A Hierarchical Modeling Framework for On-Chip Communication Architectures

Xinping Zhu and Sharad Malik
Princeton University
Princeton, NJ, 08544, USA
Email:{xzhu, sharad}@ee.princeton.edu

Abstract— *The communication sub-system of complex IC systems is increasingly critical for achieving system performance. Given this, it is important that the on-chip communication architecture should be included in any quantitative evaluation of system design during design space exploration. While there are several mature methodologies for the modeling and evaluation of architectures of processing elements, there is relatively little work done in modeling of an extensive range of on-chip communication architectures, and integrating this into a single modeling and simulation environment combining processing element and on-chip communication architectures. This paper describes a modeling framework with accompanying simulation tools that attempts to fill this gap. Based on an analysis of a wide range of on-chip communication architectures, we describe how a specific hierarchical class library can be used to develop new on-chip communication architectures, or variants of existing ones with relatively little incremental effort. We demonstrate this through three case studies including two commercial on-chip bus systems and an on-chip packet switching network. Here we show that through careful analysis and construction it is possible for the modeling environment to support the common features of these architectures as part of the library and permit instantiation of the individual architectures as variants of the library design. As part of this methodology we also show how different levels of abstraction of the model can be supported and viewed as different variants that can be used in an accuracy versus simulation time trade-off.*

I. Introduction

The distributed computation architecture of a System-on-a-Chip (SoC) can be generally decomposed into two interrelated parts: Processing Elements (PEs) and the On-Chip Communication Architecture (OCA). The PEs are responsible for the computation of the desired functions and may be hardwired (referred to as the Application Specific Integrated Circuit or ASIC part) or programmable processors. The OCA provides communication mechanisms which make it possible for distributed computation among different PEs as shown in Figure 1. A suitable OCA is vital for the success of an SoC design in terms of timing performance, energy consumption and design cost.

Technology advances have enabled designers to have greater freedom in selecting from different types of communication schemes. The traditional way of interconnecting on-chip modules is via on-chip buses, such as the IBM CoreConnect Bus Ar-
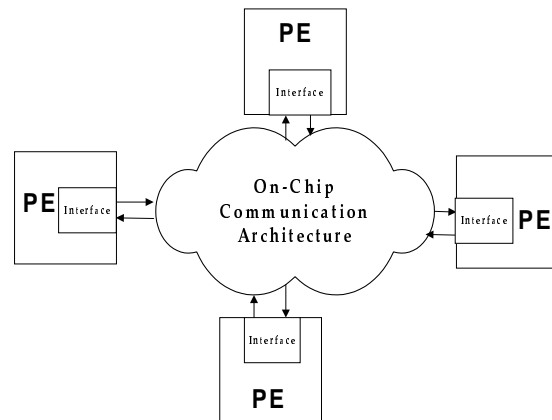

Fig. 1. An SoC with PEs and OCA

chitecture [1] and the ARM AMBA bus system [2]. An emerging option for integrating a large number of processors is to use on-chip networks as the backbone for inter-module communication [3], [4].

Clearly these two communication styles differ greatly in terms of interface, topology and communication protocols. However, a design environment that could potentially select one over the other must be able to model and support these choices with their variants. The choices between competing OCAs are driven by the requirements of the application. A streaming video SoC and an SoC targeting network routing may have very different communication characteristics, thus having completely different needs for the OCA. However, making the correct choice for the OCA is hard, because it involves a complete understanding of the computation and communication requirements for the specific application and thus interplay between the application, PE architectures and the OCA.

In this paper, we present an exploration framework that helps a designer to understand this interplay and make the right design choices, in particular the right OCA choice. The organization of the rest of paper is as follows. Some background concepts and terminology are introduced in Section II. Section III details our proposed modeling methodology for OCAs from both behavioral and structural points of view. Section IV describes the two modeling and simulation environments we are using to apply this methodology. The detailed modeling exercise together with some modeling strategies and techniques are discussed in Section V. Section VI examines the criteria to evaluate our method-

ology and modeling framework. Finally, conclusions and future work are provided in Section VIII.

II. Background

In this section, we introduce some concepts and terminology that are used in the rest of the paper. We also highlight the similarities and dissimilarities between the modeling of PEs and OCAs.

Virtual prototyping [5] helps make design choices starting from the very early stages of the design cycle. It enables us to predict the response and critical performance attributes from a constructed trustworthy prototype, even before a physical prototype is built. The virtual prototype is often a heterogeneous architecture composed of software and hardware modules in different levels of granularity. Early design space exploration is only possible through execution-driven virtual prototyping. Correct performance models include cycle-accurate heterogeneous and hierarchical models for both the PE and OCA. The coordination between these concurrent components in the PE and OCA models is described by the model of computation (MoC) [6]. Our system design methodology provides an integrated modeling framework that enables independent design space exploration of the PE architecture as well as the OCA [7].

