Using A Communication Architecture Specification in an Application-driven Retargetable Prototyping Platform for Multiprocessing

Xinping Zhu   Sharad Malik
Department of Electrical Engineering,
Princeton University, Princeton, NJ 08544, USA

Abstract—
In multiprocessor based SoCs, optimizing the communication architecture is often as important, if not more important, than optimizing the computation architecture. While there are mature platforms and techniques for the modeling and evaluation of architectures of processing elements, the same is not true for the communication architectures. This paper presents an application-driven retargetable prototyping platform which fills this gap. This environment aims to facilitate the design exploration of the communication sub-system through application-level execution-driven simulations and quantitative analysis. First, we introduce an expressive communication architecture specification which gives the designers the freedom to choose and configure their custom interconnection schemes over a wide range of communication architectures, covering the spectrum from buses to packet switching networks. This, combined with a distributed application model, drives a modular modeling and simulation environment that permits detailed simulation of the communication (and computation) architectures at the application level. Through the case studies motivated by an embedded system application, we show that through simulations, critical system information such as timings and communication patterns can be obtained and evaluated. Consequently, system-level design choices regarding the communication architecture can be made with high confidence in the early stages of design. In addition to improving design quality, this methodology also results in significantly shortening design-time.

I. Introduction

Modern System-on-the-Chip (SoC) designs increasingly consist of on-chip distributed/parallel processing units. Driven by the mounting computational complexity of multimedia and network processing applications, together with the increasing necessity to produce high-quality, low non-recurring engineering (NRE) cost IC designs in a shorter time frame, system designers are pushing forward with multiple processing elements (PEs) based SoC designs. Several such designs have been proposed and implemented. RAW [16] is a scalable processing platform which implements a simple, parallel and tiled architecture. A typical RAW processor may contain 16 identical RISC PEs connected by a 4-by-4 mesh packet-switching network. IXP 2800 [7] is a high-end network processing unit which contains 16 fully-programmable micro-engines and one XScale processor as the system control processor. The PEs and shared memory clusters are connected via a high data-width and high bandwidth bus network. These two examples highlight the increasing importance of the communication architecture in the design of high performance parallel computing systems.

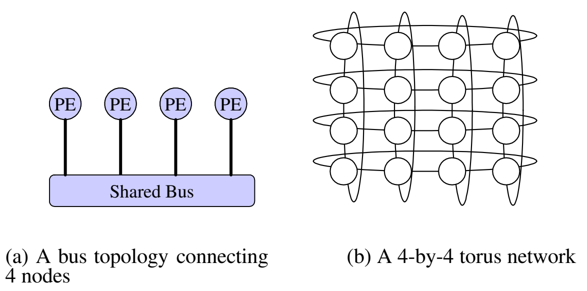
In traditional processor design flow, designers start with a cycle-accurate Instruction Set Simulator (ISS) to model and simulate single-threaded application codes. The PE model usually closely corresponds to a Register Transfer Language (RTL) based micro-architecture representation which contains PE micro-architectural components such as ALUs and register files, memories, etc. Using simulation results, designers refine the design through an iterative process. This traditional single-processor oriented modeling methodology does not completely address the new design challenges posed by the previous two examples. The complexity of these multiple-processor based processing platforms clearly exceeds the capability of modeling just at the single processor level.

One promising alternative is to simplify the modeling task by using a higher level of abstraction in hardware modeling. Instead of modeling traditional PE micro-architectural components such as ALUs, registers, we consider the entire PE as a single component and the On-Chip Communication Architecture (OCA) which connects these PEs as the first-class citizens in modeling a heterogeneous multiprocessor based SoC architecture. Just as there exist different types of PEs, e.g. VLIW vs. RISC, there also exist different types of OCAs, such as buses vs. packet-switching networks. Even within the same type of OCA, there are many design choices which are still to be made, such as the buffer size, the protocols, etc. These choices are usually not self-evident at the start of a design. Most of the time, making the choices involves a detailed understanding of the complex interplay between the application, PE micro-architecture and the OCA. System designers must consider the speed, bandwidth, power consumption and application constraints (such as real-time deadlines) when determining the type and parameters of the OCA to avoid latency penalties and achieve high system-level performance.

