for a PC maker are quite limited! However, without these constraints the PC will not be where it is today. One wonders why this success has not percolated in the embedded system domain! This question will be answered in the following paragraphs.

An embedded system hardware platform is a combination of three main sub-systems[3]: processing unit, memory, and I/O.

1. Strong cost and packaging requirements impose keeping the hardware components at minimum complexity and size, thus making the adoption of the platform concept more difficult.
2. The interaction with the environment is frequently very complex and application dependent. Hence, the I/O subsystem is an integral and essential part of the architecture as it cannot be assumed to be equal across the application space as is the case for the PC domain. Consequently, the architecture itself is application specific and not re-usable across different application domains. Even for the same application domain, embedded systems can differ very much in their I/O sub-systems. In the car dashboard example, low-end and high-end products have completely different I/O requirements: the latter use very complex display and communication devices while the former use simple display and serial communication.

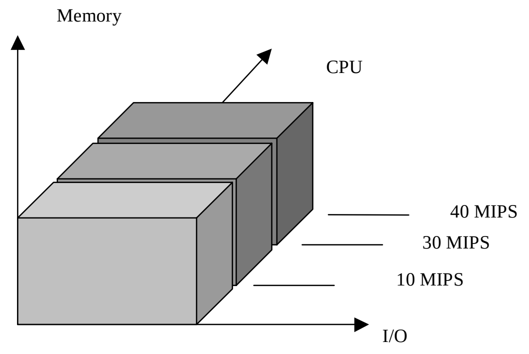

Figure 2: Example of platform space

Figure 2 shows different versions of an embedded system platform parameterized with respect to:

- processing unit performance;
- speed and footprint for the *memory system* (RAM, ROM, Flash, EEPROM...);
- analog channels, timing and digital channels, communication speed for *the I/O sub-system*.

Note that the hardware platform is a family of possible products fully identified by the set of constraints. When we define a product, we need to fully specify its components.

Given a hardware platform, we define a hardware ***platform instance*** as a particular architecture in the hardware platform where all components are fully specified.

Figure 3 shows six products requirements covered by two different instances of a unique hardware platform: high and low end. The I/O dimension is a projection of a very complex space where characteristics such as number of digital I/Os, number of PWM channels, number of A/D converters, number of input captures and output

[1] In fact, the MS-DOS operating system can be run on any compatible x86 microprocessor.

[2] Note that expansion board re-usability is limited by the technology used.

[3] Mainly for a single processor system

compares, and specific I/O hardware drivers are represented.

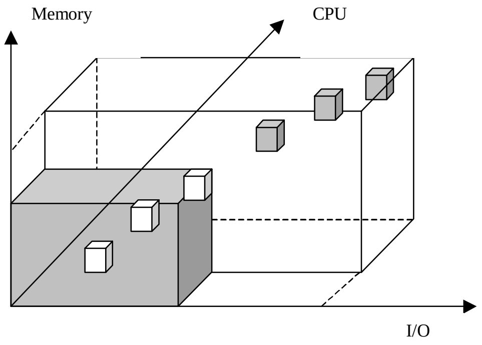

Figure 3: Platforms for different product classes

The two hardware platform instances are also shown in the features/time space (Figure 4). The diversification of hardware architectures should be carefully handled since it reduces the production volume of each diversification.