A. Modeling of PEs vs. OCAs

The discipline of computer architecture clearly distinguishes between the Instruction Set Architecture (ISA), which is seen outside of the processor, and the micro-architecture, which comprises implementation details such as the number of pipeline stages, size and organization of on-chip cache memories and number of Function Units (FUs), etc. This distinction enables computer architects to explore the functional and implementation aspects of processors separately. It recognizes that one functional specification can have many different implementations with different values for design metrics. OCAs can also be analyzed in the same fashion. On the function side, there exist different ways of sending and receiving data from the OCA, e.g. read/write data in a shared memory model, send/receive data packets in a message passing model. From a PE viewpoint, an OCA can always be abstracted as a pipe with latency with either memory reads and writes or inter-PE messages. On the implementation side, the OCA contains various details such as input/output controllers, buffers, arbiters, crossbars, etc. However, a major dissimilarity exists between the architecture and microarchitecture dichotomy between the modeling of PEs and OCAs. Both the ISA and micro-architecture for PEs are well-understood and defined, which is not the case for OCAs. In general there exist no well-defined semantic and structural primitives for OCAs. Filling this gap is a primary goal of this paper.

III. Methodology

In this section, we present our modeling methodology which will enable designers to easily classify and develop modular software models for OCAs. Section III-A describes the counterpart of the PE "ISA" for OCAs - the functional primitives that also serve as the bridge between the PE architecture and the
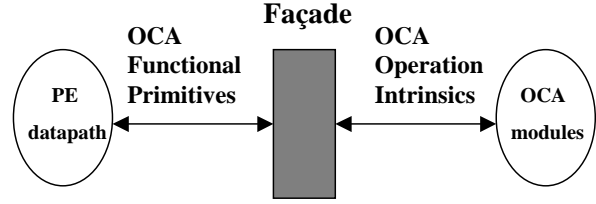

Fig. 2. Façade between PE and OCA

OCA. Section III-B discusses our proposed OCA module classification hierarchy and shows by examples how to utilize this hierarchical structure.

A. Functional Primitives of OCAs

The ISA of PEs has been well studied for more than two decades. However, there has not been much work done on its counterpart in OCAs. To shed light on these OCA functional primitives, our first step is to classify various kinds of communication mechanisms into a small and well-defined set of functional primitives which would allow the decoupling of the PE architecture and the OCA.

There are two basic models of parallel computation: shared memory and message passing. We define the following atomic constructs as the functional primitives of the OCA in the shared memory and message passing models (each PE is denoted by its address $i \in N$):

OCA read (x, u) moves data x in the shared memory into local variable u

OCA write (y, v) writes the value of local variable y into shared variable v

OCA send (x, i) sends the value of x to PE i asynchronously

OCA receive (y, j) receives the value of x from PE j synchronously

This generic OCA "ISA" provides a basis for the communication primitives needed. For a specific OCA, the actual functional primitives may vary, but they are likely to be variants of the above (e.g. remote reads and writes). This separation of the functional primitives and their actual implementation is useful in both system simulation and compilation. Functional simulation needs to understand only the semantics of the primitives and thus can be fast. Detailed timing simulation will naturally require the micro-architectural implementation models.

For example, a bus read transaction would involve one OCA read primitive and one OCA write primitive in a shared memory model. Likewise one OCA send primitive and one OCA receive primitive are needed to accomplish a packet data send/receive transaction in a message passing model. Even though these four constructs are enough for our modeling and simulation purpose, further refinement and addition is possible, such as composite primitives of remote read and write operations.

In addition, these OCA primitives serve as the basis of the Façade [8], a high level unified interface to the functionality of the OCA sub-system that is presented to the PE sub-system. As shown in Figure 2, the separation between the two complex subsystems, PEs and the OCA, enables distinct exploration within
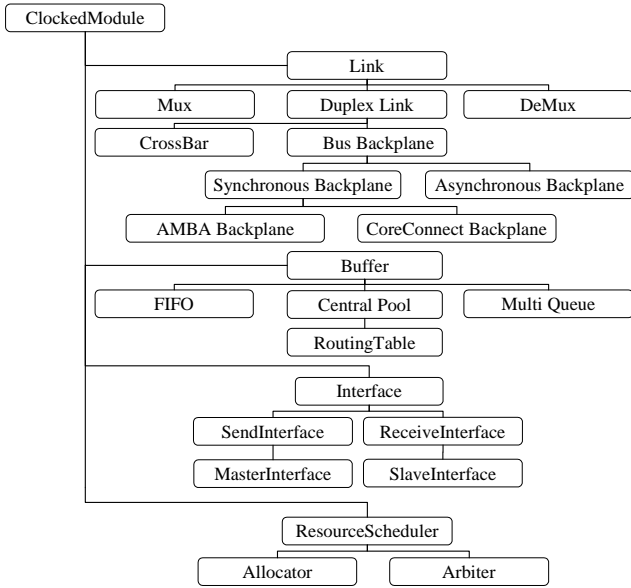
Fig. 3. Class Inheritance Hierarchy of OCAs

```
public class FIFO extends Buffer{
    Queue _queue;
    public int enqueue(PacketToken t){

       ...

    }
    public PacketToken peekAndDequeue(){

       ...

    }
}
public class MultiQueue extends Buffer{
    Queue [] _queue;
    public int enqueue(PacketToken t){

       ...

    }
    public PacketToken peekAndDequeue(){

       ...

    }
}
```