Our design environment focuses on the following two aspects of the prototyping and design exploration process: first, how to present the system architecture, especially the OCA part, in a uniform manner to the designers in the early stages of the design; second, how to enable designers to make informed choices regarding different aspects of the OCAs during the design space exploration. Clearly, one of the prerequisites of doing expressive OCA design space exploration is an on-chip communication architecture specification that is usable by retargetable design tools. This specification should be able to explicitly describe a multiprocessor based processing unit as what it is, namely a collection of PEs and the OCA. Furthermore, relevant design choices regarding the OCA should be explicit in this specification. Finally, the environment should be able to execute real-world applications through simulation to facilitate the evaluation of specific OCA choices.

The contributions of this paper are twofold. First, we provide a design environment targeting application driven design space exploration for cycle-accurate multiprocessor modeling, especially for the OCA part. This environment significantly extends previous work outlined in [20], [19] by incorporating a new modeling language and simulation infrastructure and adding a high-level language based distributed application model as input. Second, a retargetable communication architecture speci-

1530-1591/04 $20.00 (c) 2004 IEEE



(a) A bus topology connecting 4 nodes

(b) A 4-by-4 torus network

Fig. 1. Two examples of interconnection network topology

fication has been developed within this framework. The environment presents designers a wide range of OCA choices and enables them to select from these choices, using the specification, in the early stages of the design process. Compared to the traditional design flow, our design framework can significantly shrink the design turn-around time by supporting concurrent hardware and software modeling in an expressive and flexible manner.

The remainder of this paper is organized as follows. In Section II, we introduce some concepts and terminology of parallel systems and the OCAs we use in this paper. Then, in Section III we describe in detail the design flow and methodology we propose for both distributed application and on-chip communication architecture modeling. The design elements and internals of the proposed communication architecture description are also discussed here. This is followed by Section IV, where we focus on the implementation details of the simulation framework based on two distinct simulation infrastructures. In Section V, we present case studies exploring different aspects of our simulation environment: to explore the different OCA choices designers can make, to see different types of communication patterns the application can generate and how designers can make OCA design decisions after analyzing the simulation results. Some experimental results are presented and evaluated. Section VI then discusses prior related work and Section VII concludes the paper including a brief discussion of future directions.

II. System Architecture of the PE and the OCA

This section provides an overview of several relevant aspects of the PEs, the OCAs and the interactions between them.

We start with some commonly used OCAs - *viz.* buses and packet switched networks. A bus is usually composed of interfaces, arbiters and a shared backplane. The PEs are connected to the shared backplane through master and slave interfaces. The access to the backplane is arbitrated through the arbiters. Figure 1(a) shows a bus topology connecting 4 nodes. In comparison, a packet switched interconnection network consists of routers and links which are connected by a specific network topology. Figure 1(b) shows a 4-by-4 network with a torus topology. Both buses and packet-switching networks are widely used in building distributed computing systems today, both for the on-chip and off-chip cases. Examples are the RAW processor [16] for packet switched networks and the IXP2800 processor [7] for buses.

To give a high-level overview of how the OCA interfaces with the PEs, we trace the path of a message sent from PE A to B. Figure 2 illustrates this process when it is implemented in a packet-switched network. First, we assume a register-mapped OCA interface where PE instructions can access network data easily through registers. This is common in interconnection networks

