Programmers' Views of SoCs

JoAnn M. Paul

Electrical and Computer Engineering Department

Carnegie Mellon University

Pittsburgh, PA 15213 USA

jpaul@ece.cmu.edu

Abstract

System-on-chip (SoC) designs have the potential to change the way we organize computation. This potential has gone unrealized. Future SoCs will have multiple heterogeneous processing elements, most likely organized around an on-chip network. Thus, SoCs are increasingly modeled as systems in the large. But a chip also has a fixed set of programmable hardware elements that are much more closely coupled than for systems in the large. New application types will require the chip to be considered programmable along with the individual processing elements on the chip. New programmers' views of SoCs are required to capture this new design space. A set of primitives for next generation design languages that support the development of new programmers' views of SoCs is motivated.

Categories and Subject Descriptors

C.0 [Computer Systems Organization]: General – *hardware/ software interfaces.*

General Terms

Performance, Design, Languages.

Keywords

Design Languages, Heterogeneous Multiprocessing, Networks on Chip, Programmers' Views, Systems-on-Chips (SoCs).

1. Introduction

Soon it will be possible to integrate a billion transistors on a single chip. Currently, two points of view dominate how they will be utilized. The first is that they will be applied to single processors that preserve existing programming abstractions with increasingly sophisticated hardware that is hidden from the programmer. The second is that Systems-on-a-Chip (SoCs) will result.

SoCs have been likened to other system types including system on a board that happens to be on a single chip, heterogeneous multiprocessors on single chips [1], and more recently, networks on chips (NoC) [2][3]. All of these views are borrowed from systems in the large. Systems in the large are systems that can be built across many chips. This includes virtually all computer systems other than single chip systems. Similarly networks in the large may even be wide area networks (WANs) which can describe systems with elements that may be literally a world apart from each other.

Virtually all current SoC design and modeling approaches can be applied to other kinds of custom systems, such as the traditional, embedded functionality in automobiles [4]. A separation of computation and communication [5][6] leads to the need to partition and design the system around explicit communications interfaces. Individual processing elements (PEs) have separately

compiled programs and memories for which all interactions are static and determined at design time, much like low-level hardware design. Many are left wondering if there is anything really new about SoCs that will impact single chip computer organization.

Future technology will provide the opportunity to consider a single chip as a network of heterogeneous processors, with the potential for hundreds of ARM equivalent processors to be integrated onto single chips within the next 5 years. The application space of next generation computers will pose new challenges beyond the present day focus on network processors, multimedia and embedded applications as primary SoC design challenges. Size and power constraints will result in a need to tune ever more sophisticated software to the machine upon which it executes. These systems will exhibit heterogeneous parallelism at the system level, prompting an entirely new type of system design.

Approaches to SoC design that are limited to dataflow are organized only around computation and communication, limiting the view of the SoC as an embedded, reactive system. Without a system-wide view of control, these approaches do not capture the rich set of possibilities when an SoC is considered a programmable entity. SoCs design languages such as SystemC are really only extensions of hardware description languages, focusing only on the net physical behavior of a reactive style of design.

Single chip hardware design has long permitted performance-optimization of control flow for a custom application. For SoCs the challenge will be in preserving the ability to optimize control flow for a semi-custom set of applications when design elements are parallel processors. The opportunity over that of systems in the large is the optimization of control flow across a *fixed* set of PEs tuned to a set of applications. New forms of programmable control flow in SoCs can result in new programmers' views.

A programmers' view (PV) is to a heterogeneous multiprocessor SoC what an Instruction Set Architecture (ISA) is to a single PE. The SoC may contain many individual ISAs for the many individual PEs it contains. However, the PV of the SoC captures the programmable nature of the chip as a distinct design artifact.

The set of instructions in an ISA includes three classes: that of computation, communication, and control of the underlying machine. The class of control instructions is what permits the machine to be considered programmable; this is so in single PEs because control instructions permit the program counter to be dynamically changed. A program counter is a pointer to a program which may, itself, be updated as a result of the program execution.

