StepNP: A System-Level Exploration Platform for Network Processors

Pierre G. Paulin, Chuck Pilkington, and Essaid Bensoudane
STMicroelectronics

The fast-changing communications market requires high-performance yet flexible network-processing platforms. StepNP is an exploratory network processor simulation environment for exploring applications, multiprocessor network-processing architectures, and SoC tools. Supporting model interaction, instrumentation, and analysis, the platform lets R&D teams easily add new processors, coprocessors, and interconnects.

THE CONTINUING GROWTH in network bandwidth and services, the need to adapt products to rapid market changes, and the introduction of new network protocols have created the need for a new breed of high-performance, flexible system-on-a-chip (SoC) design platforms. Emerging to meet this challenge is the network processor unit. An NPU is a SoC that includes a highly integrated set of programmable or hardwired accelerated engines, a memory subsystem, high-speed interconnect, and media interfaces to handle packet processing at wire speed.[1]

Programmable NPUs preserve customers' investments by letting them track ongoing specification changes.[2] By developing a programmable NPU as a reusable platform, network designers can amortize a significant design effort over a range of architecture derivatives. They can also meet technical challenges arising from a product's time-to-market constraints, as well as economic constraints arising from a product's brief in-market time.

StepNP is a system-level exploration platform for NPUs developed at STMicroelectronics. Its main components are a high-level multiprocessor-architecture simulation model; a network router application framework; and a SoC control, debugging, and analysis toolset. We focus here on the hardware architecture simulation platform, with emphasis on the transaction-level communication channel interface and our model interaction, instrumentation, and analysis approach.

Wire-speed packet forwarding

Packet forwarding over a network includes the following main tasks: header parsing, packet classification, lookup, computation, data manipulation, queue management, and control processing. Control processing usually takes place on a standard reduced-instruction-set-computing (RISC) processor linked to the NPU and is not the focus of this article.

Wire-speed packet forwarding, at rates often exceeding 1 Gbit per second, poses many more challenges than general-purpose data processing. In network processing, both memory capac-

0740-7475/02/$17.00 © 2002 IEEE

ity and bandwidth are extremely demanding. The interconnect between processors, memories, and coprocessors must support a very high, cost-effective, scalable bandwidth.[3,4]

A key aspect of efficient NPU hardware use is latency hiding. The most common latency-hiding approach is multithreading, which efficiently multiplexes a processing element's hardware. Multithreading lets the hardware process other streams while another thread waits for memory access or coprocessor execution. Most NPUs have separate register banks for different threads, with hardware units that schedule threads and swap them in one cycle. We call this function *hardware multithreading*.

StepNP overview

In developing the StepNP platform, we had several objectives. We wanted the platform to be

- a challenging internal driver for our existing embedded systems[5] and system design technology development,[6] as well as a driver for high-level multiprocessor platform methods under development;
- a vehicle for long-term research in multiprocessor architecture exploration, design tools, and methods;
- an open, easily accessible environment, built with public-domain components as much as possible; and
- a baseline from which designers can easily derive realistic NPU architectures.

Figure 1 shows the StepNP platform. It consists of three main frameworks: the application software development platform, the NPU-architecture simulation platform, and the SoC tools platform.

Application software platform

Our application software platform is a direct port of MIT's open source Click modular router framework for the rapid development of embedded routing-application software.[7] We chose it for its modularity, flexibility, and ease of reconfiguration. Figure 1a shows sample Click modules performing packet classification, discarding, stripping, and queuing. These modules can be linked together to quickly create a routing application. The StepNP architecture platform includes a network-address translation application that we developed using the Click framework. This application emulates a virtual host by capturing packets from the workstation's physical Ethernet connection, performing network address translation algorithms, and reinjecting the result on the Ethernet line.

Architecture platform

Commercial NPUs feature a wide range of architectural styles, from fully software programmable to almost completely hardwired. For the StepNP initial platform, or *base platform*, we use a fully programmable architecture based on standard RISC processors, and a simple interconnect. The base platform allows easy plug-and-play replacement with more specialized processors, coprocessors, and interconnect.

We wanted the StepNP modeling approach to support easy integration of debugging and analysis tools, as well as top-down verification methodologies. In particular, we concentrated on the analysis of communication properties between NPU building blocks, including interface behavior, data throughput, and latencies. As Figure 1b shows, the base platform includes three main IP component types:

- *processor engines* (RISC-based architecture models augmented with configurable hardware multithreading capability and a simple *n*-stage pipeline),
- a *network-on-a-chip communication channel* (in this case, a high-level split-transaction channel model), and
- *specialized coprocessors* (in this case, a semaphore engine as a demonstration coprocessor).