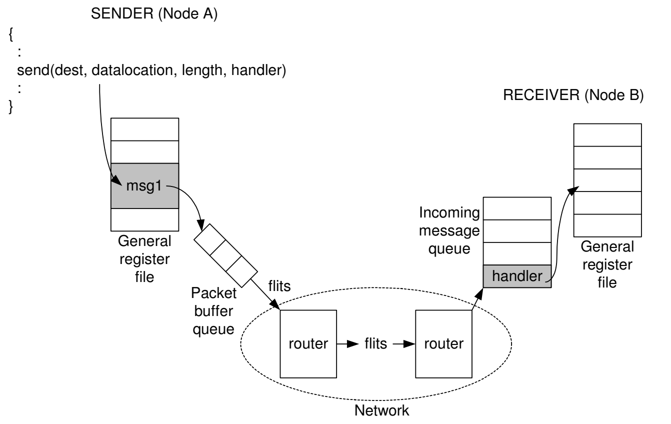


Fig. 2. A trace of what occurs in an interconnection network upon a *send* call

aggressively designed for low latency, such as the RAW Microprocessor [16]. The process starts with user programs communicating with each other using pre-defined library routines. Then, these library routines are compiled into PE instructions such as *send* and *receive*. Each instruction has a one-to-one correspondence with one OCA functional primitive defined in [20], e.g. *OCA_send, OCA_receive*. They are in turn understood by the OCA interface and translated by the OCA modules into specific OCA operation sequences. Each instruction has its own syntax and semantics. For example, a *send* instruction encodes the destination address, the data location and length, and the handler to be invoked when the message arrives at the destination. At PE A, when the *send* instruction issues, the OCA "send interface" extracts the data from the specified register, and if the network router (in the case of the bus, the master interface) is able to accept the data unit, injects it into the OCA right-away. If not, the data unit is buffered in a dedicated buffer queue which sits between the OCA interface and the router (or the bus backplane). On the receiving side, a *receive* instruction does no more than pull out the data from the incoming message queue and write it into specific register locations.

The above description points to the requirements imposed on the modeling environment to quantitatively analyze the system performance - it should be able to model each functional unit and each operation illustrated by Figure 2 faithfully.

III. Modeling Methodology

A. On-chip communication architecture modeling

In modeling the OCA, we apply the methodology proposed in [20]. We model an instance of an OCA as a collection of software primitives (executable models) which correspond to their hardware counterparts in the OCA, e.g. crossbars, buffers. In hardware, the assembly of these micro-architectural elements constitutes the OCA datapath. On the software side, these software modules communicate to each other through *messages* and *ports*. The control logic is implicitly embedded in the software module. In addition, these structural elements are organized in an object oriented class inheritance hierarchy. This hierarchy is implemented as a reusable and flexible module library which contains extensively tested and trustworthy components. Consequently, in designing a new type of OCA, designers need to examine and implement individual building blocks by either instantiating or extending available modules in the library. Furthermore, these new functional modules can be placed in the appropriate spot in the module library after testing. As a result,

Bus

```
CLUSTER my_bus;
NODE n0, n1, n2, n3;
n0.addr = 0; n1.addr = 1; n2.addr = 2; n3.addr = 3;
my_bus.data_width = 32;
my_bus.buffer_size = 64;
my_bus.protocol = "round_robin";
my_bus = bus(n1, n2, n3, n4);
```

Torus

```
CLUSTER my_net;
NODE n0, n1, n2, n3;
n0.addr = 0; n1.addr = 1; n2.addr = 2; n3.addr = 3;
my_net.vc_size = 1;
my_net.init_credit = 64;
my_net.routing = "dimension";
my_net = torus(n1, n2, n3, n4);
```

Fig. 3. Interconnection network description examples

an OCA design is simplified into a process of integrating these "plug and play" modules.

B. Template based retargetable communication architecture specification

As in most hardware designs, there are two basic parts which define the OCA: the control and the datapath. Currently in our modeling approach, the control of the OCA is implicitly defined in each OCA module. The OCA datapath is explicitly defined by a netlist of interconnections between OCA modules. The task of the communication architecture specification is to be able to abstract this netlist in a succinct manner and facilitate the design exploration process for the OCA part.

The specification is relatively short to ensure its ease of use. It focuses on the most important aspects of the communication architecture, namely the *topology*, *routing protocol* and *flow control algorithm*. First, we observe that most current OCAs use one of a set of common interconnection topologies in the datapath part, such as a shared bus, torus, mesh, hypercube or ring. In describing the netlist based topology, we start from several well-know interconnection topology templates, namely *bus*, *torus*, *mesh* and *hypercube*. An arbitrary topology is also possible through more low-level primitives. Then, we describe the control part through parameter setting. After we specify the custom routing protocol through several keywords, we are able to transform this specification into an internal modular representation for use in specific simulation platforms. Since we adopt a modular approach in modeling, it is relatively easy to extend our approach to support other type of interconnection topologies as well. Furthermore, even though this specification is developed with the aim of constructing machine specific SoC simulators, its semantics are not restricted by the fact we only use this for on-chip interconnection structures. Other work has shown that this methodology is also applicable to off-chip interconnection schemes [19].

The specification has C-like syntax. This syntax supports the basic elements of a script language: recognition of variable names and constants, declarations, expressions, and evaluation of function statements. It supports both implicit and explicit clustering. It supports generalized implicit clustering expressions which connect a list of nodes with a schema specified by a topology keyword. In addition, explicit connectivity graphs can be built through individual connection statements. The *cluster* can also be constructed hierarchically. Thus, a hierarchical bus or network can be specified by connecting *clusters*. The interconnections between the nodes are captured by an internal graph model after the specification is processed. The basic syntax of the specification language is listed as follows in the Backus Naur Form (BNF) format.