New PVs for SoCs must be defined and developed, organized around fixed sets of heterogeneous PEs and memories, ultimately permitting SoCs to be new, programmable devices that are not just mere extensions of hardware design. The PVs must include computation, communication and control as in Figure 1. We define system state for heterogeneous multiprocessing systems as the
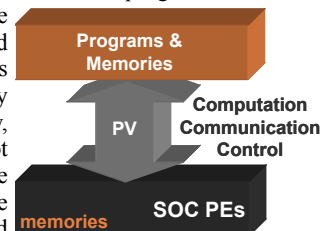

Figure 1 Programmers' View of an SoC

equivalent of a program counter in a single PE; it is finite state that can be used as the basis for dynamic, data-dependent decision making. An SoC can respond to situations programmatically, with data-dependent control flow optimized to a fixed set of PEs.

We start by considering the future application space of SoCs, motivating a need for effective implementation of sophisticated software on parallel processors. Then, we consider how control flow has been fundamental to both software and hardware computer organization by comparing and contrasting it to dataflow. We then consider how PEs might be organized on NoCs, again comparing control and data flow organization at the highest levels of design. By considering an SoC to have a fixed set of PEs, we contrast it with systems in the large, ultimately motivating the need to explicitly include coordination of PEs in support of system level control flow in PVs of next generation SoC design.

2. Future Applications

Probably the most competitive arena for computer design is in consumer electronics — where design time, size, speed, power, and innovation come together to make for competitive advantage. Most consumer-oriented future computing devices will be portable. A subset of applications that might appear on these 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, and a broad set of human computer interaction (HCI) software.

Research in ubiquitous and pervasive computing results in new scenarios for even more complex functionality with possibilities only limited by the ability to design and technically realize these scenarios in the individual computing devices [7]. No longer will it be possible to consider computer systems as falling into traditionally separate categories of embedded, general purpose, multimedia or network processors; future systems will have aspects of each. Novel applications will result from the interactions of previously separate types of functionality as computers simultaneously interact with the physical world, other computers, and humans.

The term "embedded computing" resulted from thinking of a computer system as being embedded in a non-computer system. Embedded system design is often thought of in terms of designing "hardware in the loop," which resulted in thinking of a computer system as reacting to the physical world with guaranteed response times. A reactive style of design naturally captures the need to communicate and compute in a fixed relationship with the non-computer system. However, most computers already interact with other computers in a networked sense.

HCI will become an increasingly important application area in its own right as traditional means of interacting with the computer (keyboard, mouse, display) will be replaced with multi-modal forms of interaction that include various forms of human "recognition" such as speech, handwriting and face recognition. Advanced forms of HCI are like real time simulations; they execute sophisticated models of humans in real time. Other kinds of simulations have been a successful application of parallel processing. HCI functionality that lies between the SoC's input and output pins will become increasingly sophisticated. While some kinds of embedded systems have functionality that is naturally limited by the requirements of the physical world, it is almost impossible to conceive of limits to the benefits of additional compute power applied to HCI.

The mix of new application types on these systems will result in a variety of performance demands. Some tasks may continuously execute while others may be event driven, and still others may be considered appropriate for best effort models. Different operating modes for the system may result in different performance demands for entire sets of tasks, as when different applications execute on today's desktop computers.

The important point is that the anticipation of what is possible in these systems must clearly utilize design principles that are at least as powerful of those of the past. While there will be increased ability to design for parallel execution of tasks, it will also be necessary to design the chip as a programmable entity.

3. Control Flow and Data Flow

Hardware Design Languages (HDLs) remain organized largely around the design of single FSMs. A significant feature of the FSM is that computation decisions are made around global machine state. The coordination of incoming data with the machine state permits a control flow style of design, in which the next state of the machine is computed as an interaction of incoming data with current state. This is depicted in Figure 2.