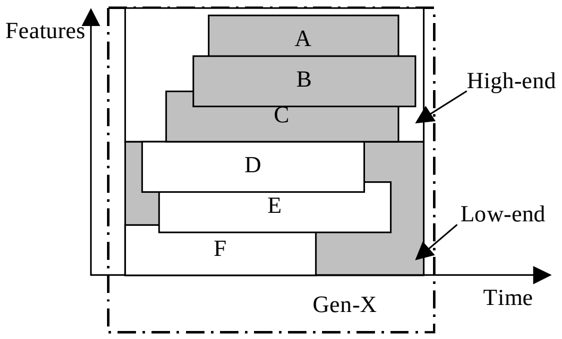

Figure 4: Platforms in the feature/time space

Seen from the application domain, the constraints that determine the hardware platform are often given in terms of performance and "size". For the dashboard example, we require that, to sustain a set of functions, a CPU should be able to run at least at a given speed and the memory system should be of at least a given number of bytes. Since each product is characterized by a different set of functions, the constraints identify different hardware platforms where more complex applications yield stronger architectural constraints. Coming from the hardware space, production and design costs imply adding hardware platform constraints and consequently reducing the number of choices. The intersection of the two sets of constraints defines the hardware platforms that can be used for the final product. Note that, as a result of this process, we may have a hardware platform instance that is over-designed for a given product, that is, some of the power of the architecture is not used to implement the functionality of that product. Over-design is very common for the PC platform. In several applications, the over-designed architecture has been a perfect vehicle to deliver new software products and extend the application space. We believe that some degree of over-design will be soon accepted in the embedded system community to improve design costs and time-to-market. Hence, the "design" of a hardware platform is the result of a trade-off in a complex space that includes:

- The size of the application space that can be supported by the architectures belonging to the hardware platform. This represents the flexibility of the hardware platform;
- The size of the architecture space that satisfies the constraints embodied in the hardware platform definition. This represents the degrees of freedom that architecture providers have in designing their hardware platform instances.

Once a hardware platform has been selected, then the design process consists of exploring the remaining design space with the constraints set by the hardware platform. These constraints can not only be on the components themselves but also on their communication mechanism. When we march towards implementation by selecting components that satisfy the architectural constraints defining a platform, we perform a successive refinement process where details are added in a disciplined way to produce a hardware platform instance.

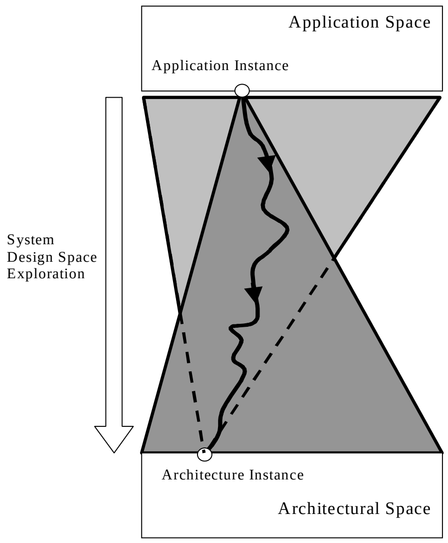

Figure 5 Architectural space implementing the application instance (at different cost).

Ideally the design process in this framework starts with the determination of the set of constraints that defines the hardware platform for a given application. In the case of a dashboard, we advocate to start the design process before splitting the market into high-end and low-end products.

The platform thus identified can then be refined towards implementation by adding the missing information about components and communication schemes. If indeed we keep the platform unique at all levels, we may find that the cost for the low-end market is too high. At this point then we may decide to introduce two platform instances differentiated in terms of peripherals, memory size and CPU power for the two market segments. On the other hand, by defining the necessary constraints in view of our approach, we may find that a platform exists that covers both the low-end and the high-end market with great design cost and time-to-market improvements.

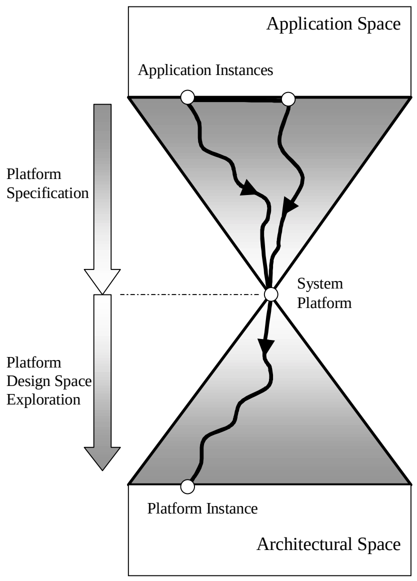

Figure 6. The system platform concept forces the design exploration to find somewhere in the abstraction a common solution.

Hardware platform-based design optimizes globally the various design parameters including, as a measure of optimality, NRE costs in both production and design. Hardware platform-based design is neither a top-down nor a bottom-up design methodology. Rather, it is a "meet-in-the-middle" approach. In a pure top-down design process, application specification is the starting point for the design process. The sequence of design decisions drives the designer toward a solution that minimizes the cost of the architecture. Figure 5 shows the single application approach, the bottom of the figure shows the set of architectures that could implement that application. The design process, the black path, selects the most attractive solution as defined by a cost function. In a bottom-up approach, a given architecture (instance of the architectural space) is designed to support a set of different applications that are often vaguely defined and is in general much based on designer intuition and marketing inputs.