⟨ns_statement_list⟩ → ⟨ns_statement⟩ | ⟨ns_statement_list⟩
⟨ns_statement⟩ → ⟨ns_expression⟩ ⟨";"⟩ | ⟨ns_declaration⟩ ⟨";"⟩
⟨ns_operator⟩ → ⟨BUS⟩ | ⟨MESH⟩ | ⟨TORUS⟩ | ⟨RING⟩ |
⟨CUBE⟩ | ⟨CONNECT⟩
⟨ns_expression⟩ → ⟨ns_name⟩ ⟨"="⟩ ⟨ns_operator⟩ ⟨"("⟩
⟨"ns_name_list"⟩ ⟨")"⟩ |
⟨ns_name⟩ ⟨"="⟩ ⟨ns_operator⟩ ⟨"("⟩ ⟨INTEGER⟩ ⟨")"⟩ |
⟨ns_name⟩ ⟨"."⟩ ⟨ns_name⟩ ⟨"="⟩ ⟨LITERAL⟩
⟨ns_declaration⟩ → ⟨NODE⟩ ⟨ns_name⟩ |
⟨CLUSTER⟩ ⟨ns_name⟩ | ⟨NODE⟩ ⟨ns_name_list⟩

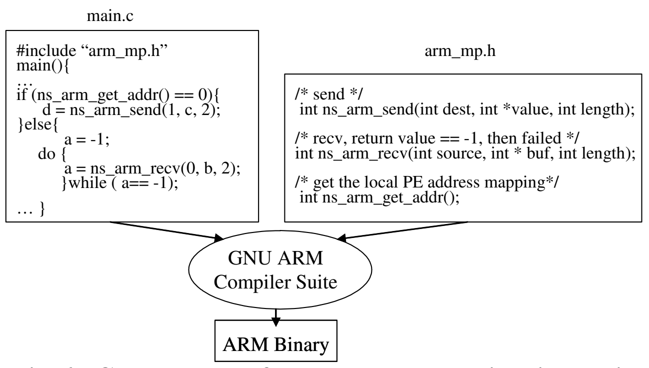
We will explain the detailed semantics of the specification language by walking through the two examples of interconnection schemes shown in Figure 3. On the left side, it shows a sample configuration of 4 nodes connected by a shared bus. On the right side, a torus packet-switching network connecting 4 nodes is shown. Most statements are C-like *assignment* statements or *declaration* statements. Only two kind of entities are allowed in the description, *cluster* and *node*. The *cluster* consists of a collection of *nodes*. Both *cluster* and *node* can be parameterized as shown in the examples. If *cluster* is parameterized, the parameter applies to all the nodes which are contained in the *cluster*. For example, the bitsize of data unit used in the interconnection scheme is defined through a parameter called *data_width*. Thus, all the nodes in the cluster have the same data unit size.

Furthermore, the torus network shown uses virtual channel(VC) scheduling [5] and a credit-based flow control scheme which are defined using the cluster-wise parameters *vc_size* and *init_credit*. In the *routing* parameter section, the torus example uses a dimension-order routing which is denoted by the string *"dimension"*, where we route along the y-axis first and then the x-axis. The bus example uses a round-robin arbitration algorithm denoted by the string *"round-robin"*. For each interconnection schema provided by the language, one or more routing or bus protocols are implemented. For example, the *bus* schema implements a parameterized shared bus model. This model supports separate data and control signals and request/grant style of centralized arbitration schemes. Several different arbitration algorithms are provided in addition to the round-robin one.

As we previously discussed, our current emphasis is on describing the datapath of the OCAs. Thus, the specification language at present only supports a number of pre-defined routing and bus protocols for each schema. These control protocols are embedded and implicitly defined in the implementation of the involved library modules (written in C/C++). This approach has its advantages and disadvantages. The specification itself is relatively short. Also, it gives the designers the ease of choosing several well-defined control protocols during OCA design exploration. However, this approach is not extensible enough to cope with novel routing protocols and emerging complex bus protocols. To accommodate this challenge, our future work will explore the possibility of including a synthesis methodology to automatically generate the control logic of the library modules facilitated by a formal description. Hence, the proposed control description, together with the existing datapath description, would provide a system-level OCA description which in turn drives the design space exploration.