The positive aspects of the FSM include the ability to sequence operations in a programmatic style of design while the drawbacks largely
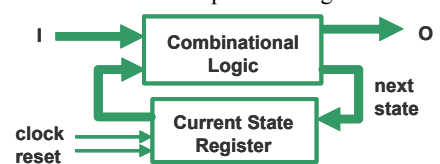

Figure 2 Finite State Machine

center upon the serialization of execution. This serialization extends to individual processors. A processor is an FSM where machine inputs are separately categorized as program and data. The interleaving of programmatic and data inputs at runtime allows for processors to execute more complex forms of control flow than FSMs, such as permitting design for conceptually unbounded amounts of state.

Figure 3 shows a single PE as a relationship between a program, its data, an ISA and an FSM. The figure shows that there are really two kinds of state that contribute to the overall performance of a conventional processor, that of the data and program stored in external memory and that of the underlying FSM. The state in the FSM is

Figure 3 Single PE

thought of as the state in the registers, including not only the data registers, but also the instruction pointer and program counter. This permits the finite state of the processor to control the conceptually unbounded state of the program it executes.
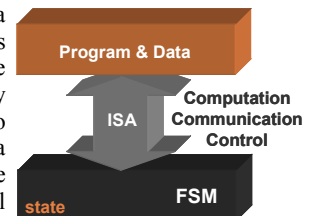
Unlike the state in an FSM, a processor memory location can mean different things at different times during the execution of a given program; this permits conceptually unbounded execution as memory resources are allocated and deallocated at runtime either explicitly by the program or implicitly by the context of the local variables in nested function calls. The conceptually unbounded execution allows a processor to approximate a Turing Machine; a software program is fundamentally different from a hardware description. System-level models that distill software and hardware to single models over-restrict the design space [8].

The state relationships in a processor may be considered layered; the memory external to the processor is in a layered relationship to the state of the FSM in the underlying processor [9][10]. This layered relationship permits software programs to rely on only a few features in the underlying FSM in order to execute on the FSM. However, one of the key features is the ability to dynamically load the program counter.

A processor not only loads and stores instructions and data (communication) and does arithmetic and logical operations on them (computation), but it also can load the program counter in response to specific situations, thus changing its execution sequence in response to system data (control).

Note the similarities between Figure 3 and Figure 1. While Figure 3 is a classic view of a PE, Figure 1 is what we propose for future SoCs where the hardware is a collection of heterogeneous, programmable processors. In section 5, we propose system state as the equivalent of the program counter in the analogy. It is finite state which can be updated dynamically as a result of computation and around which data-dependent decisions can be made. However its coordination across a fixed set of PEs must be explicitly supported. The point is that PVs that permit designers to define and effectively manipulate control flow across a collection of programmable resources on an SoC are required.

In recognizing the parallel nature of computation on SoCs, many have proposed that SoC system-level design should be organized around dataflow style graphs such as shown in Figure 4. The figure shows a dataflow graph in which sequencing is unidirectional (no feedback). In such graph-based approaches, computation is presumed to be triggered by tokens [4][5][6]. Static

Figure 4 Dataflow Graph

analysis is required so that response times are predictable and queues need not be considered unbounded. The concept is not unlike that of gate-level design where the execution of individual gates is triggered by events.
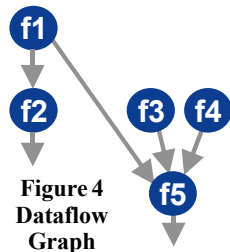
Dataflow tends to result in a graph-based style of design with static execution schedules; this is evidenced by the observation that dataflow tends to be described by graphs with vertices and edges while control flow often results in program-like descriptions. Dataflow graphs are poorly suited to describe arbitrary, complex software programs. Programmatic data dependencies resolved at runtime permit the system to respond to a wider range of situations. Programs can have conceptually unbounded loops on single processors and multithreaded descriptions where M tasks can be mapped to N resources on multiprocessors.

For the past several decades, the synchronous finite state machine (FSM) and the programmed processor have been the dominant design abstractions for digital computation. They have conveniently captured the underlying hardware abstractions as well as provided a basis for effective specification of system functionality in no small part because it is easier to design computer system functionality in terms of a control flow sequence.