Fig. 4. Code example in PtolemyII

the two domains. After we identify this desired unified interface for a whole family of communication architectures, we design a "wrapper" class that can encapsulate the use of the sub-systems. The client of the communication sub-system, which is the inter-PE communication part is only exposed to a façade which provides a universal "wrapper" of data exchange functionality without the complex details of the communication sub-system. This façade translates the inter-PE communication instructions into the pre-defined internal OCA intrinsics of the communication subsystems. These OCA intrinsics are implementation specific and not seen on the PE side. Then the communication subsystem acts upon these OCA intrinsics to generate an implementation specific OCA operation sequence, such as generating a request for a communication resource, injecting a data packet, etc.

For instance, on one side of the façade, the PE datapath issues OCA shared memory functional primitives, such as *write (y, v)*. On the other side, it is a on-chip bus system with OCA intrinsics such as *bus_request*, *bus_write*, *bus_release*, etc. It is the job of the façade to translate between these two domains, generating intrinsics corresponding to a bus write transaction conforming to the specific bus protocol. Then the OCA modules execute upon these OCA intrinsics as a script for concrete OCA actions in sequence.

B. The Reusable Class Hierarchy of OCA Structural Primitives

Figure 3 shows the derived class inheritance hierarchy after the object-oriented analysis (OOA) [9] of OCA functional units. In this diagram there are 4 basic types of OCA structural components with which it is sufficient to construct a complete OCA, such as buses and packet-switching networks. They are *links*, *buffers*, *resource schedulers* and *interfaces*. *Links* connect different types of OCA functional units together. *Buffers* temporarily store data or tokens. *Resource schedulers* decide which module has the right to access specific communication resources.

Interfaces translate datapath communication actions into implementation specific actions, such as bus request and injection of generated packets. All of them share certain parameters and characteristics, such as data-width and latency which are commonly inherited from the root class *ClockedModule*. Furthermore, each class has its own non-inherited features. For example, by incrementally inheriting the *link* class, more complicated interconnection latency and functional characteristics are added. Modules such as *bus backplane* and *crossbar* are child classes of *link*. In short, this class diagram captures both the structural primitives and the behavior relationship between these similar components.

Figure 4 shows some Java code skeletons of the buffering modules we have developed in PtolemyII. By inheriting the common ancestor, *Buffer*, both *FIFO* and *MultiQueue* override the methods *enqueue* and *peekAndDequeue* to implement their own buffering algorithm. Also, both of them share the same input and output ports which are not shown in Figure 4.

To develop a reusable *resource scheduler*, we studied different types of implementations of OCA arbitration units. We differentiate them from both the interface and arbitration algorithms. A base *Arbiter* class encapsulates basic skeletons of request/grant style port lists and arbitration algorithmic methods. A priority based arbitration module, known as *PriorityArbiter*, inherits the *Arbiter* class while adding one more input port *priority*. Correspondingly the arbitration method of *PriorityArbiter* needs to be modified to reflect the priority based algorithm. On the other side, a round-robin style arbitration module, known as *RRArbiter*, only need to override its inherited arbitration method from its superclass *Arbiter* without modification in the interface. Cycle accurate timing information of each module is also modeled at this stage. Together with the buffering modules, the module library is eventually populated with tested and trustworthy modules upon which a complete OCA can be built.

A behavioral hierarchy structure also helps in locating different design choices from an organized module repository. Sim-
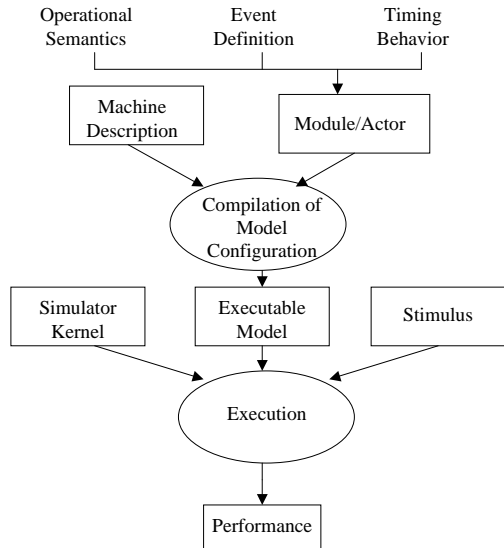
Fig. 5. Modeling and simulation process using PtolemyII and LSE

ilarly novel buffering algorithms can be incorporated by easily overriding the inherited methods. Richer functional details can be added to specific modules through inheritance and "white-box" reuse of the existing modules.