C. Distributed application modeling

As our first step, we model the distributed application based on a number of simplified and explicit MPI-1 [10] like message passing library routine calls. Currently the library is based on high-level languages such as C/C++. To compile the program written with message passing function calls, we use the GNU compilation toolsuite. The toolsuite includes a target-processor compatible cross-compiler to compile C/C++ programs into target-processor native machine instructions. This process is helped by writing an assembly language based message passing library implementation for each target processor. For example, an ARM-V ISA [8] compatible PE has a dedicated implementation of message passing library routines with an interface defined in "arm_mp.h", which are in turn compiled



```
#include “arm_mp.h”
main(){
...
if (ns_arm_get_addr() == 0){
      d = ns_arm_send(1, c, 2);
}else{
          a = -1;
     do {
          a = ns_arm_recv(0, b, 2);
         }while ( a== -1);
... }
```

```
/* send */
 int ns_arm_send(int dest, int *value, int length);

/* recv, return value == -1, then failed */
int ns_arm_recv(int source, int * buf, int length);

/* get the local PE address mapping*/
 int ns_arm_get_addr();
```

Fig. 4. Code example for the message passing library implementation for ARM-V PEs

to ARM coprocessor instructions, e.g. *STC* and *LDC*. Some code examples are shown in Figure 4 to illustrate the flow of using message passing library routine calls. “main.c” is a C application program using the message passing library. In this program, if instantiated with two PEs, the process local to PE 0 sends a message to the process at PE 1 while the process local to PE 1 does a blocking receive of the message. Although the application model is straightforward, our experience shows that these two primitives together with several house-keeping function calls (such as PE addressing) are sufficient to model message passing based distributed applications.

During the application modeling, the particular choice of OCA for the multiprocessor is not yet obvious. Once the distributed program is written, it can be correctly executed by a multiprocessor model connected by any type of OCA. However, the performance will depend significantly on the specific choice made. The following section describes the total design flow that helps in making the right choice.

D. Total design flow

Figure 5 shows the system design flow of the prototyping platform. The blocks with shaded patterns represent the new software additions to the framework. The ovals represent the steps provided by the simulation infrastructure. On the top is the system architecture description which consists of the OCA model and the PE model. After the model parameters are checked against specified value ranges, the specification is fed into the simulator builders to construct a machine-specific executable simulator. As shown in this figure, we support two distinct simulator builders based on Liberty and SystemC. Details on this are discussed in the next section. The distributed application model is compiled into machine-specific binaries according to the steps outlined in Figure 4. Then the binaries are used by the simulator for execution driven simulation. After the simulation, relevant performance statistics, such as execution cycles and link usages, are gathered and analyzed. This is used in an iterative process in design space exploration. Since both the application model and the OCA model are straightforward and reusable, this leads to a significant shortening of the design turn-around time.

IV. The Simulation Framework

A. Simulation infrastructure

We have implemented the methodology described in Section III within two distinct modeling and simulation environments: Liberty Simulation Environment (LSE) [18] and SystemC [11]. The following taxonomy is used throughout the following sections. The basic building blocks of the modular modeling environment are called *modules*. Each *module* has *ports* through which *tokens* are sent to each other.

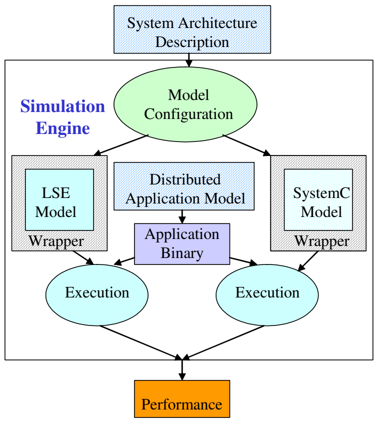


Fig. 5. Design, modeling and simulation process

Both environments support construction of concurrent executable models in a modular and hierarchical fashion. This enables us to read in our system architecture specification shown in Figure 3 as one of the inputs of the simulation engine. This system architecture specification contains both the retargetable communication architecture specification and the pointer to the pre-compiled PE model library. Then the simulator builder constructs a machine-specific executable model inside the simulation engine. Since both environments provide simulation kernels, after the execution of the simulator, cycle-accurate performance results can be obtained as output of the simulation engine.