Sequential specification of functionality requires some context for global state. For single processor systems and FSMs, the global state is not problematic, it is a natural part of the specification of the behavior of the machine. But for parallel computer machines, such as future SoCs, architecting global state so that dynamic decision making across a set of parallel processors can be effective has been challenging [11].

Heterogeneous multiprocessing on single chips and the increasingly sophisticated applications that mobile computers will execute are each virtual certainties. Some state will be private and local to resources. Overall performance for many applications will benefit from utilizing private state and the establishment of dataflow across a number of PEs. However, the need to maintain some forms of global control flow across a number of PEs will also be important for many of these applications.

This leads us to the first of a set of modeling primitives. We propose that future design languages that support the development

of PVs for SoCs must include:

PV-P1) trade-offs of control flow vs. data flow implemented across a number of PEs.

We will number our proposed set of modeling primitives throughout the paper as PV-Pn, for design primitive number n.

By considering SoCs as being organized around NoCs while at the same time pointing out the differences between NoCs and networks in the large, we next motivate possibilities for effectively preserving control flow in high-level PVs of SoCs.

4. Networks on Chips

Networks on Chip (NoC) have been proposed to support modular design while enabling high-performance circuits to reduce latency and increase bandwidth. They also support the globally asynchronous, locally synchronous (GALS) nature of design as single clock domain systems will no longer scale to future device densities and chip sizes. [2][3]

While NoC may be suitable for solving problems related to on-chip communications it does not address how state will be advanced across the chip. Concerns that single chip systems conceived as networks of processors will suffer from the same kinds of obstacles that made it difficult to achieve performance benefits from parallel systems in the large seem valid. Yet, while there are many similarities between NoC and networking in the large, there are differences between even the most fundamental objectives of the two. By taking advantage of the fixed and finite set of PEs on a chip in conjunction with the programmable nature of a network, new PVs for SoCs become apparent.
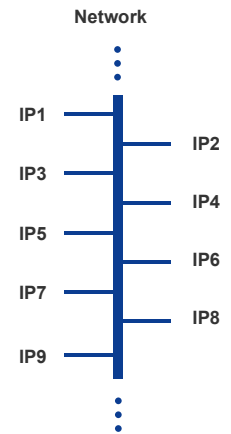

Figure 5 9 nodes on an Extensible Network

4.1 Finite, Programmable Networks

Networking in the large had a primary goal of robust communications. The network protocol was designed so that communications would take place despite the disabling of a large portion of the infrastructure. Large portions of the network can be disabled and communications will still work between remaining portions of the network. Accordingly, networking in the large is designed to support a conceptually infinite number of network nodes; the network protocol was designed to support a system for which the exact number of nodes on the network at any one time need never be known, which is clearly the current reality!

By contrast, a chip has finite extent. Consider a general purpose network, as in Figure 5. The figure shows nine nodes, labeled as nine Intellectual Property (IP) elements. While a simple diagram, Figure 5 contains several implications. First, all of the nodes in the network are considered peers. There is no dominant network node unless one is established with some higher level of programming that sets up the cooperation of

Figure 6 9 IP blocks with NoC interfaces

the nodes. Second, the figure implies that there is no preferred direction of information exchange between any two nodes on the

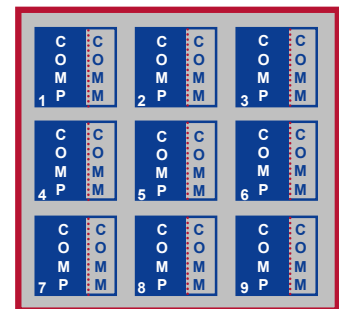network. Again, if one node is preferred to another, this must be set up in a programming context. Third, the diagram implies that the network is extensible; nodes may be added to (or subtracted from) the network and the network abstraction will be preserved. The number of nodes is conceptually unbounded. Finally, the state in the system is presumed to be perfectly distributed. Each node in the system may contain state, but the state of the system is the aggregate state of all nodes in the system at any point in time.