Determining a reusable and flexible hierarchical class diagram is not a one-time effort. The class diagram and the library need to evolve over the design process. Technology advances and new application domains often give fresh inputs to the models. These new functional modules can be placed in the appropriate spot in the module library while sometimes part or the whole hierarchical structure may need to be modified to respond to new modeling requirements.

After the instantiation of these developed OCA modules into specific OCAs, the original system block diagram is transformed into a netlist of modules such as arbiters, buffers, links and interconnections between them. After each module is provided the needed parameters, the OCA model is ready to be simulated together with the OCA stimulus or the PE model.

IV. Modeling Infrastructure

We have implemented the methodology described in Section III using two distinct modeling and simulation environments. The first one is in-house - the Liberty Simulation Environment (LSE) [10] developed for programmable architectural platforms. In addition to this, we have also implemented our methodology using PtolemyII [6]. The following taxonomy is used throughout the following sections. The basic building blocks of the modular modeling environment are called "modules" or "actors". Each module has ports through which "tokens" are sent to each other.

Liberty Simulation Environment. LSE is a fast execution-driven compiled-code modeling and simulation framework. LSE constructs concurrent structural models and retargetable compiled code simulators from a unified structural machine description and specification database. These descriptions are eas-

ier to write than hand-coded micro-architecture simulators, enabling rapid evaluation of many architectural designs, from general purpose (e.g. Alpha, IA-64) to application specific such as IBM PowerNP and Intel IXP1200. Its flexibility ties in with our end-goal of providing a complete environment for exploring emerging SoC architectures, where both PE architectures and OCAs play a critical role.

In LSE, physical hardware blocks are modeled as logical functional modules that communicate through ports. Data is sent between module ports via message passing. Each module has its own pre-defined parameters and custom control functions. For example, typical parameters for a buffer module include buffer size and width, with ports such as read/write ports. Each LSE module is a piece of sequential code executing concurrently with other modules. Modules are written in C for fast simulation. LSE instantiates the modules from a custom machine configuration, separately compiling and linking them in building an instance of the simulator.

PtolemyII. PtolemyII is an object-oriented, heterogeneous design and modeling framework. It targets the modeling, simulation and design of concurrent heterogeneous systems which are usually hard to model in one unified design framework. Its description language can easily describe complex hierarchical interconnections of parameterized executable components. PtolemyII, like LSE, supports construction of executable models in a modular fashion.

Figure 5 shows the general process of building a custom simulator using PtolemyII and LSE. PtolemyII is a fully object-oriented modeling environment written in Java - a modern object-oriented language. In comparison, LSE uses C and macro processing languages to support the modular construction of compiled-code simulators. The usage of C by LSE provides it with an efficiency advantage, but more effort is required in implementing the hierarchical class diagrams by explicit code inheritance within a modular infrastructure.

For both PtolemyII and LSE, a simplified distributed discrete event (DE) [11] model as a MoC enables us to simulate multiple PEs and an OCA together in an execution driven fashion. In OCA modeling, cycle-accurate models are built based on the DE timestamp mechanism. Clocks may also be used to synchronize events. The system simulator updates its state at each clock cycle. In this scenario each simulator, including PE simulators and OCA simulators can be treated as individual processes and may proceed independently and process events in their own event queues.

V. Case Studies

In this section, we present our modeling effort of two distinct OCAs which are currently employed by SoC designers - on-chip buses and on-chip packet switching networks. Section V-A describes the modeling of two commercial on-chip bus systems and Section V-B discusses the modeling of an experimental on-chip packet switching network reusing some components developed in the bus modeling exercise.
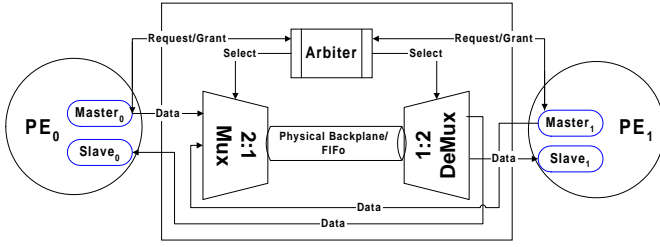
Fig. 6. Architectural Model of a Simplified On-Chip Bus

Fig. 7. Router architecture as modeled in both PtolemyII and LSE

A. Modeling of Two Modern On-Chip Bus Systems

We modeled the AMBA [2] and CoreConnect [1] bus systems as our first exercise of modeling OCAs using the methodology described above. We studied these two commercial on-chip bus standards together with the European-developed bus standard, the OMI Peripheral Interconnect (PI) Bus [12]. They share many common characteristics and are similar, both in performance metrics and in detailed operation. All three of them are high performance synchronous buses using central arbitration and master/slave interfaces. Although the detailed implementation of a modern on-chip bus system often involves complex bus protocols and precise timing behavior, a modular structural decomposition of these two systems is possible.