B. Integrating the PE model in both infrastructures

To be able to execute pre-compiled binary executables in a cycle-accurate fashion within our hardware software co-simulation environments, we need to incorporate an accurate PE micro-architecture model in our platform. There exist two approaches, either we write from scratch two separate PE models in both SystemC and LSE, or we reuse a third-party PE model in our environment. Since we opt for an open platform where different types of PEs can be experimented using the retargetable system specification, the former approach would be too rigid and time-consuming to achieve our goal. Hence, the latter one becomes the natural choice. In the reuse process, we keep the interface simple and adopt related software patterns [6] to streamline the integration. In short, we provide an open interface which enables us to incorporate heterogeneous PE models in both SystemC and LSE.

As the first step, we choose a public-domain cycle-accurate ARM simulator described in [13] as an example PE model. As shown in Figure 5 minor addition and modification of the source code of the simulator is needed to export necessary interfacing library routines to both LSE and SystemC using the Wrapper Façade Pattern [6]. For each PE to be wrapped, a *clock_tick* method is provided to advance the PE simulator cycle by cycle. Furthermore, the integrated PE model needs to provide an input and output interface to the OCA written in LSE and SystemC. These interface are instances of the reusable modules defined in [20]. Each interface contains a queue structure to buffer and format the tokens which are to be sent out or stored. This queue based interfacing mechanism is proved to be general enough to be applied to both LSE and SystemC. Since only a small set of library routine calls, or the Façade, are exported to SystemC and LSE, the calling environment does not know the PE-specific variations and internal implementation details of each PE model, e.g. whether it is a RISC processor model or a VLIW processor model, or even the Model of Computation (MoC) used in the PE timing model. Thus, other types of PE simulators, such as a PowerPC timing model [13] could be incorporated as well to construct a model of a heterogeneous multi-processor based machine. This further give us the possibility to utilize commercial PE reference models or Intellectual Property (IP) models.

V. Case Studies

In our past work we have been able to evaluate the latency of an interconnection network by testing it under statistical traffic patterns [19]. However, the statistical approach has its limitations in both the validity and the applicability of predicting real-world performance of the interconnection network [9]. Hence, we perform the following case studies focusing on application-driven design space exploration for the communication architecture.

As a sampled design space for the OCA part, we select the following three different interconnection schemes:

- *TORUS*: a 3-by-3 torus network connecting 9 nodes
- *BUS*: a single shared bus connecting 9 nodes
- *FULL*: a fully enabled 9-by-9 crossbar connecting 9 nodes

Each node contains an ARM-V compatible PE and its related interfaces. Both the PE model and OCA models are cycle-accurate. We assume that the OCA and all the PEs share the same clock. For the sake of comparison, we set the parameters of the three cases as uniform as possible, such as the datawidth, transfer latency, etc. For the packet switched network schemes (*TORUS* and *FULL*), each connection is a bi-directional 32-bit pipelined link. The crossbar uses a 3-stage pipeline and is fully buffered. We utilize dimension-order routing as the torus routing protocol and worm-hole routing as the link scheduling algorithm [5]. Hop-by-hop credit-based flow control is used to prevent the buffer overrun problem. For our simple bus scheme (*BUS*), it takes 3 cycles to write one 32-bit word from the master interface to the slave interface if there is no contention. The arbitration is done by a one-cycle centralized round-robin arbiter.

Based on these machine configurations, we implement a high throughput crypto-processor running 3DES encryption/decryption applications. DES [15], the Data Encryption Standard, is one of the most popular methods of encryption. 3DES encryption means encrypting data three times instead of one. 3DES is now widely used in secure networking and other security applications. Our experiments show that our modeled machines are able to provide a maximum aggregate throughput of up to 1250 encryption/decryption tasks per second. In comparison, one ARM processor in our experiment setup can only achieve the throughput of 250 tasks per second. Each encryptions/decryption task consists of one 3DES encryption operation of a 1K byte plain-text and one 3DES decryption operation of the result cipher-text.