- An essential feature to achieve re-usability of a hardware platform is that the application designer sees it as a unique object in his design space,. For example, in the PC world, the application software designer sees no difference among the different PCs that are satisfying the PC platform architectural constraints.

Figure 6 shows graphically the features of an hardware platform. The cone above the point represent its flexibility, the cone below represents its generality. In general, hardware platforms tend to have a large cone above and a small cone below to witness how important design re-use and standards are versus the potential optimization offered by loose constraints.

5. Software Platform

The concept of hardware platform by itself is not enough to achieve the level of application software re-use we are looking for. To be useful, the hardware platform has to be abstracted at a level where the application software "sees" a high-level interface to the hardware that we call **Application Program Interface or API**. There is a software layer that is used to perform this abstraction (Figure 7). This layer wraps the essential parts of the hardware platform:

- 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[4].

This layer is called the **software platform[5].**

In our conceptual framework, the programming language is the abstraction of the ISA, while the API is the abstraction of a multiplicity of computational resources (concurrency model provided by the RTOS) and available peripherals (Device Drivers). [6] There are different efforts that try to standardize the API [12], [13], [14]. In our framework, the API is a unique abstract representation of the hardware platform. With an API so defined, the application software can be re-used for every platform instance.

[4] In some cases, the entire software layer, including the Device Drivers and the network communication subsystem is called RTOS.

[5] This layer embeds also the middle-ware.

[6] There are several languages that abstract or embed directly the concurrency model avoiding the RTOS abstraction [10],[11].

Of course, the higher the abstraction layer 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.

5.1. RTOS

In our framework, the RTOS is responsible for the scheduling of the available computing resources and of the communication between them and the memory subsystem. Note that in most of the embedded system applications, the available computing resources consist of a single microprocessor. However, in general, we can imagine a multiple core hardware platform where the RTOS schedules software processes across different computing engines. There is a battle taking place in this domain to establish a standard RTOS for embedded applications. For example, traditional embedded software vendors such as ISI and WindRiver are now competing with Microsoft that is trying to enter this domain by offering Windows CE, a stripped down version of the API of its Windows operating system. In our opinion, if the conceptual framework we offer here is accepted, **the precise definition of the hardware platform and of the API should allow to synthesize automatically and in an optimal way most of the software layer**, a radical departure from the standard models borrowed from the PC world. **Software re-use, i.e. *platform re-targetability*, can be extended to these layers (middle-ware) hopefully resulting more effective than binary compatibility.**

5.2. Device Drivers

The "virtualization" of input and output devices has been the golden rule to define generic and object-oriented interfaces to the *real word*. The I/O subsystem is the most important differentiator between PCs and embedded systems. In our framework, the communication between software processes and the environment, i.e. *physical processes*, is refined (input/output refinement) through different software and hardware layers (Figure 8). The device drivers identify the software layers. The layer structure is introduced to achieve reusability of components belonging to each layer.

In this scheme, the acquisition of a physical variable is seen as a communication between a sender, that "lives" in the physical domain, and a receiver, that "lives" in the *virtual* domain. At the application level, the acquired value is an image, i.e. a representation with a certain precision and time validity, of the physical variable correctly sampled and quantized. No details about the sensor, analog-to-digital converter and other hardware components are known. Using the analogy with protocols, to implement this communication three main layers are identified: the presentation, transport and physical layer. For the sensor, i.e. the sender, the presentation and transport layers serve as a "transducer" of the physical variable to an electrical variable, which is transported, with wires, to the hardware of the system. I/O communication is asymmetric, i.e. the communication behavior is typically *sampling and quantization* for the link from the environment to the system and *hold* for the opposite direction.