The architecture platform includes a coprocessor I/O port. We also integrated a model of a network-packet search engine coprocessor in the architecture platform; this model is for STMicroelectronics' internal use and is not part of the base platform.

SoC tools platform

The SoC tools platform provides two main categories of tools:

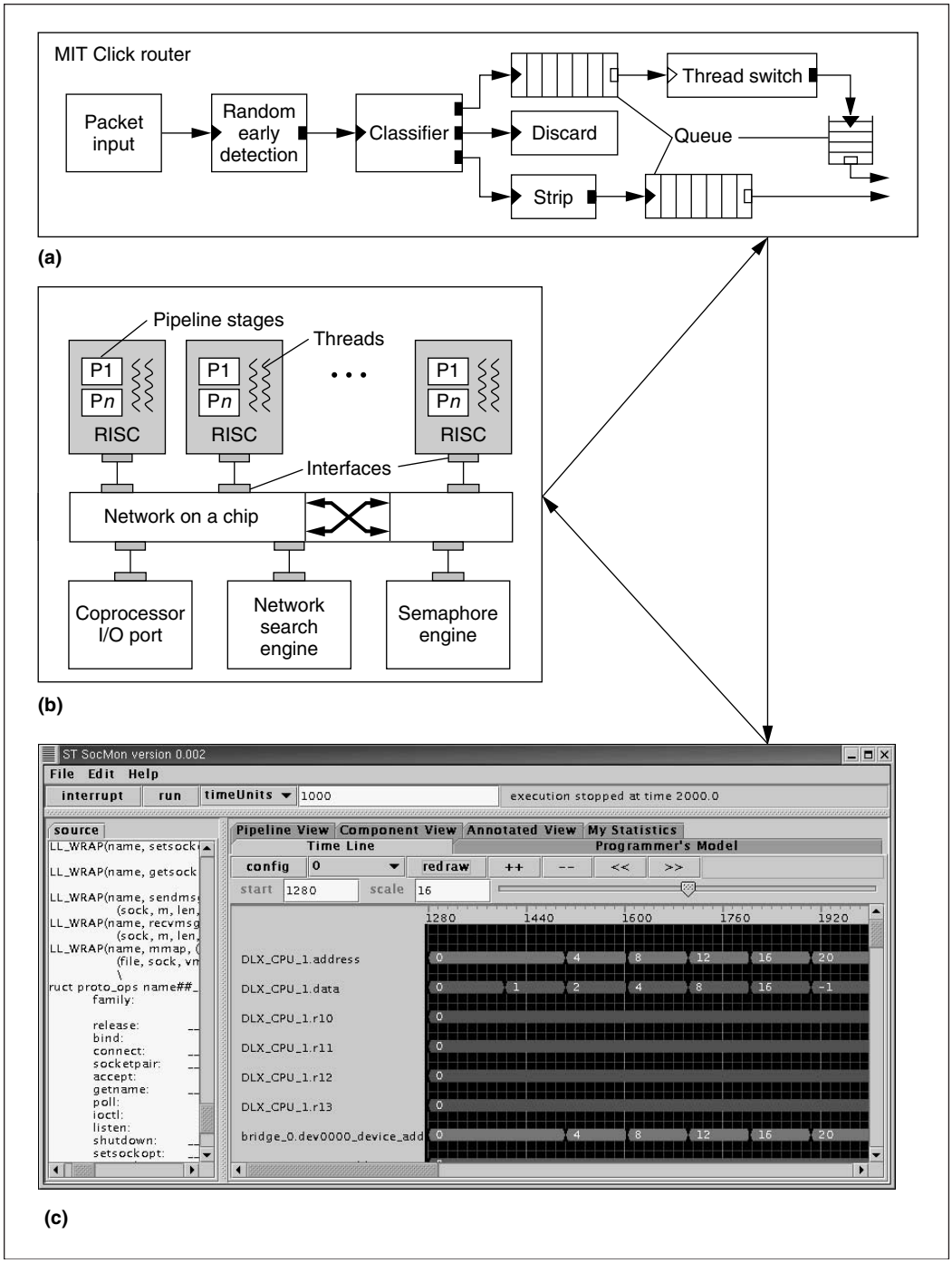

Figure 1. StepNP overview: application software development platform (a), NPU-architecture simulation platform (b), and SoC tools platform (c).