While the network of Figure 5 is the inspiration for NoC, Figure 6 illustrates how nine intellectual property (IP) blocks with common network interfaces might be placed on an SoC. For most network models, each PE on a network includes support for network communication. Figure 6 shows this for the single chip where the nine IP blocks are divided into computation and communication parts. The similarities between Figure 5 and Figure 6 are that each of the IP blocks have equal access to a global network that allows the blocks to communicate in a GALS manner. Further, packets will be used for information exchange between the IP Blocks.

However, there are significant differences between the networks of Figure 5 and Figure 6. While the number of IP Blocks shown in Figure 6 is conservative at only nine, the important point is that it is a fixed set. While it will be possible to conceive of hundreds of PEs in the next several years the actual number and types of PEs on a single chip will be known at design time. Packet exchange can take place in a pre-programmed manner; the state upon which dynamic decision making is based can be finite.

Note that acknowledging the finite nature of an SoC does not diminish the scalable properties of communications organization around a network. It remains useful to have a communications standard that applies for ever-larger systems. However, the scalable abstraction must not constrain designers from taking advantage of optimizing software and hardware around a fixed set of programmable resources.

Because the chip has a fixed set of PEs, it is possible to explicitly define and coordinate a level of global state around which dynamic, control flow decisions can be made. This leads us to our second modeling primitive:

PV-P2) *selection of a fixed set of PEs to be considered a programmable collection.*

The fixed set of PEs can then be optimized to the applications for which the chip is being architected.

4.2 Local vs. Global State

At the heart of the issue of SoC design is what state on the chip should be local (private) vs. global (shared). Current design languages and methodologies do not permit designers to reason about trade-offs between the two. Clearly not all state should be global to the chip. However, design paradigms that consider all state to be privatized in individual PEs or IP blocks from the start fail to capture the singular strength of single chip designs — that large collections of PEs can be more closely coupled than ever before with sets of PEs and the method of communication between them more tuned to applications than ever before.

With advances in the ability to integrate multiple computers onto single boards and then chips, shared memory gains advantages [12]. On single chips, busses that support shared memory can become major bottlenecks. However, the use NoCs need not result in the abandonment of global state.

NoCs can create a new class of network, with more specificity of than the prioritization of packets for Quality of Service (QoS) for networks in the large, and less specificity than networks on FPGAs where the network is intended to mimic static routing.

NoCs can be programmed to dynamically make routing decisions for different types of packets. Some packets can be

exchanged with greater efficiency than others. Some state can appear shared among resources because it can be exchanged far more efficiently, while other state appears distributed — even though all state is exchanged across the network. This can be based solely on how the network is programmed. This leads us to two additional modeling primitives:

PV-P3) *trade-offs of global vs. local state.*
PV-P4) *customization of the on-chip network protocol.*

These provide a basis for distinguishing system level state around which decisions are made across multiple programmable resources [13], and state which supports other computation.

5. System State

While some problems are designed to scale with the number of resources in a system, others require the optimization of a fixed set of processing resources to a problem of fixed size. Applications that have near perfect scalability are rare. Thus, explicitly tuning a set of heterogeneous, interacting resources to specific portions of computation can provide a vast performance improvement over that of a purely scalable or extensible solution. Design optimization for this class of problems must be achieved by optimizing the coordination of a fixed set of elements to carry out the behavior of one or more computational problems. This can be considered analogous to the design of an application specific processor (ASIP) for a class of applications, except that the processor is now a collection of individually programmable, networked PEs.

We define *system state* as state which supports the coordination of information flow among a set of PEs. Like a program counter, system state is finite but may be dynamically updated at runtime. System state can support programmatic, data-dependent decision making across a set of PEs. C*omputation state* is all other state in the system, such as that of individual PEs that compute the net system functionality.

Figure 7 System State (s) & Computation State (c)

Figure 7 shows the same nine IP blocks on a single chip. The state in each block has been divided into computation state (c_i) and system state (s_i). System state can be used as a basis for coordinating information flow among a set of processing resources in a variety of ways.