Basic cycle accurate operational semantics are implemented at the transaction level according to the specification, e.g. read/write, etc. The architecture of a simple bus system connecting two PEs is illustrated in Figure 6. Typical on-chip bus architectures often include a single bus through which other on-chip modules are connected to each other. Transfers occur between a master interface and a slave interface.

Various bus implementations often share common master/slave port assignments, such as address/data, request/grant, etc. Our first step is to design a generic master and slave interface which can be further extended by bus implementations with more advanced features. For example, our generic bus model only uses one set of arbitration signal pairs, i.e. request/grant, which is common among a number of bus implementations. However, there are more complex bus protocols calling for separate data and address arbitration. One way to accommodate this capability is to add a new set of arbitration signal ports. Another approach is to reuse the previous ports with different data fields indicating a different kind of arbitration type. Both approaches are possible within our framework. After we decomposed and analyzed these two bus architectures, we observed that they share a number of key building blocks. Thus our final bus model is based on several key modules described as follows:

• *Master Interface:* This is a module interpreting datapath communication events into data tokens compliant with bus protocols. As part of the Façade pattern, datapath communication events are represented in a unified format. It is the job of the master interface to translate these general communication operations into application specific tokens understood by the hierarchical communication channels.

• *Slave Interface:* This is a module which can translate received data tokens back into datapath communication events. More ad-
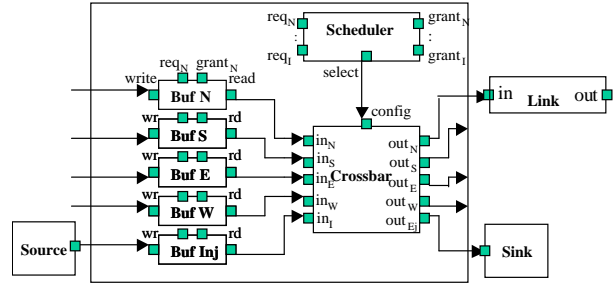
vanced slave interfaces can initiate bus transfers by themselves, such as in the case of split-transaction buses. Usually it needs to store incoming data tokens in FIFO-like buffers. The major difference between the AMBA interface and the CoreConnect interface is the support of deep address pipelining and unaligned byte. As shown in Figure 6, both PE_0 and PE_1 are equipped with instances of the Master Interface and the Slave Interface. Both Master Interface and Slave Interface are child classes of *interface* in Figure 3.

• *Bus Control Unit/Arbiter:* This unit arbitrates the ownership of the communication resource (physical backplane) with the request/grant port convention. Usually there can be only one owner of the bus and the arbitration policy can be flexible, such as round-robin, fixed-priority, etc. It is a child class of *resource scheduler* in Figure 3. In comparison to the AMBA bus, the CoreConnect bus supports up to four levels of priority by having extra priority signals.

• *Physical backplane:* This is a communication channel which transmits data between different bus masters and slave interfaces. The backplane is connected to all master and slave interfaces, but at any time, there is at most one enabled connection between the master and slave interfaces. It is a child class of *link* in Figure 3. CoreConnect supports up to 64-bit address bus while AMBA only supports 32-bit address bus. Both of the protocols support up to 256-bit data bus implementations.

• *Buffers:* In most circumstances, bus interfaces send and receive tokens from and into buffers. In our bus models, we only use buffers with the FIFO policy.

B. Modeling RAW - An On-Chip Packet-Switching Network

An on-chip packet-switching network appears radically different from the bus system modeling exercise described in Section V-A. Instead of establishing dedicated circuits from the source to the end, packet-switching networks transport data from one terminal to another in small formatted data chunks called packets. Performance of such a packet-switching network can be measured by its throughput and latency. Several interrelated design factors exist in the design of packet-switching networks, such as buffering, routing algorithm, etc. Our methodology is able to expose all these design factors directly to the users of the modeling and simulation environment by supporting a wide range of design choices such as wormhole, virtual channel and central buffer based systems.

As a point-to-point pipelined 2D mesh network, the RAW [4] network is highly modular and flexible, facilitating our decomposition effort. Figure 7 shows the interconnection between instantiated modules in PtolemyII and LSE for a simplified router architecture in RAW. Large boxes correspond to modules and small shaded boxes represent ports. Each module shares significant commonality with its counterpart in the bus model which is demonstrated as follows.

