

# Scenario-Oriented Design for Single Chip Heterogeneous Multiprocessors

JoAnn M. Paul  
Electrical and Computer Engineering Department  
Carnegie Mellon University  
Pittsburgh, PA 15213 USA  
jpaul@ece.cmu.edu

## Abstract

*The design of next generation single chip computers may require the selection, programming and coordination of tens to hundreds of individual processing elements (PEs) of several or tens of heterogeneous types. This paper outlines differences in the design process for these next generation single chip systems from that of traditional designs. Then it focuses on a novel design strategy, Scenario-Oriented Design (SOD). SOD is just one design strategy that arises because of the potential of considering applications, schedulers, and hardware as they interact to form a system instead of viewing them as separate design domains. By leveraging one against the other, the system and its design process can be optimized in new ways. This is made possible by reducing the modeling detail of each design domain within a system in high level simulation.*

## 1 Introduction

The rich history of digital computation has been enabled by three categories of support for the design process: models, tools and strategies. Ideally these three are strongly related. A model provides a basic view of cause and effect that allows for coarse level prediction as model elements are manipulated at finer levels by tools according to design strategies. For example, the models of a finite state machine (FSM) are implemented and simulated in a wide variety of tools according to widely accepted design strategies — systematic narrowing of a complex design space towards convergence on good designs. Ideally, models, tools and strategies are developed together according to a widely accepted and even obvious definition of what the key characteristics of the design space are and how they should be conceived. However, in practice, some experience typically needs to be first obtained using inelegant models, tools and strategies.

For single chip computers, we are at the point where existing models, tools and strategies are failing to permit designers to efficiently capture the design space at hand. Hardware Description Languages (HDLs) and their associated simulators and synthesis tools do not capture software as part of the system model. Instruction Set Simulators are too detailed and slow to capture systems with many processors. System-level specification tools such as UML, Statemate, SystemC, Ptolemy and Simulink (Matlab), do not capture the effects of software as it executes on hardware at all. Currently, designers are left to their own devices, which limits the effective realization of many potentially significant designs.

Basic research in the development of next generation models, tools and strategies for both the software and

hardware of single chip heterogeneous multiprocessors is required. For solutions used by many designers, elegance matters — the models and strategies must resonate with more effective means of manipulating the problems at hand.

Within the next five years, it will be possible to provide enough transistors on a single chip for on the order of a hundred ARM-equivalent processors. At that point, individual, programmable processors will be like registers were to early VLSI design — building blocks within a larger organizing framework. And yet those same processor building blocks will be differentiated from each other in the capabilities of the hardware in the processors, the way they are programmed, and their manner of interconnection.

A subset of applications that might appear on such future devices includes that of current cell phones, current personal digital assistant (PDA) applications, global positioning system (GPS) sensing, Bluetooth, motion sensing, ad hoc networking, 3-D image processing, compression/decompression, security, multimedia and a broad set of human computer interaction (HCI) software. Ubiquitous and pervasive computing will produce newer scenarios with even more complex functionality. But these possibilities are limited by the ability to design and technically realize these scenarios in the individual space and power-constrained computing devices with a practical amount of design effort.

The stage is set for a new generation of computing. For this as for previous generations, what enables the next design level is the ability to maximize the ratio DQ/DE, where DQ = Design Quality and DE = Development Effort for a given design. DQ can include many performance factors, such as overall speed of execution, power, size, extensibility and correctness. Some of the quality metrics such as functional correctness are absolute — must be met for a design to be useful at all — while others such as power consumption may be more relative, creating a vast design space of design trade-offs. Development Effort can be similarly difficult to quantify, since it can include such issues as re-use which has been difficult to measure for even pure software designs [1].

The potential of the next generation of technology cannot be reaped without a common basis for design, at a modeling level where the design decisions with the most impact can be effectively manipulated.