The mechanisms for coordination of system state can be architected at design time. But it can also support run-time decision making. Thus, system state is more like a chip-level program counter than a schedule synthesized at design time. System state directly supports chip-level programming of SoCs. As such, it is more of an architectural feature which is a by-product of the fact that the design elements of the SoC are heterogeneous multiprocessors. This leads us to our next modeling primitive:

PV-P5) *chip-level architectural infrastructure, such as finite system state, distinguished from computation state, around which runtime decisions concerning the coordination of collections of PEs are made.*

5.1 Two Forms of System State

Network interfaces coordinate network routing decisions according to a common protocol. While it may be the goal of system designers to keep the amount of packets stored in network

queues to a minimum, only in a purely statically scheduled system will there be no information stored in the network. When decision making is made in the presence of state, programming is possible.

Thus, the programming of a network can be a primary factor in coordinating state advancement across the elements of the chip. In addition to state in individual processor resources, the state of the network can also contribute to the system state. This situation is shown in Figure 8, where the system state is shown to have two parts. At each node, i, where i=1...9 for the SoC, there is a contribution to system state
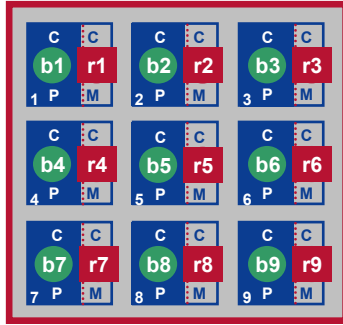

Figure 8   Routing (r) and IP Block (b) Forms of System State

from the network as routing state, r_i, as well as from the computation at each IP block, the b_i. In general, system state at each node, i, is $s_i=(r_i,b_i)$, while the overall system state, S, on the SoC, is the set of all system state contributed by each node, or $S=\{s_1,...s_9\}=\{(r_1,b_1),...(r_9,b_9)\}$.

While the non-ideal properties of a network may be thought of as a liability, the possibility to program the network to make decisions about which packets are routed next can be an asset. For instance, system state may be routed at a much higher priority than computation state so that control flow constructs may be set up across collections of PEs. Unlike quality of service (QoS) for networking in the large, which must be based largely upon statistical and analytical techniques because of the lack of a priori knowledge about applications and underlying resources, a chip with a fixed set of PEs permits a network to be considered programmable, setting up a richer set of possibilities for the coordination of PEs than pure dataflow or pure control flow.

5.2 Coordinating System State

Currently approaches simplify the problem of multi-element coordination on SoCs by considering the chip to fall into one of two broad categories, that of control flow based upon a central controller and that of data flow where static forms of scheduling route information between elements. In each case, system state, S, is greatly simplified. We illustrate each of these on our simple

Figure 9   Central Controller

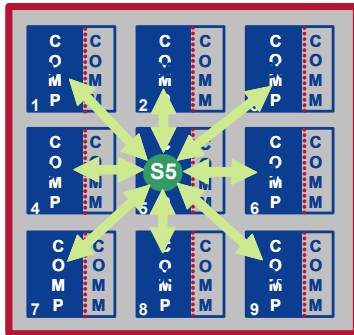SoC diagrams, thus motivating why they are overly simplistic for the rich design space of next generation SoCs.

5.2.1 Control Flow Central Controller

Control flow is easiest to realize when a central controller coordinates all system resources as master-slaves; this is shown for the nine node SoC in Figure 9 where block 5 is a central controller. In this case the network might be designed with a naive protocol. All system state resides in the central controller, i.e., S=s5. The main advantage of coordinating system state in this way is in the preservation of a sequential control flow scheme for programming the chip; this is largely a programmatic solution in which distinct blocks of software may be accelerated by execution on custom

hardware or on many blocks in parallel, if not both. Many current SoCs fall into this category of design, where a general purpose processor serves as the central controller.

The main disadvantage of this approach is that the central controller can be a bottleneck The system can only execute as fast as the sequential fractions of software [14] can execute on block 5. In general, the software executing on the central controller must be as simple as possible or the central controller must have sophisticated scheduling that tunes the application to the slave processors so that throughput can be maintained.