- *component level* (embedded-software-development tools for the individual processors used in StepNP, including a C compiler, an instruction-set simulator, and a source-level debugger), and
- *SoC level* (tools for controlling, debugging, and analyzing the top-level multiprocessor architecture).

The SoC-level tools allow visualization and control of the model's execution from various perspectives (programming, logical, temporal,

spatial, and user-defined) for a range of abstraction levels (functional, transaction, and cycle-based). The programming perspective is a multiprocessor version of a conventional source-level debugger. The logical perspective lets the user track a single packet's processing logically, even though the processing is distributed over multiple processor, interconnect, and hardware resources. The logical view is a natural extension of the programming perspective.

The temporal perspective lets the user visualize parallel activities on a time line. The time line can represent various abstraction levels—for example, the name of the top-level C function running on a processor at a given time, or the signal value on a bus.

The spatial perspective allows event tracking in a hierarchical block diagram. StepNP automatically extracts the graphical representation of the hierarchy from the model via an introspective application programming interface described later. Finally, well-defined APIs that allow easy connection of scripting languages or graphical environments enable user-defined perspectives, a key requirement for STMicroelectronics' customers.

StepNP architecture simulation platform

Now we take a closer look at the architecture simulation platform and its three IP component types.

Modeling language

We chose SystemC 2.0 (http://www.systemc.org), with its wide range of modeling abstraction capability, as the StepNP architecture platform's main modeling language. Where appropriate, we also included more-specialized languages—for example, Tool Command Language (Tcl) for user scripts and Java/Forte for graphical user interfaces and user-defined extensions.

Multithreaded processor model

The base StepNP architecture platform includes the public-domain models of the ARM v4 and the PowerPC (versions 603, 603a, and 604) instruction-set architectures (http://www.fsf.org) and the Stanford DLX processor model.[8] Our approach is to encapsulate functional instruction-set models into a SystemC wrapper. The encapsulation produces a cycle-based model implementing a configurable hardware multithreading capability and a simple *n*-stage pipeline.

Each thread in the SystemC processor wrapper calls the instruction-set simulator (ISS) to implement the thread instructions. The wrapper interleaves the sequencing and timing of these calls to model the execution of a hardware-multithreaded processor. The ISS returns memory reference operations to the wrapper for implementation in SystemC. The wrapper communicates with the rest of the StepNP platform via the SystemC Open Core Protocol (SOCP) communication channel interface described later.

For research teams interested in network-specific instruction-set optimizations, we are planning to integrate Xtensa configurable processor models from Tensilica (http://www.tensilica.com) and models from the LisaTek instruction-set simulation model generator toolset (http://www.lisatek.com). These models will help researchers explore a wide range of possible instruction-set architectures.

Coprocessors

Users can easily integrate new coprocessor models into StepNP, using the SOCP communication channel interface. Currently, the base platform includes a semaphore engine model that allows concurrency management in the processor array.

The network search engine is a pipelined SRAM-based solution to the lookup and classification problems. It also aids virtual private networking. This solution uses significantly lower cost and power than a component based on content-addressable memory with built-in table management. The StepNP platform serves as a validation environment for this search engine during the architecture design phase. Later, it serves as a reference platform for customers.

Communication channel

The interconnect model developed in the base StepNP platform is a simple functional model supporting split-transaction communi-

cation. We gave the communication channel's interface definition particular attention, and we describe it in the next section.

We are planning to integrate other interconnect technologies into StepNP: STMicroelectronics' Octagon[4] network on a chip and LIP6's (Laboratoire d'Informatique de Paris 6) SPIN network on a chip (http://www-asim.lip6.fr/cerme/presentations_28_01.html).

Communication channel interface

Our goal was a standardized interface that would let users plug in and play SoC IP components at various abstraction levels.

Requirements

The following requirements motivated the StepNP communication channel interface's design:

- The interface must operate at the functional and transaction levels. The interface should contain no bit-level signals, polarities, clock cycles, or detailed timing. This requirement does not preclude an adapter that, for example, maps the interface to a cycle-accurate internal model.
- The interface-modeling approach should use SystemC 2.0 constructs in a manner as close as possible to their original intent. In other words, the communication between master, channel, and slave should use the SystemC 2.0 port, interface, and channel methodology.
- The interface should make no assumptions about the underlying interconnect architecture. It must support anything from a simple point-to-point connection to an arbitrarily complex, multilevel network on a chip.
- The interface must support split transactions and should not assume that requests and responses are atomic.
- It must support multithreaded masters and slaves.
- If possible, the interface should be compatible with existing interfaces designed for IP reuse.