In order to illustrate the importance of the communication patterns exhibited by different applications, we select two separate parallel applications. The first one, *KEY_EXCHANGE*, distributes the required 3DES keys to all the PEs by performing one-to-one key exchange between the nodes. Another one, *3DES*, uses a 3-stage DES pipeline to encrypt/decrypt the text. The SystemC simulation results of the relative system throughput are shown in Table 1. Here we set the system throughput of the *BUS* configuration as the baseline performance and calculate the speedup over the *BUS* case. We observe that for *KEY_EXCHANGE*, a communication-intensive application, both *TORUS* and *FULL* are doing very well, achieving up to 4-5X speedup over the *BUS* configuration. After our detailed analysis, we conclude that this significant speedup is due to the fact that *KEY_EXCHANGE* results in a bus contention rate of up to 8.82% (out of a bus utilization rate of 15%) during the *BUS* simulation. This indicates that the bus is already saturated and the contention seriously degrades the entire system performance. Also, the results indicate that *TORUS* is able to achieve similar performance as the *FULL* configuration which offers the best possible interconnection scheme for a packet switching network. However, it is usually much easier and less costly to implement an on-chip network like *TORUS* compared to *FULL*. For *3DES*, a computation intensive application, it is shown that the differences of performance among these three cases are negligible. We believe that this is because *3DES* is relatively “silent” compared to *KEY_EXCHANGE*. It only causes a bus contention rate of less than 0.1% for the *BUS* case. From the analysis of the obtained results, we see that understanding the characteristics of actual application traffic patterns is critical to the selection of the appropriate OCA for real-world systems.

Table 1
Speedup comparison of different machine configurations

Configuration	KEY_EXCHANGE	3DES
BUS	1.0	1.0
TORUS	4.454	0.997
FULL	4.632	1.001

Table 2
Comparison of the two simulation platforms

Models	**Simulation Speed (Kcycles/sec)**	**Lines of Source Code**
LSE	15.8	3800
SystemC	10.2	1600

Table 2 gives the simulation speed and lines of source code we obtain in the process of simulating *3DES* on top of the *TORUS* machine configuration in both SystemC and LSE. For the lines of code, we only count additional software modules we need to write besides the SystemC and LSE frameworks. Both the application binary and the simulator builder are compiled using GNU g++ 3.2 with the compile flag “-O3 -fomit-frame-pointer”. The simulation is run on a PIII 1.1 GHz machine with Redhat Linux 8.0. In terms of measuring the performance, we count the system cycles advanced by all the PEs as the simulation speed. We see that LSE simulation is significantly faster than SystemC simulation. We attribute the speed advantage of LSE to its faster scheduling kernel and the benefits of the low level modeling language overhead (C++ vs. C) and compiled code simulation. But it is still quite slow when we consider the fact that the simple PE model we have integrated is capable of a simulation speed of 492.3K cycles/sec (about 4X slower after we count the 9X slow down of the multiprocessor simulation). We believe that the slow down is due to the fact that there still exists no efficient static scheduling technique to speed up the sequencing of local OCA operations in the OCA models, while there exists an efficient simulation kernel for the PE counterpart [13]. In addition, there exists a non-negligible overhead related to the interface between the PE and the OCA models. Finally, from lines of code developed we see that SystemC considerably surpasses LSE in terms of software productivity even though its simulation speed is slower.

A. Toolsuite evaluation and discussion

Once the designers have a correct application model and machine configuration ready, the design turn-around time is relatively short. For the 3DES application running on top of each of the previous three machines, the entire building process takes less than 10 minutes for both SystemC and LSE. The simulation could take longer time depending on the dataset size and number of iterations. In general, the process illustrated by Figure 5 is short enough to allow multiple iterations of model modification, either in the application model or in the machine structure model, in one day. The shortening of design turn-around time enables system architects to not only explore the design space more carefully but also to generate more reliable prototypes in the early stages of the process, thus enhancing their productivity.

VI. Related work

On-chip communication architecture modeling is a relatively new research area. [20] proposed a detailed methodology based on modular design and an object-oriented class hierarchy as part of the system-level design flow for programmable computing platforms [14]. This methodology was later applied in [19] to construct a fast power-performance simulator for on- and off-chip interconnection networks. Sharing similar goals in designing heterogeneous embedded systems, Metropolis [2] starts with a formal semantic foundation, and then refines the design to the lower levels. Instead, our approach is a bottom-up one. We start with analyzing various types of OCA instantiations and then derive a module library at different abstraction layers.