For certain applications with highly independent parallel threads, this can be a valid solution. However, most of these applications fall into two categories which differ from that anticipated for future SoCs: highly regular computations such as matrix product or server applications in which multiple users access the same set of resources. However, new PVs that permit tuning the application to a heterogeneous set of processors on a programmable network can reveal how control flow can be effectively supported.

5.2.2 Dataflow

Many designs can benefit from optimizations when information flows through a fixed number of hardware resources in a fixed pattern. This dataflow style of design is suited to algorithms for which computation can be effectively pipelined. This situation is depicted in Figure 10 for the nine node SoC.

In the figure, the data is routed from node to node in a statically scheduled manner that allows the nodes to be thought of as interconnected as in a graph. Information is shown entering the system on node 1, then pipelined to exit the system via nodes 6 and 8. The first pipeline is nodes 1, 2, 3, 5, 9, 8, 7, and the second pipeline is nodes 1, 2, 4, 6. All of the system state can be thought

Figure 10   Dataflow on Chip

of as being in the network where packets act as tokens that trigger computation elements in a static manner, i.e., $S=\{r_1,...r_9\}$.
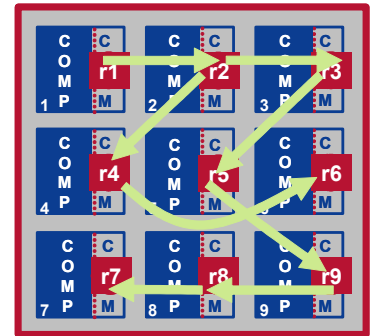
A significant amount of information about both the application and the processing capabilities of each node needs to be known in advance of designing a statically scheduled system. For data-independent functionality, static scheduling can give the greatest performance; it is much like the pipelining of a processor. For an SoC, with many computation units, the possibility that certain pipeline stages can take advantage of parallel computation units can further enhance system performance. However, the limitations of data flow all focus on the need to consider the system statically scheduled with no data dependencies. Thus, the next primitive:

PV-P6) system state as residing in both the programmable network and the programmable IP blocks.

Beyond either of these two categories, better techniques for coordinating sets of PEs are needed for future SoC designs.

6. The Need for New Programmers' Views

We assert that in the absence of new PVs for SoCs, the demands that future computer system applications place on designers will not be met. Optimal SoC designs will most likely lie between that of a central controller residing on a single PE (Figure 9) and pure dataflow designs (Figure 10) with no central controller to dynamically direct resource cooperation. More complex models of

S will appear in new forms of control flow coordinated across multiple PEs. Designers must be able to effectively consider trade-offs between system state and local state in next generation SoCs.

A PV for an SoC is distinguished from an ISA because it must represent not only the available set of computation, communication and control primitives, but must also include a *means* by which control flow is coordinated around system state. For example, when system-level control flow does not reside in a single processor accessing a single memory (i.e. unlike Figure 9), system infrastructure must be defined and coordinated in new PVs.

6.1 Finite, Distributed Control Flow

Consider the central controller of Figure 9. While many programming models include the controller as a PE, SoCs afford the possibility that the state and functionality upon which control flow decisions are made does not reside on a central PE. This situation can be likened to the way a single scheduler can be formed by the cooperation of schedulers on individual PEs. In distributed system design in the large, the coordination of separately cached copies of data must be explicitly considered. Currently this problem is solved by various operating systems and middle-ware. However, the problem of achieving performance enhancement over a collection of resources is difficult for systems in the large, where extensibility is prioritized. The overall philosophy is one of hiding the properties of the underlying hardware platform from the distributed software applications so that net performance is relative to the available resources. This is an extension of the philosophy that software design should be considered independent of as many properties of the underlying machine upon which it executes as possible. Mechanisms for state coordination across multiple resources in the form of middleware provide the means of thinking in terms of distribution primitives without the need to think in terms of the actual size and capacity of the underlying machine.

However, once a chip is designed, nodes will not be added or subtracted. Control flow can be optimized over a fixed set of PEs by explicit coordination of finite system state if explicit support is provided for design of system infrastructure as a part of the PV of the chip. Finite, distributed control flow is one optimization possible when support is provided for viewing the chip as a programmable set of heterogeneous PEs.