SOCP channel interface

We developed the SOCP channel interface model using the SystemC 2.0 language. The model follows the same high-level semantics as the Open Core Protocol (OCP) and the Virtual Component Interface (VCI) but has no notion of signals or detailed timing. For transaction-level modeling, these standards (http://www.ocpip.org and http://www.vsi.org) can be considered functionally identical, and we refer to them interchangeably. Our modeling approach has the following advantages:

- The SOCP channel interface model can inherit semantics of parameters and behavior largely from the OCP/VCI specification.
- The StepNP user can refine the SOCP to transform it to an OCP/VCI lower-level interface or to other interconnect implementations, such as industry-standard buses or complex networks on chips.
- The channel interface model achieves higher simulation speeds than OCP/VCI or bus-level channel implementations because of its higher abstraction level.

Master-slave interface. The OCP standard uses unidirectional connections between components, with one side driving the signals and the other receiving. For the SOCP interface, we model this unidirectional connection in SystemC, using a port and an interface. The driving end sends data to the receiving end through the port. The receiving end implements the port interface.

OCP has one unidirectional interface for sending requests and another unidirectional interface for receiving responses. It allows single- or split-transaction channel implementations. Requests can be pipelined, and responses can come back out of order in some situations.

The SOCP interface used by the master for sending a request is expressed in SystemC as

```
virtual void putReq(InterfaceData
&data)=0;
```

The interface used by the slave for its response is similarly expressed:

```
virtual void putRsp(InterfaceData
&data)=0;
```

Table 1. OCP signal definitions.

Data field	Description
MConnID	A connection identifier—a number that uniquely identifies a component.
MThreadID	A thread identifier that assists with multithreaded IP components, memory consistency, and out-of-order responses.
MthreadBusy	A hint to the slave IP component indicating which master threads are busy and cannot accept responses.
MAddrSpace	Allows access to different address spaces in the slave such as registers and memory.
MCmd	Identifies the nature of the requested transaction. Values denote commands such as read, write, and read exclusive.
MBurst	Assists in the flow of burst data across the interface.
MAddr, MAddrPtr	The data's address. We extend this parameter in the SOCP interface to make it a pointer to an array of values as well as a scalar value. The array option is for burst transfers.
MData, MDataPtr	The data itself. We extend this in SOCP to make it a pointer to an array of values as well as a scalar. The array option is for burst transfers.
MByteEn, MByteEnPtr	Byte enables, for selective byte writes. We extend this in SOCP to make it a pointer to an array of values as well as a scalar. The array option is for burst transfers.
SResp	Slave response code, indicating whether data is correct or an error has occurred.
SThreadID	Pairs the slave's response with the thread in the master that made the request.
SThreadBusy	A hint to the master, indicating threads that are busy in the slave.

The SOCP allows a direct connection between master and slave, without an intervening channel. This capability is a good test of interface symmetry and correctness. An SOCP channel must implement both master and slave interfaces and supply ports with these interfaces.

Interface data. The data crossing the SOCP interface is essentially the same as that defined in OCP but is expressed in a C++ structure. However, we omitted OCP signals related to low-level handshaking of data across the interface because they are at a lower abstraction level. Table 1 lists the interface data with brief descriptions (the OCP specification gives more details). The table follows the convention that data the master sends to the slave starts with "M," and data the slave sends to the master starts with "S."

Extending OCP semantics for high-level modeling

Although we followed the base OCP and VCI semantics, the SOCP needed selective extensions for functional- and transaction-level modeling.

Burst transfers. Data crosses an OCP interface one word at a time. In SOCP, however, we also allow a complete burst transaction in one put request. We do this by specifying an additional length parameter with a value greater than one and allowing pointers for the address-, data-, and byte-enable parameters. We assume that the other parameters are constant for each data item of the transfer, except for the MBurst parameter, which needn't be set by the master. However, a cycle-accurate adapter on the other side of the interface can choose to feed the data into a lower-level internal model and generate the burst signal for each word as appropriate. The SOCP interface requires the response length to match the request length.