There exist extensive efforts in modeling and designing multiprocessors such as Network Processor Unit (NPU) based systems (both chip-level and board-level). StepNP [12] is a system-level design exploration platform for NPUs. It uses SystemC to model a multi-threaded architecture platform. Benini *et al.* [3] use SystemC and a GNU GDB enabled Instruction-Set Simulator (ISS) to model and simulate multiprocessor based architectures. This approach is limited by using a GDB based ISS while our approach utilizes a fully functional and timing PE model at the micro-architectural level. CovergenSC [4] is a SystemC based system-level modeling and verification tool provided by CoWare. Equipped with a fast SystemC simulation engine, it enables designers to rapidly create transaction level simulation models for multiprocessor systems (e.g. systems with integrated LISA 2.0 processor models) connected by complex on-chip buses. It also provides analysis tools to view system information such as bus and processor utilization. However, all these three platforms only cover a bus based interconnection model which connects all the processor cores together. In comparison, our platform is general enough to model a variety of different and complex interconnection choices driven by a retargetable communication architecture specification, both of which are lacking in the three previous approaches. Also our platform provides a flexible and open interface to integrate the needed PE model depending the required system type, simulation granularity and timing accuracy. Additionally, StepNP and CovergenSC focus on the transaction level and Benini *et al.* emphasize the ISA level. Finally Tensilica provides XTMP [17] which can integrate multiple C-callable Xtensa instruction simulators connected by customized interconnect modules. Currently XTMP only supports functional simulation. In comparison, we currently provide a library of cycle-accurate models and our approach is extensible to incorporate both transaction-level models and mixed-level models.

VII. Conclusions

This paper describes our approach in constructing a fast and retargetable modeling and simulation platform for multiprocessor design space exploration. This platform seeks to explore the choices of various types of OCAs while considering the interplay between the application, the PE micro-architecture and the OCA at various levels of granularity. To facilitate the process, we use a relatively short and easy-to-use retargetable communication architecture specification and a pre-defined PE model to synthesize a machine-specific high-performance cycle-accurate simulation engine. This enables the designer to obtain and analyze critical system information such as the timing, communication patterns and channel utilization for each configuration. Using a proof-of-concept embedded system application, we demonstrate that system architects can rely on the simulation results to make intelligent design choices regarding the OCA. Also the design cycle is significantly shortened, especially in the early stages of the design. Finally model re-usability is guaranteed and thus productivity of designers is enhanced.

Future directions include integrating the network configuration language with a flexible control description within a unified framework. Currently we release the source code of the library of stable modules in SystemC together with the simulation infrastructure for research usage [1], so that more interesting work on distributed computing platforms can be done.

References

[1] ARMn Simulator. http://www.ee.princeton.edu/~mescal/software.html.
[2] The Metropolis Project. http://www.gigascale.org/metropolis.
[3] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino. SystemC cosimulation and emulation of multiprocessor SoC designs. *IEEE Computer*, 36(4), 2003.
[4] CoWare Inc. CoWare ConvergenSC System Designer. http://www.coware.com.
[5] W. J. Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2), 1992.
[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, New York, NY, 1995.
[7] Intel Corp. Intel IXP2800 Network Processor. http://www.intel.com/design/network/products/npfamily/docs/ixp2800_docs%.htm.
[8] D. Jaggar and D. Seal. *ARM Architecture Reference Manual (2nd Edition)*. Addison-Wesley, 2000.
[9] W. B. Ligon III and U. Ramachandran. Toward a more realistic performance evaluation of interconnection networks. *IEEE Transactions on Parallel and Distributed Systems*, 8(7), 1997.
[10] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. Technical report, University of Tennessee, Knoxville, Tennessee, US, 1994.
[11] Open SystemC Initiative. SystemC. http://systemc.org.
[12] P. G. Paulin, C. Pilkinton, and E. Bensoudane. StepNP: A system-level exploration platform for network processors. *IEEE Design & Test Computers*, 19(6), 2002.
[13] W. Qin and S. Malik. Flexible and formal modeling of microprocessors with application to retargetable simulation. In *Proceedings of 2003 Design Automation and Test in Europe Conference (DATE 03)*, 2003.
[14] W. Qin, S. Rajagopalan, M. Vaccharajani, H. Wang, X. Zhu, D. August, K. Keutzer, S. Malik, and L.-S. Peh. Design tools for application specific embedded processors. In *Proceedings of Second International Workshop on Embedded Software (EMSOFT '02)*, 2002.
[15] B. Schneier. *Applied Cryptography*. John Wiley & Sons, New York, NY, 1996.
[16] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw Microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2), 2002.
[17] Tensilica, Inc. Xtensa Instruction Set Simulator and Xtensa Modeling Protocol. http://www.tensilica.com.
[18] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)*, 2002.
[19] H.-S. Wang, X. Zhu, L.-S. Peh, and S. Malik. Orion: A power-performance simulator for interconnection networks. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)*, 2002.
[20] X. Zhu and S. Malik. A hierarchical modeling framework for on-chip communication architectures. In *Proc. International Conference on Computer-Aided Design*, 2002.