- *Router interface:* This interface serializes communication events emitted by the datapath into packets recognized by the routers in the OCA. Functionally the router interface can be decomposed into two separate parts, *Send Interface* and *Receive Interface*, which share common functions with the *Master Interface* and *Slave Interface* described earlier in the bus example. Here we implement a generic sending and a receiving interface class. The *Router Interface* is the composition of these two interfaces. In Figure 7, "source" and "sink" are instances of these two interfaces.
- *Allocator (Scheduler):* This child class of *resource scheduler* in Figure 3 implements almost exactly the same functionality as the arbiter in the bus architecture, except for the fact that in the bus example, there are only one or two resources available, while usually the allocator here decides for multiple resources. Both of them nonetheless share the same request/grant style interface. We are able to successfully model a wide range of variant allocator designs described in the literature [13], [14], including a RAW round-robin allocator, by inheriting the *resource scheduler* class and modifying only a few algorithmic methods defined in the superclass.
- *Crossbar:* This is a space-division parameterized switch and a child class of *link* in Figure 3. In the RAW case it is a 5×5 crossbar. It is very similar to the bus backplane employed in the bus example both in term of interface and functionality. The only difference is that while there is only one input port available for the bus backplane at one time, the crossbar can have multiple input/output ports enabled at the same time. In our hierarchical class diagram, they are both sub-classes of *DuplexLink* which encapsulates the input-output relationship of a duplex link, i.e. a conceptual 2×2 crossbar.
- *Buffers:* Buffering is widely used in both bus systems and RAW-like networks. In our modeling process, we are able to freely reuse the buffer module developed in the bus exercise. Even though the RAW network utilizes a rather straightforward buffering scheme, we are able to employ more complex buffering schemes such as central buffering and multi-queue buffering by reusing and then specializing the existing buffer module. For example, a subclass of a *central pool* buffer could be employed as a routing table that store established routes at each router. Furthermore, buffer organization schemes such as input queuing, output queuing or combined input/output queuing [15], [16] could be tried out by simply instantiating existing buffer modules and manipulating their interconnections with other switch modules.

Summary. The modeling process demonstrates that the RAW router modeling helps us reexamine modules with similar behavior used in the previous bus models. After the extraction of commonality based on interface, attribute and functionality analysis, a generalization process leads us to devise a more abstract model which encapsulates key common interfaces, attributes and functionalities without loss of generality. At the same time, this commonality extraction process also helps us find the "discriminating" features which distinguish one from another. Finally specific functionality can be achieved by inheriting this reusable root model and adding or modifying new functionalities and attributes within a specific application context.

VI. Toolset Evaluation

In this section, we discuss the basic criteria we have used to evaluate the proposed OCA modeling methodology.

A. Design Space Exploration

As a proof of concept we have implemented two on-chip bus systems and one on-chip packet-switching network within two different modular modeling and simulation environments. Our experience shows that both of them can be decomposed and modeled as a collection of communicating software modules. Further, a cycle-accurate system model and subsequent performance analysis is also achievable through the composition of performance models of individual modules.

In our methodology, design space exploration is both explicit and trustworthy. Because each PE is exposed to a uniform interface when communicating to other PEs via the OCA, designers have choices over the types of communication mechanism for the OCA, e.g. buses vs. packet-switching networks. On the other hand, inside each OCA, module-wise design choices can be implemented and evaluated quickly through explicit parameterization and specialization of individual modules.

B. Reusability Analysis

Through analyzing the structural and behavioral similarities of the above OCA schemes, we take advantage of the commonalities across the board to design a tree-like inheritance OCA module structure. As observed in our modeling exercises on on-chip bus architectures and packet switching networks, these commonalities can greatly simplify the process of future design once a fairly complete module library is ready. "White-box" reuse boosts productivity by directly reusing the code previously developed and gives the new model designers freedom to add and modify new functionalities to the enhanced model.

We measure the reusability and productivity through code size and development time comparison. Figure 8 illustrates this in a hierarchical way in the context of developing different types of bus slave interfaces. It is worth noticing that when we developed the AMBA slave interface, a large amount of code and knowledge reuse enabled us to finish the modeling job much quicker than usually required. As our experience shows, adopting the reusable class hierarchy can greatly reduce the development time, thus reducing the length of the entire design cycle and the overall SoC design cost.
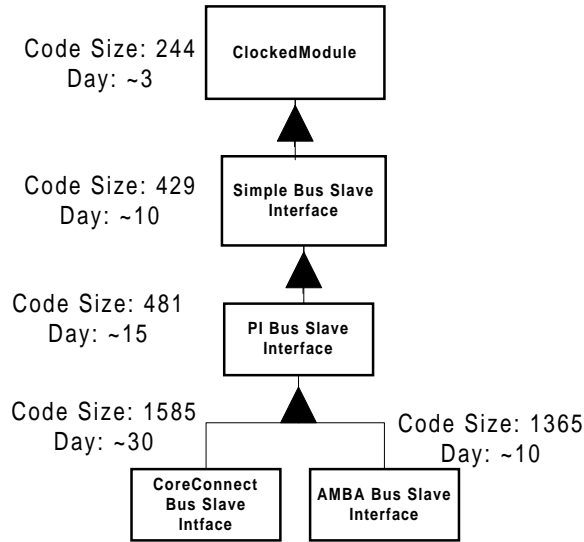
Fig. 8. Comparison of code size in lines of code (LoC) and development time