## 2 PHMs

In order to meet performance, power and space requirements while retaining the flexibility of programmable devices, single chip designs will take on the appearance of heterogeneous multiprocessors. We refer to

this class of devices as Programmable Heterogeneous Multiprocessors (PHMs), implying that not only will the individual processing elements (PEs) on the chip be programmable, but collections of PEs must be considered programmable, along with the chip as a whole. The chip must be viewed as a programmable collection of processors where groups of dynamically selected processors can be programmed (i.e., scheduled) as needed to execute applications. But this class of computing poses distinctly different design challenges from board and network-level heterogeneous multiprocessors for three primary reasons:

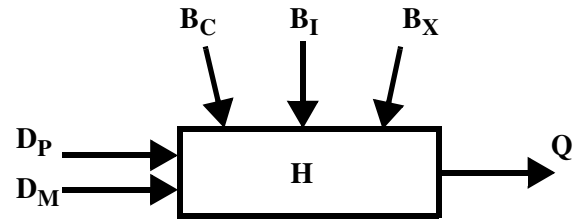
- A single chip is a finite resource. A single chip forces collections of processing elements to be considered finite and fixed, unlike wide-area networks for which the system resources are flexible, dynamic, hidden by layers of services, and possibly even unknown. Single chip systems will include a distinct, hardware specification phase, even if the hardware is used for a wide variety of applications.
- The design will be semi-custom. The underlying hardware will be more customized to the application space than for traditional programmable systems. Traditional heterogeneous multiprocessors are based upon providing transaction-like services on a diverse collection of resources, and single chip devices such as Application Specific Integrated Circuits (ASICs) and Systems on Chips (SoCs) are customized to meet fixed latency requirements as a reactive system. PHMs will be semi-custom and have aspects of both design styles.
- Coordination of system resources will be required. The large differential in on-chip vs. off-chip communications will force efficient utilization and management of on-chip system resources — including processing elements, memory, communications bandwidth and chip I/O. In contrast to the emphasis on the distributed and extensible nature of wide-area networks, close coordination of the on-chip resources will be required.

### 3 The Design Space

A high-level view of a design environment that addresses the challenges of single chip PHM design is shown in Figure 1. The specification of the Single Chip Heterogeneous Multiprocessor, H, is shown in the middle of the diagram. This specification is of the key high-level features of the hardware chip design, including the numbers and types of processing elements, their manner of interconnect (such as bus or network-on-chip [2] bandwidth and protocol configurations), firmware and hardware configuration. The result is a view of the chip that can be evaluated against a variety of programming and use scenarios. However, it is not at a detailed ISA level.

The heterogeneous multiprocessor, H, has inputs coming from the left and the top of the diagram. From the left are two classes of system data inputs. The first,  $D_P$  are time-stamped system inputs that are conceptually presented to the system hardware on I/O pins. They are test vectors that would be presented to a pure hardware design or traditional time-constrained embedded system design. The second,

$D_M$ , are data values that reside in some external memory, possibly in a remote computer system, that H may access when it has the available computation resources to process them. This class of inputs is analogous to jobs, packets or other requests in a queue waiting to be processed by H. By separating these classes of inputs, we are able to separate the timing properties of the system that are driven by external demand from those that are driven by internal processing capability of the system under design.



**Figure 1 Single Chip Heterogeneous Multiprocessor Quality Evaluation**

From the top of the diagram, three other classes of inputs are shown. These are not data inputs, but programs. The first class of programs are clocked benchmarks,  $B_C$ . These are programs or tasks with fixed latency requirements with required latency specified to a fixed time reference. Like a software benchmark, they represent a class of representative functionality that the system may carry out. But unlike a software benchmark, the programs (or program fragments) in  $B_C$  are characterized by having fixed performance requirements, determined by physical time reference value. Because of this fixed performance requirement for the individual programs in  $B_C$ , additional processing capacity is wasted — these benchmarks are designed to meet the worst-case demands that are presented to the system by  $D_P$ . This is typical of many embedded designs and virtually all pure hardware designs.

The second class of programmatic inputs are internally timed,  $B_I$ . This is a class of benchmarks for which performance is calculated not in advance, but by the internal timing of the processing capabilities of the design. This is a more conventional class of software benchmarks, where performance, defined as outputs produced over time is discovered as a result of the execution of the functionality on the architecture. In the case of a programmable heterogeneous multiprocessor this can be over many processors in a design. The final class of inputs at the top are schedulers,  $B_X$ . Unlike other benchmarks, schedulers do not represent the potential net functionality of the design, but rather act as a means of resolving the other benchmarks to the architecture. And yet, they are an important class of representative behavior that affects the overall system behavior.