Blocking semantics. Calls across an SOCP interface can block the caller. Therefore, the caller should be a SystemC thread construct (SC_THREAD). If a slave IP component blocks a put request from a channel, however, the channel could be blocked until the request is serviced. Therefore, if channel blocking is an issue, slave IP components should avoid blocking a put request (by using the SThreadBusy back-pressure mechanism or by buffering requests). The same applies to a response from the slave (or channel) to the master.

It is possible to implement both master and

slave in a purely functional manner. If a purely functional channel model is used, some mechanism should be provided in the channel to schedule new threads; otherwise, one master will dominate the simulation, and no other threads will run.

Adapters and refinement. The SOCP supports refinement on either side of the interface and plug-and-play component replacement at various abstraction levels. For example, a purely functional master could use a pin- and cycle-accurate model on the other side of the SOCP interface. Cycle-accurate models on both sides are also possible if put requests and responses occur on a word-by-word basis. The master could also set the burst parameters to the appropriate values for accurately modeling data-streaming across the interface.

IP components modeled in a functional style, with requests and responses referring to data blocks, should work with a cycle-accurate channel model. To feed the internal model, the channel model would need an adapter to translate the data in the requests into low-level signals. The adapter should translate low-level responses back into the format in which the master received them. For example, a request that specified a data block should have a response corresponding to that block. However, the channel adapter could feed the slave on a word-by-word basis, even if the master request was a block.

Profile support. The SOCP master-slave interfaces are template classes and can take user-specified address and data type parameters to support OCP-like profiles (basic, simple extensions, and complex extensions).

An IP component can choose to use a subset of the data (for example, only 11 bits of the address field), with the plug-and-play operation rules defined by the standard.

SOCP performance

To evaluate the SOCP modeling approach, we performed two tests. The nullModem test uses a simple functional master, which generates read and write requests, and a simple functional memory as the slave. The master plugs directly into the slave with no intervening channel. The funcTest has the same master and slave components as nullModem but introduces a simple functional implementation of an SOCP channel. The funcTest creates eight master components and eight slave components. The channel has a simple time-accounting model, in which transactions are delayed every *n* calls, for time *t* (*n* and *t* are channel parameters).

Table 2. Performance of functional SOCP channel.

Test	Read/write rate (millions per second) on Solaris Sun U80 at 450 MHz	Read/write rate (millions per second) on Linux PC (Pentium 3 at 800 MHz)
nullModem	2,000	1,745
funcTest TA1	248	1,530
funcTest TA10	903	1,530
funcTest TA100	1,220	1,530

Table 2 shows the results of these tests. The reads and writes referred to in the table are complete master-slave transactions, consisting of four atomic transactions (master to channel, channel to slave, slave response back to channel, and channel response back to master). We obtained these results using the public-domain SystemC 2.0 class library on a Solaris Sun workstation running Unix and a PC running Linux. In the table, "TA*n*" indicates that transaction-time accounting occurred every *n* transactions. On the Solaris platform, this parameter made a big difference in simulation speed (a factor of 5 between TA1 and TA100). Under Linux, this parameter had no effect on performance.

Distributed simulation using SOCP

A benefit of a well-defined communication interface is that it can serve as the basis for partitioning the SystemC simulation model over multiple workstations. To achieve this capability in StepNP, we implemented an SOCP channel called *dsim* that allows distributed simulation over a pool of workstations. The masters and slaves plug into the dsim channel without any changes. The channel implementation, however, is aware of the workstations involved in the simulation and of the components' addresses and identifiers. If a master sends a request to a local slave, the transaction com-