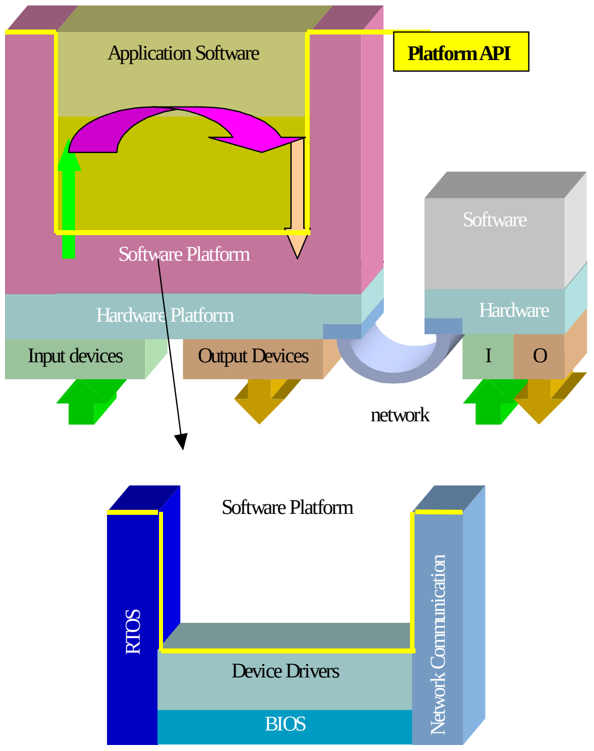

Figure 7: Layered software structure

The hardware channels are abstracted via a software layer that "lives" underneath the device driver: the Basic I/O system or BIOS. This software layer must guarantee the reliability of the electrical link by continuously monitoring the link and its quality. The BIOS represents the transport layer that typically can support different types of sensors/actuators satisfying bandwidth and accuracy of the I/O *channel*. Thus the layering of the device driver offers another opportunity of design re-use.

For the receiver, i.e. the application process, the presentation layer masks the operations required to extract the image of the physical variable out of the electrical information provided by the transport layer. Typical operations are filtering, diagnosis and recovery of the sensor signal and sensor calibration.

In this communication abstraction, transducers implement the translation of the physical variable towards its electrical representation and device drivers represent its abstraction towards the application software with no attention towards the transport mechanism that is left to the BIOS.

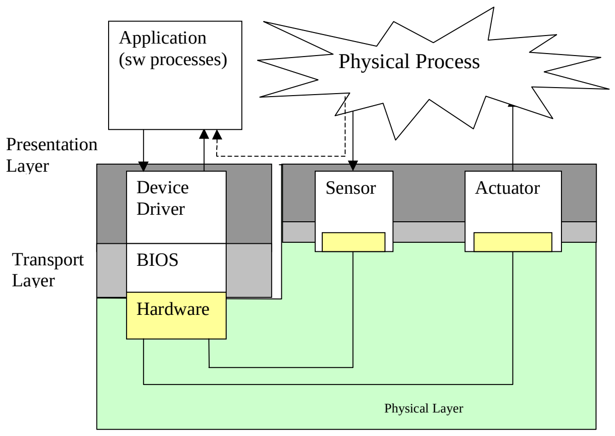

Figure 8: IO subsystem as a communication link

5.3. Network Communication

The network communication subsystem supports the connection between software processes running on different electronic systems. Information about the status of remote parts of the system is typically exchanged. In a car, in-vehicle communication is required to continuously update the status of physical variables acquired by a different electronic control unit, such as car speed, battery voltage or brake pedal status. In these cases, the network subsystem is the mean to "acquiring" a physical variable from remote systems but no difference is seen from the application software making re-use really effective.

6. System Platform

The combination of hardware and software platforms is called **system platform.** It is the system platform that completely determines the degree of re-usability of the application software and identifies the hardware platform. In fact, the hardware platform may be deeply affected by the software platform as in the case of the PC platform. In fact, the re-usability of the application software made possible by the PC system platform comes at the expense of a very complex layered structure of the Windows O.S. families and of the size and complexity of the microprocessor cores supporting the instruction set. In this domain, this price is well worth being paid.