Finally, the design environment has a single output  $Q$ , the overall quality metric of the design. While shown as a single output,  $Q$  represents a tuple of values that include performance of the system for the two classes of behaviors implied by the benchmarks  $B_C$  and  $B_I$  (performance is used here in the general case that includes speed, power, etc.).

In general,  $Q$  is the set of data that comes from exercising the system  $H$  under the data values and loading scenarios indicated by the  $D$  inputs of the system when it is programmed according to the set of  $B$  benchmarks. Significantly, the general form of the relationships of the design environment  $E = \{D, B, H, Q\}$  must be more general and at a higher level than previous design environments. For example, if the environment is limited to  $E = \{B_C, D_P, Q\}$  where  $B_C$  is the end-functionality of the design (rather than a representative set of functionality), the result is an executable specification. This is typical of Hardware Description Languages (HDLs) and even high-level behavioral modeling environments such as System-C. The only Quality metric for such environments is pass/fail — the system either meets the required behavior to process the  $D_P$  inputs in the time as specified or it does not. Thus,  $H$  is considered to be fully specified by  $D_P$  and  $B_C$  and not a separately performance-evaluated architecture at all. If the environment is limited to  $E = \{B_C, B_X, D_P, H, Q\}$  where  $H$  is a single processor, then the focus is once again on pass/fail, but the question is whether the system is schedulable — this is the kind of analytical modeling typical of research in real-time operating systems (RTOSs). Finally, if the environment is limited to  $E = \{B_I, D_M, H, Q\}$  where  $H$  is a single processor executing at the instruction set simulator level or below and  $B_I$  is the SPEC benchmarks, then it is typical of simulators such as SimpleScalar used to model a micro architecture or ISA.

Consider the complexity of the application space represented by the interaction of  $B$ ,  $H$ , and  $D$ . Clearly, the current-day approach, instruction-set level simulation, cannot permit effective exploration of the design space because of the sheer level of detail required in the model and the time it takes to generate any single value of  $Q$ .

#### 4 Scenario-Oriented Design

As each new level of design is enabled by appropriate models and the means to manipulate and evaluate them through simulation, new organizing principles, or design strategies emerge. Scenario Oriented Design (SOD) is one such novel design strategy. A scenario orients heterogeneous multiprocessor single chip design according to a blend of performance requirements, implemented in new chip-wide programmer's views. SOD leverages increased heterogeneity in the future application space so that greater efficiency in both the design process and the resource utilization may be enabled.

In order to meet latency requirements of systems with type  $D_P$  inputs with fixed performance (FP) response times met by the  $B_C$  class benchmarks, current systems must be overdesigned. This overdesign occurs for two reasons. First, the time it takes to match functionality to available processing power is prohibitive. Since worst case (WC) behavior must be met, the capacity of system resources is wasted unless the available processing power exactly matches the computational demand (as it does in pure

hardware designs). Second, irregular loading situations and data dependent processing times also contribute to under-utilized processing resources except in peak loading situations where the data values in the system result in the worst case execution times. For example, MPEG4 decoding will have as much as an order of magnitude differential in processing time or more, depending upon the data values in portions of an image. And yet the system must be designed for the worst case loading and data processing times. The result of each is an inefficient design process and/or under-utilized system resources.