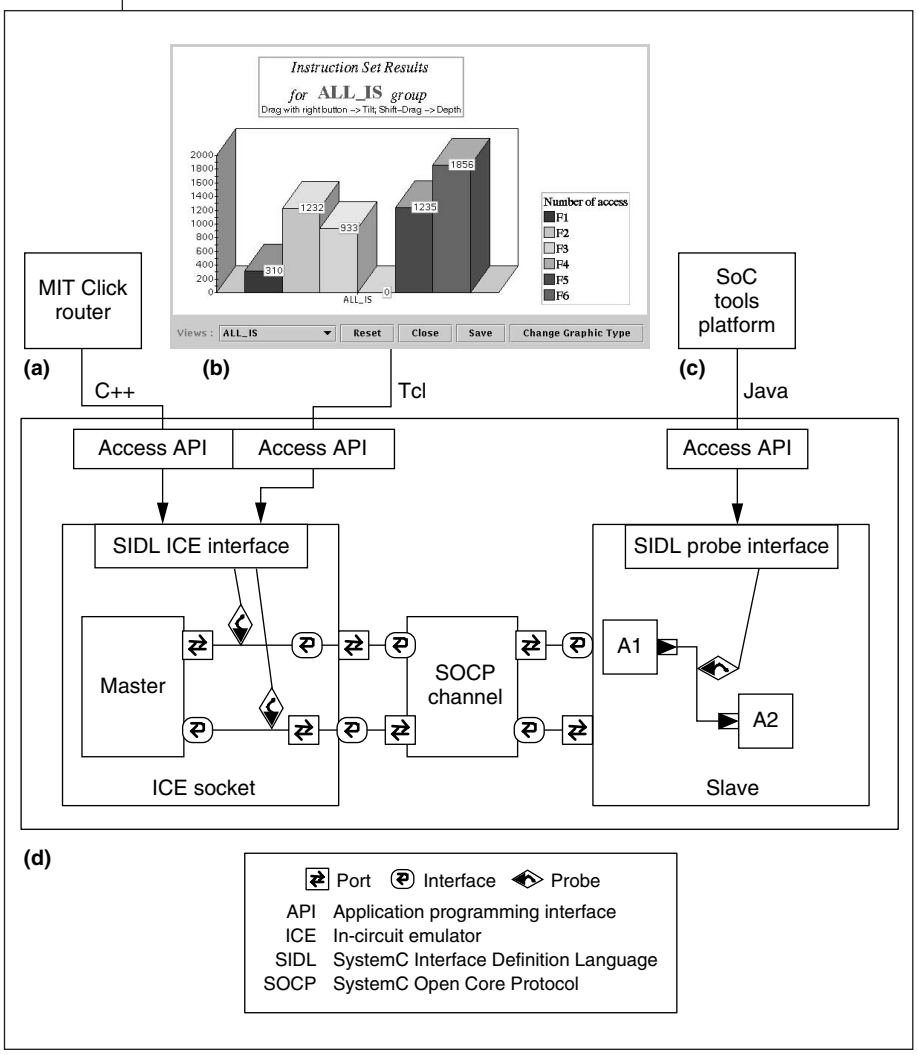

Figure 2. StepNP control-and-view methodology: application software development platform (a), performance analysis tool (b), SoC tools platform (c), and SystemC model (d).

pletes in much the same way as the functional channel. However, if the slave is on another workstation, the channel implementation packages the request and sends it over the network to the destination slave. In this approach, either the communication rate or the slowest distributed SystemC model component limits the overall simulation speed.

We measured communication rates of about 30 kHz using the dsim channel implemented with the transmission-control and Internet protocols (TCP/IP) over a standard 100-Mbit Ethernet line. In cooperation with researchers at the University of Montreal, we are working on optimizing the implementation for higher-performance interconnects, using the Myrinet network, which runs an order of magnitude faster.

Model control-and-view support framework

Designers and programmers using StepNP need to debug, verify, understand, initialize, and measure the platform model's execution. This diversity of uses requires a robust methodology for controlling and viewing models.

In most cases, the control-and-view components are parts of a process separate from the model itself and are often implemented in another language. For example, scripting languages such as Tcl are often used to automate verification. A typical verification scenario could have a script that populates the routing table in the model with known values and then injects packets into a network device in the model. The script would then start the model execution and examine the emitted packet for correct header values and processing latency.

Given the control-and-view requirements, StepNP supports the introspective approach illustrated in Figure 2. An external control-and-view component—for example, the SoC tools platform—connects to the model's access API. This component can query the model's structure, discover components the model uses, and discover the supported interfaces to these components. In Figure 2, the external SoC tool component has discovered a SystemC probed signal between two slave model subcomponents, A1 and A2. The StepNP control-and-view framework automates much of this process.

SystemC structural introspection

SystemC provides an API for traversing the model's structural hierarchy. The access API enhances this basic support and builds a structural representation of the model.

Low-level signals (such as objects of type sc_signal) and simple state representations can use a StepNP-supplied probe class. This probe class extends signals and state representation variables with functionality that connects to external control-and-view components. These components can use this functionality to discover the model's signals and probed state and to recover the time history as needed. They can also use the probe class for automating sc_trace control, for custom dumps, or for other functions.