6.2 Multi-path Control

Dynamically scheduled datapaths may be established across sets of PEs on an SoC. Either a central controller or distributed, finite control may dynamically set up patterns of cooperation among PEs where the patterns are tuned to specific programming situations. We refer to this property as *multi-path control*.

Consider PE 1 of Figure 10 to be a central controller, which implements all of the functionality of a present-day personal digital assistant (PDA), plus enhanced HCI and computer-computer networking, as discussed in section 2. Figure 10 shows one way the central controller (PE 1) might set up the cooperation of the remaining PEs in the system when the PDA is acting as a cell phone in the middle of a call. However, when the PDA is not actively processing a call, PE 1 might well set up a different pattern of dataflow among the PEs in the system. In multi-path control, multiple data paths are set up by a scope of control flow, either a single processor or a cooperative collection of processors; PE 1 need not be considered a single processor when control decisions are made using distributed, finite control. This leads us to our final proposed modeling primitive:

> *PV-P7) novel architectural views for coordinating system state and control flow across a finite number of PEs such as finite, distributed control flow and multi-path control.*

7. Conclusions and Future Directions

The set of proposed primitives that new design languages must support as discussed throughout this paper might seem straightforward. However, the important point is that existing system design approaches and languages such as SystemC do not directly include them.

Current languages do not permit designers to think in terms of systems in which not only will individual PEs be programmable, but the chip, as a whole, is considered programmable, including the network that supports information exchange across the chip. SoC hardware is a fixed set of PEs; its PVs can significantly differ from past forms of system organization.

Next generation design languages must enable the architecting of novel SoC designs, focusing on explicit support for the design and programming of system infrastructure that coordinates information flow across a fixed set PEs. The modeling primitives of such languages must capture the performance cost-benefit of a rich middle ground, between pure dataflow, pure control flow, pure global state, and pure distributed state. Design languages must also support the design of custom networks and the selection of the PEs that are programmed as a collection, forming the basis of new programmers' views of SoCs.

8. Acknowledgements

9. References

[1] "Are Single-Chip Multiprocessors in Reach?" *IEEE Design & Test*, Jan - Feb 2001.

[2] W. Dally, B. Towles. "Route Packets, Not Wires: On-Chip Interconnection Networks," *DAC* 2001.

[3] L. Benini, G. De Micheli, "Networks on chips: A New SoC Paradigm," *IEEE Computer*, January 2002.

[4] F. Balarin, et al. *Hardware-Software Co-design of Embedded Systems. The Polis Approach.* Boston: Kluwer. 1997.

[5] W. Cesario, G.. Nicolescu, et al. "Colif: A Design Representation for Application-Specific Multiprocessor SOCs," *IEEE Design & Test of Computers*, Sept. - Oct. 2001.

[6] K. Keutzer, et al. "System-Level Design: Orthogonalization of Concerns and Platform-Based Design," *IEEE T. CAD*. '00.

[7] Special Issue on Ubiquitous Computing, *Communications of the ACM,* December 2002.

[8] B. Grattan, G. Stitt, F. Vahid. "Codesign-extended applications." *CODES 2002.*

[9] J. Paul, D.Thomas. "A Layered, Codesign Virtual Machine Approach to Modeling Computer Systems," *DATE* 2002.

[10] A. Cassidy, J. Paul, D. Thomas. "Layered, Multi-Threaded, High-Level Performance Design," *DATE* 2003.

[11] D. Skillcorn, D. Talia. "Models and Languages for Parallel Computation," *ACM Computing Surveys.* 1998.

[12] D. Culler, J. Singh. *Parallel Computer Architecture, A Hardware/Software Approach.* S.F.: Morgan Kaufman. 1999.

[13] J.Paul, A.Bobrek, J. Nelson, J. Pieper, D. Thomas. "Schedulers as Model-Based Design Elements in Programmable Heterogeneous Multiprocessors," *DAC* 2003.

[14] G. Amdhal. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *AFIPS 1967 Spring Joint Computer Conference*. pp. 483-485.