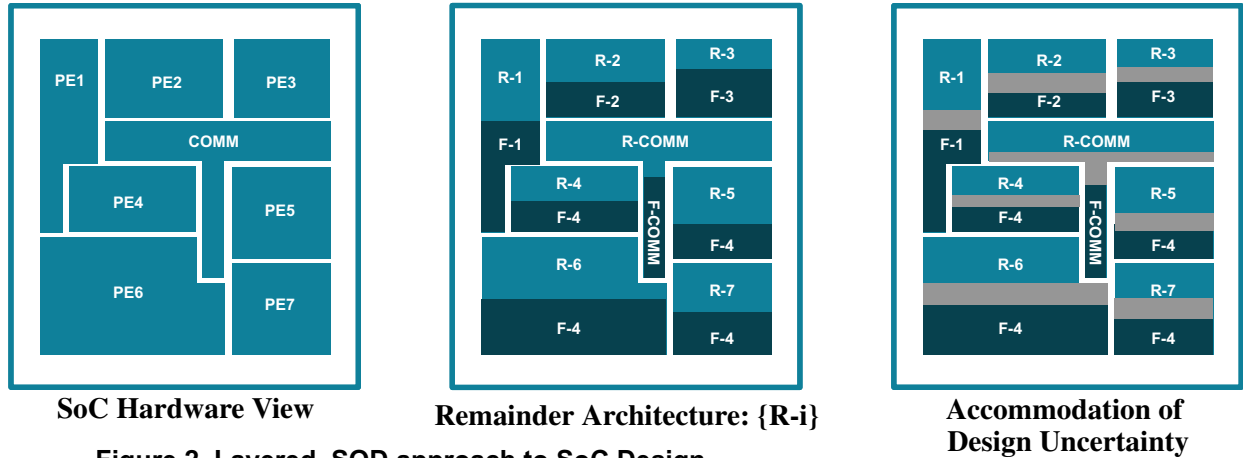
By contrast, software benchmarks, the  $B_I$  in Figure 1, are designed to be a broad, representative set of program types used to evaluate and optimize a programmable device's throughput performance (TP) over a set of applications,  $B_I$ , by processing a representative set of available data values averaged over a period of time. Instead of ensuring that WC behaviors are met, designers that seek to optimize throughput optimize for the common case (CC), where rarer cases might be accommodated at reduced performance. For example, network switches are designed to occasionally drop packets that are presumed to be resent by other parts of the design. This permits more optimal performance overall, because network switch buffers can be sized to handle the common case over the rare case far more efficiently. Dropped packets are sent at reduced performance. (The same CC philosophy applies to caches, branch predictors, and even operating system scheduling strategies.)

Consider how computer system design is currently separately optimized toward two separate ends: worst case (WC) response times to physical inputs presented to the system ( $D_P$ ) as processed by behaviors with fixed and bounded, clocked response times ( $B_C$ ), and design oriented to maximize throughput performance (TP) of the system by optimizing for the anticipated, common case (CC) of a class of representative behaviors ( $B_I$ ) processing a class of representative data ( $D_M$ ) limited only by internal processing and data access times. Philosophically, the design strategies are worlds apart.

Significantly, future single chip designs will execute a mix of the  $B_C$  and  $B_I$  behavior classes, as systems will respond to a mix of  $D_P$  and  $D_M$  input classes, resulting in the need to ensure that a set of FP behaviors are met while a broader class of anticipated CC behaviors are optimized.

Currently, for those systems where FP and TP performance oriented design is present, the development process for each type of functionality is separated into different devices where it is incidental that they reside on the same chip. General purpose programming resides on the general purpose processor, while other processors utilize individual RTOSs to ensure WC behaviors are met, or WC behaviors are ensured by implementation in custom hardware.

SOD fully leverages the observation that systems increasingly include FP and TP behavior classes in scenarios that lie beyond simple benchmarking. *Scenarios*



**Figure 2 Layered, SOD approach to SoC Design**

[3] include a mix of  $D_p$  (traffic patterns and data values) and  $D_M$  (data values) input classes and  $B_C$  and  $B_I$  behavior classes. Performance, size, and design time can be simultaneously optimized for a chip that executes a scenario by first considering the mapping of the FP functionality across the entire chip, consuming part of the proposed architecture. The remainder architecture is then further designed to handle a class of applications for which the chip is anticipated to be programmed. The architecture and software on the chip that satisfies the FP behavior is a design layer, which, in turn, provides another layer of chip-wide, programmable design.

Thus, SOD can satisfy performance for FP functionality and provide a basis for a TP-optimized remainder architecture. SOD moves beyond the mere accommodation of both classes of behaviors in a single chip to one that leverages the presence of both classes for the simultaneous optimization of design time and design quality. It removes the burden of measuring exact execution times for FP behaviors at the start of design, and opens up new avenues of designer creativity when both a *hardware architecture* and a *remainder architecture* are co-designed.