Table 1
Comparison of average simulation speed and development effort

Models	Simulation Speed (request/sec)	Code Size (lines)	Development Time (days)
PI-Bus Control Unit (VHDL)	30K	1008	~ 100 (estimated)
PI-Bus Control Unit (PtolemyII)	5K	445	~ 30
Bus Arbiter (LSE)	72K	475	~ 50

C. Trade-off Capabilities in Modeling

Different levels of abstractions allow efficient trade-off between model accuracy and system simulation speed. For the functional simulation of the OCA "ISA", we abstract away the low-level protocol specific details of the implementation of the OCA. The usage of a simplified concurrency model such as PRAM [17]can speed up the system simulation significantly.

This enables designers to quickly evaluate the multi-PE computation environment with a less accurate OCA model in the very early stages of design.

In the next step, cycle-accurate models are constructed corresponding to the hardware architecture of the chosen OCA that can reflect detailed timing behavior. However, within each module, there also exist different levels of granularity. For example, in the implementation of a variant of high speed symmetric crossbar arbiter/scheduler [18], the average latency of arbitration for a crossbar arbiter is almost constant if the input load (i.e. the number of request in progress) is less than half of the arbiter capacity, but it depends heavily on the input load after the load exceeds half of the capacity. To accurately capture this kind of input-sensitive dynamic timing behavior, the model needs to incorporate the empirically observed implementation behavior in calculating the realistic arbitration latency. This can be easily achieved by our modeling process. However under the assumption that the probability of the arbiter being overloaded is relatively small, constant arbitration latency is acceptable for a less accurate, data independent but more efficient model and modeling is made much simpler.

Table 1 gives the simulation results we obtained in the process of developing a PI-Bus Bus Control Unit (arbiter) within the two modeling environments described in Section IV. We have compared our simulation results with a reference implementation provided by the VHDL design toolkit under a uniform random stimulus. We simulated our models on a Pentium

III 800GHz machine running Linux. On average the simulation speed of LSE model is more than twice as fast as the VHDL model. Within LSE, even with a rather complex OCA model such as a 4×4 packet switching network with flow control, the simulation speed of the OCA model is able to match the simulation speed of multiple PE models, leading to a seamlessly integrated multi-PE simulation environment. We believe that LSE's performance advantage comes from the efficient module scheduling kernel. Also, LSE is significantly faster than Ptolemy II model due to benefits of low level specification language overhead (C vs. Java) and compiled code simulation.

D. Flexibility and Extensibility

Our methodology can accommodate unorthodox OCA candidates. Since OCA design is still an emerging field where the appearance of novel ideas is the norm, we have invested significant effort to ensure that our methodology is flexible enough to test and evaluate novel ideas. In addition to the successful modeling of traditional bus architectures and crossbar on-chip packet switching networks, we have also successfully modeled a shared central buffer [19] based on-chip networks by adding only two more modules. We believe the following key design characteristics make our toolset flexible and easy to extend to new paradigms:

- New functionality can be defined as mere enhancements to existing modules through inheritance. New interface ports can be added, semantics can be changed, parameters can be tuned, etc.
- If the above procedure is not sufficient, a complete new module can be written and inserted into the hierarchy.

VII. Related Work

Rowson *et al.* [7] proposed a system design methodology which separates behavior and communication based on the observation that systems are composed of modules which are communicating with each other. Object oriented analysis and modeling has been used in hardware-software co-design and synthesis [20], [21]. Our work draws upon the methodology of separating PE architecture and OCA while integrating a reusable

class hierarchy to classify various kind of OCA function modules based on object oriented analysis. Based on a modular modeling infrastructure encompassing both PEs and the OCA, our framework focus on the process of analyzing and deriving a library of reusable components for a wide range of architectural candidates of OCA. Further, our design flow targets the early stages of SoC design to facilitate design space exploration.