In the automotive domain, the use of the OSEK/VDX RTOS [4] specifications, sponsored by the German automotive manufacturers, was conceived as a tool for re-usability and "flexibility" using the PC platform as a model. However, the peculiarities of embedded systems where cost, reliability and geometric constraints play a fundamental role, make this approach much more difficult to use. The OSEK initiative intends to set a standard for operating systems to be used in the car so that software produced by different vendors could be easily integrated. OSEK constrains the RTOS and part of the network communication system, but does not constrain any input/output system interface exposing the hardware details of the I/O subsystem to the software. Because in the OSEK specification a device driver API is not defined, this specification is not an architectural constraint strong enough to enable alone the re-use of software components even at source code level, letting each system maker define its own interface to interact with the attached devices. If we add to the OSEK standard for RTOS and Network management, the definition of the I/O sub-systems and the use of a single ISA, even binary software components can be re-used. On the other hand, OSEK fully specifies the scheduling algorithm for the operating system. It does guarantee that no priority inversion occurs but it does not help in a substantial way to make the verification of the timing properties of the application any easier. In our opinion, this is an over-specification with respect to software re-usability since, even if the scheduling algorithm is not fully specified, we can still re-use software at the source level. The jury is still out to determine whether the overall OSEK constraints are so strong that economic and efficient solutions are eliminated a priori. We doubt that this specification in its present form is going to yield the desired results just because of the difficult trade-off space that embedded applications entail.

7. Conclusions

In this paper we presented a conceptual view of platform-based design. This view encompasses hardware and software platforms for embedded systems. It is based on semi-formal definitions of the various components. It allows the use of design methodologies that favor global optimization across multiple objectives, from production cost to design cost and time. We have used this framework to place approaches to platform-based design in the PC world and in the automotive world in context. We believe that platforms will dominate system and IC design in the Deep Sub-Micron era. We believe that much remains to be done to extend these concepts and use them in different application domains.

8. Acknowledgments

We thank Antonino Damiano, Nicola Staiano (Magneti Marelli), Luciano Lavagno (Cadence Berkeley Lab) and Marco Antoniotti (PARADES) for their insights and suggestions.

9. References

[1] G. Martin and H. Chang, et al, Surviving the SOC Revolution: A Guide to Platform Based Design, Kluwer Academic Publishers, September 1999.

[2] G. Martin and B. Salefski. Methodology and Technology for Design of Communications and Multimedia Products via System-Level IP Integration, In *Proceedings of the Design, Automation and Test in Europe*. February 1998.

[3] John R. Hauser and John Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of the IEEE Symposium on Field Programmable Gate Arrays for Custom Computing Machines*, pages 12-21. April 1997.

[4] OSEK/VDX, Operating System v. 2.0r1, (http://www-iiit.etec.uni-karlsruhe.de/~osek)

[5] Intel Corp. and Microsoft Corp. PC 99 System Design Guide. (http://www.pcdesguide.org)

[6] F. Balarin et al, "Hardware-Software Co-Design of Embedded Systems: the Polis approach", Kluwer, 1997

[7] B. Selic, Using UML for modeling complex real time systems. In *Proceedings of ACM Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 250-60, 19-20 June 1998

[8] J.L. Pino, S. Ha, E.A. Lee, J.T Buck, Software synthesis for DSP using Ptolemy, *Journal of VLSI Signal Processing*, Jan. 1995, vol.9, (no.1-2):7-21

[9] P. Faraboschi, G. Desoli, J.A. Fisher, The latest word in digital and media processing, *IEEE Signal Processing Magazine*, March 1998, vol.15, (no.2):59-85.

[10] International Organization for Standardization. Information technology – Programming Languages – Ada. *Ada Reference Manual. 1995*

[11] G. Berry. The Foundations of Esterel. http://www.inria.fr/meije/Esterel. 1998

[12] POSIX: Portable Operating System Interface for UNIX. The 1003.0/1003.1b IEEE POSIX Standard. See also *The ISO JTC1/SC22/WG15 POSIX and the IEEE's Portable Application Standards Committee web sites* (http://anubis.dkuug.dk/JTC1/SC22/ WG15 and http://www.pasc.org).

[13] K. Arnold, J. Gosling. The Java Programming Language. *Addison-Wesley, ISBN 0-201-31006-6*

[14] K. Arnold et al, The JINI Specification, *Addison-Wesley, ISBN 0-201-61634-3* (see also http://www.sun.com/jini/specs).