Figure 2 illustrates how SOD uses layering of heterogeneous functionality. The figure contains three views of an SoC that has seven processing elements and a single communications channel. On the left is the hardware view, showing seven heterogeneous processing elements (PE1-PE7) of irregular shapes and sizes, artistically implying a variety of processing capabilities. Each PE is adjacent to (connected to) a common communications channel, COMM. The figure is simplified in order to more clearly illustrate the main points.

The middle view of the chip is a software partitioning of the chip, where each processing element has now been subdivided into two parts. Each  $PE_i = \{F-i, R-i\}$  and  $COMM = \{R-COMM, F-COMM\}$ . The hardware view of the chip has not changed. The entire chip has been subdivided with a functional overlay onto a set of processing resources. The functional overlay,  $\{F-i\}$ , is the set of  $B_C$  benchmarks to the processing resources on the chip. After the functional overlay has been mapped, processing power

remains across the chip, forming a *remainder (or virtual) architecture*,  $R = \{R-i, R-COMM\}$ . This is an architecture that can be designed so that the chip can be optimized to also carry out the  $B_I$  behaviors.

The right-hand view illustrates how design uncertainty is accommodated by SOD. Grey boxes are superimposed at the border between the F-i and R-i parts of the design, since they may fall into either the fixed performance or remainder architecture portion of the design. These grey boxes enlarge the boundary between the performance-group partitions and illustrate how SOD can reduce design time in two ways. First, the FP functionality that is mapped to the chip need not be precisely known beforehand — this enables the class of  $B_C$  behaviors to be viewed as benchmarks instead of application specific functionality because latency properties matter more to this phase of design. Second, as the remainder architecture optimizes TP functionality relative to the processing capabilities of the virtual architecture, the  $B_I$  behaviors will adjust to over- and underestimations of  $B_C$  latencies within a certain degree of tolerance.

Previous views of SoCs have taken a side-by-side view of the partitioning of a system [4] where physical insight to design the cost/benefit of resource sharing is essentially non-existent. This modeling does not capture systems where there is considerable system state [5] involved with interactions among the processors of the system. SOD partitioning produces a chip-wide, horizontal view where hardware resources are modeled in the bottom layer, schedulers in the middle layer, and general software at the top layer (these latter two could have multiple internal layers). A scheduler is a software program that is distributed to permit the co-operation of other software programs that conform to its conventions. However, the layering concept is a strategy that leverages schedulers as a basis for a soft partitioning of a hardware design.

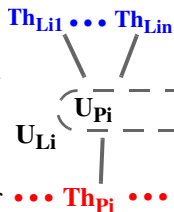
## 5 Simulation Foundation — MESH

The vision of Figure 1 is enabled by simulation that raises the level of design for concurrent, heterogeneous

software executing on concurrent, heterogeneous software above the instruction set simulator level, while defining new design elements. The Modeling Environment for Software and Hardware (MESH) has previously been published, including details of how timing is resolved and the simulator scheduler works [6][7][8]. MESH is unique in providing a layered modeling basis above ISS models and in using schedulers to model concurrent, high level software running on high level models of processor resources. The resolution of timing through design layers where unrestricted software executes on hardware models without relying upon instruction set simulators is what distinguishes MESH. Figure 3 illustrates MESH's layered, logical-on-physical relationship while Table 1 describes the elements in more detail.

A dynamic number of logical threads  $Th_{Li1} \dots Th_{Lin}$  (the software, labeled as  $Th_{Li1} \dots Th_{Lin}$ ) are shown at the top of Figure 3. Their execution is scheduled onto a single resource (a processor, modeled as a physical thread,  $Th_{Pi}$ ) by a scheduler  $U_{Pi}$ . The  $U_{Pi}$  threads are actually models of schedulers in the system that can make scheduling decisions based on the state of the threads being scheduled and other system state. This scheduling resolves the logical events of the software threads to physical timing. Thus the schedulers serve two roles: modeling scheduling decisions, and resolving logical computation to physical time. Figure 3 shows a single vertical slice of a model, a complex system would have many resources ( $Th_{Pi}$ ) along with their schedulers and logical threads.