SystemC [22], [23] is a general digital synchronous modeling and design framework using C++ system-level models. Using some basic primitives, SystemC 2.0 provides a mixed-level abstract model for inter-function-unit communication, providing an up to a 100x simulation speedup versus traditional RTL models. A new design unit, interface, is provided by SystemC to enable the separate design of system communication models and system functionality models. Using a built-in simulation mechanism, the interface is capable of translating communication between different levels of communication abstractions. In this context, designers are interested in refining the communication mechanism with a progressively detailed model thanks to the seamless integration of the interface units. Given the fact that SystemC is a design language based on a C++ class library, we believe that our proposed methodology can also be easily applied to a SystemC based design process.

VIII. Conclusions and Future Work

In this paper we show that a systematic and effective architectural exploration of diverse on-chip communication architectures is possible based upon a limited set of OCA functional primitives, a disciplined hierarchical module classification scheme based on an object-oriented domain and reusability analysis of OCA models, and a modular modeling/simulation environment. Our experience in developing this library and associated tools demonstrates the effectiveness and flexibility of this methodology. This framework enables designers to make a variety of design choices in the early stages of OCA design with different degrees of modeling granularity. Fast and accurate performance results help designers to make comparisons between different design options. Modeling accuracy and speed trade-offs can be made throughout the design process. In future work we will try to identify a benchmarking approach for evaluating OCAs and apply our methodology to SystemC [22] based designs.

Acknowledgements

References

[1] IBM Corp., "The CoreConnect^{TM} bus architecture", Technical White Paper, 1999.

[2] ARM Holdings PLC, "Advanced Microcontroller Bus Architecture (AMBA) specification rev 2.0", 2001.

[3] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli, "Addressing the system-on-a-chip interconnect woes through communication-based design", in *Proc. Design Automation Conference*, 2001.

[4] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal, "The Raw microprocessor: A computational fabric for software circuits and general-purpose programs", *IEEE MICRO*, vol. 22, no. 2, 2002.

[5] C. Valderrama, A. Changuel, and A. Jerraya, "Virtual prototyping for modular and flexible hardware-software systems", *Journal of Design Automation for Embedded Systems*, vol. 2, no. 3/4, 1997.

[6] John Davis II, Christopher Hylands, Bart Kienhuis, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, Jeff Tsay, Brian Vogel, and Yuhong Xiong, "Heterogeneous concurrent modeling and design in java", Tech. Rep. UCB/ERL M01/12, University of California at Berkeley, 2001.

[7] Alberto Sangiovanni-Vincentelli James A. Rowson, "Interface-based design", in *Proc. Design Automation Conference*, 1997.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Company, New York, NY, 1995.

[9] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall International Editions, New York, NY, 1991.

[10] Manish Vachharajani, Neil Vachharajani, David Penry, Jason Blome, and David August, "Architectural exploration with Liberty", Tech. Rep. Liberty-02-01, Liberty Research Group, Princeton University, June 2002.

[11] Jayadev Misra, "Distributed discrete-event simulation", *ACM Computing Surveys*, vol. 18, no. 1, pp. 39–65, 1986.

[12] Open Microprocessor Systems Initiative (OMI), "Peripheral Interconnect Bus (PI-Bus)", 1998.

[13] Pankaj Gupta and Nick McKeown, "Design and implementation of a fast crossbar scheduler", in *Proc. Hot Interconnects 6*, 1998.

[14] Yuval Tamir and Hsin-Chou Chi, "Symmetric crossbar arbiters for VLSI communication switches", *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 1, 1993.

[15] M. Karol, M. Hluchyj, and S. Morgan, "Input versus output queueing on a space-division packet switch", *IEEE Trans. on Communications*, vol. 35, no. 12, 1987.

[16] Shang-Tse Chuang, Ashish Goel, Nick McKeown, and Balaji Prabhakar, "Matching output queueing with a combined input output queued switch", in *Proc. IEEE Conf. Computer Communications (INFOCOM '99)*, 1999.

[17] Steven Fortune and James Wyllie, "Parallelism in random access machines", in *Proc. the Tenth Annual ACM Symposium on Theory of Computing*, 1978.

[18] James Hurt, Andrew May, Xiaohan Zhu, and Bill Lin, "Design and implementation of high-speed symmetric crossbar schedulers", in *ICC-99*, 1999.

[19] Manolis Katevenis, Panagiota Vatsolaki, and Aristides Efthymiou, "Pipelined memory shared buffer for VLSI switches", in *Proc. Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1995.

[20] Frank Vahid and Linus Tauro, "An object-oriented communication library for hardware-software codesign", in *Proc. International Workshop on Hardware/Software Co-Design(CODES '97)*, 1997.

[21] Martin Radetzki, *Synthesis of digital circuits from object-oriented specifications*, PhD thesis, Univ. Oldenburg, Germany, 2000.

[22] Open SystemC Initiative, "SystemC", http://systemc.org.

[23] Preeti Ranjan Panda, "Systemc-a modeling platform supporting multiple design abstractions", in *Proc. International Symposium on Systems Synthesis (ISSS '01)*, 2001.