In MESH, there are two dimensions of scheduling: one based on physical time, and the other on logical state. The resource threads are scheduled to activate based on simulation time; at each timed activation they provide resource power to their scheduler which then selects a logical thread to execute based on its state (e.g., is it waiting for data or not).  $U_{Li}$  threads can also logically group resources for inter-resource scheduling as shown by the dashed oval going off of the figure. This permits M threads to be dynamically mapped to N resources. MESH's execution is approximately 2.5 orders of magnitude faster than an internal ISS level simulator of the same example with roughly equivalent accuracy. It currently includes simple power annotations, permitting evaluation of power-performance trade-offs at the system-level and enabling new system-level design strategies to preserve performance while reducing power consumption



**Figure 3**  
**A Slice of the Layering**

<p><math>Th_{Lij}</math> — One of <math>j</math> logical threads (software) that will execute on processor <math>i</math>.</p> <p><math>Th_{Pi}</math> — A model of the <math>i</math>th physical resource in the system, such as a processor.</p> <p><math>U_{Pi}</math> — A scheduler that selects logical threads intended to execute on resource <math>Th_{Pi}</math>.</p> <p><math>U_{Li}</math> — A logical scheduler that can schedule M threads to N resources. e.g., a pthread scheduler</p>
---

**Table 1 Glossary of Thread Types**

[9]. System-level power-performance trade-offs are yet another motivation for the use of collections of simpler processors executing at lower frequencies in lieu of complex processors that multiplex many tasks.

## 6 Conclusions and Future Directions

Performance has many metrics, of which speed is but one. Chip area and power consumption will be important for next generation computing as virtually all computing leaves the desktop and becomes more portable and mobile. Thus, computers will increasingly need to be designed as systems where a complex set of design trade-offs will need to be evaluated that include not only hardware design but the programming of individual processing elements as well as their coordination as a collection.

The full potential of Scenario-Oriented Design (SOD) has yet to be realized; it remains an ongoing research effort. SOD is just one design strategy that arises because of the potential of considering applications, schedulers, and hardware as they interact to form a system instead of viewing them as separate design domains. By leveraging one against the other, the system and its design process can be optimized in new ways. This is made possible by reducing the modeling detail of each design domain within a system in high level simulation.

## 7 Acknowledgements

The author thanks the other members of the MESH team, especially Don Thomas, Alex Bobrek and Brett Meyer. This work was supported in part by ST Microelectronics and the National Science Foundation (CNS-0406384). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

## 8 References

- [1] R.L. Glass, "Reuse: What is wrong with this picture?", IEEE Software, March/April 1998.
- [2] L. Benini, G. De Micheli. "Networks on chips: A New SoC Paradigm," *Computer*. pp. 70-78, 1/02.
- [3] J. M. Paul, D.E. Thomas, A. Bobrek. "Benchmark-Based Design Strategies for Single Chip Heterogeneous Multiprocessors," (CODES+ISSS), pp. 54-59, 2004.
- [4] K. Keutzer, et. al. "System-Level Design: Orthogonalization of Concerns and Platform-Based Design," *IEEE TCAD*, p. 1523-1543, Dec. 2000.
- [5] J. Paul. "Programmers' Views of SoCs," *CODES-ISSS*, '03.
- [6] A. Bobrek, J.J. Pieper, J.E. Nelson, J.M. Paul, D.E. Thomas. "Modeling Shared Resource Contention Using a Hybrid Simulation/Analytical Approach," *DATE*, 2004.
- [7] J. Paul, A. Bobrek, J. Nelson, J. Pieper, D. Thomas. "Schedulers as Model-Based Design Elements in Programmable Heterogeneous Multiprocessors," *DAC* 2003.
- [8] A. Cassidy, J. Paul, D. Thomas. "Layered, Multi-Threaded, High-Level Performance Design," *DATE* 2003.
- [9] B. H. Meyer, J.J. Pieper, J.M. Paul, J. E. Nelson, S. M. Pieper, A. G. Rowe. "Power-Performance Simulation and Design Strategies for Heterogeneous Multiprocessors," in press, *IEEE Transactions on Computers*.