SIDL interface

For software written using high-level SystemC 2.0 modeling facilities, automatically extracting state information and allowing control access are more difficult. Therefore, we developed an instrumentation methodology and an associated language called the SystemC Interface Definition Language. An SIDL interface allows external control-and-view components written in various languages to access a SystemC model object implementing this interface.

SIDL looks much like a pure virtual C++ class and is patterned after the sc_interface approach in SystemC. For example, the SIDL interface to a simple counter could be

```
class CounterCandV {
  public:
    virtual int getCount() = 0;
    virtual int setCount(int) = 0;
};
```

An SIDL compiler parses an SIDL header file and produces all the client-server glue. The glue on the server end connects an object implementing this interface to the low-level access API. The compiler produces the client-end glue in the desired language (for example, Java, C++, or Tcl). The function parameters can be basic types (integers, floats, strings, and so forth), structures, or vector containers of these types. The SIDL compiler handles all marshaling and remote procedure call issues.

In the **`CounterCandV`** example, a counter in the SystemC model needs only to inherit the server instance of this class (**`CounterCandVServer`**) generated by the compiler and implement the **`getCount`** and **`setCount`** methods.

The client can call the access API in the server to discover all control-and-view interfaces and the names of these instances. For example, a client might find that the model supports a **`CounterCandV`** object, named **`counter0`**. The client can then create a **`CounterCandVClient`** object, supplying the name **`counter0`** to the constructor. The client can then call the **`getCount`** and **`setCount`** methods of this object, which transparently calls the **`getCount`** and **`setCount`** methods in the corresponding SystemC object.

At one level, SIDL looks like a distributed object model such as the Common Object Request Broker Architecture (CORBA). However, SIDL is more restricted in scope than CORBA, follows the interface style of SystemC, and is integrated in the SystemC environment.

Instrumentation of an SOCP socket with SIDL

It is possible to develop generic control-and-view interfaces for common master and slave components, such as processors and memories. However, the instrumentation of an SOCP interface is of particular interest because instrumentation tools developed for an SOCP interface can be used with any IP block.

An IP block with an SOCP interface can be plugged into an object whose function is analogous to an in-circuit emulator (ICE) socket, with no change of the channel object or the device under test, as illustrated for the master module in Figure 2. The abstract ICE socket can transparently pass master requests and slave responses. However, external software can monitor or generate the transactions using an SIDL interface. The ICE socket can also perform transaction recording and store the transactions in a trace file for viewing by standard CAD tools or, as Figure 2b shows, by more specialized SoC performance analysis tools such as STMicroelectronics' FlexPerf[5] and SysProbe.[6]

WITHIN THE NEXT YEAR, we plan to make the base StepNP platform available as a public-domain environment accessible to the research community. In this endeavor, STMicroelectronics

will partner with the Canadian Microelectronics Corporation to benefit from its experience in large-scale university interaction and infrastructure support (http://www.cmc.ca).

We also plan to integrate new processors, with instruction sets tuned to networking applications, and a selection of network-on-a-chip interconnects into StepNP. Finally, we will develop a more extensive set of networking applications, derived from the MIT Click modular-router environment. This set of applications will give STMicroelectronics and its research partners a valuable network-processing reference platform for developing new multiprocessor design and automation methodologies. ■

References

1. N. Shah, *Understanding Network Processors*, internal report, Dept. of Electrical Eng. and Computer Science, Univ. of California, Berkeley, 2001; http://www-cad.eecs.berkeley.edu/~niraj/papers/UnderstandingNPs.pdf.
2. P.G. Paulin, F. Karim, and P. Bromley, "Network Processors: A Perspective on Market Requirements, Processor Architectures and Embedded S/W Tools," *Proc. Design, Automation, and Test in Europe* (DATE 01), IEEE CS Press, Los Alamitos, Calif., 2001, pp. 420-429.
3. L. Benini and G. De Micheli, "Networks on Chip: A New SoC Paradigm," *Computer*, vol. 35, no. 1, Jan. 2002, pp. 70-72.
4. F. Karim et al., "On-Chip Communication Architecture for OC-768 Network Processors," *Proc. Design Automation Conf.* (DAC 01), ACM Press, New York, 2001, pp. 678-683.
5. P.G. Paulin and M. Santana, "FlexWare: A Retargetable Embedded-Software Development Environment," *IEEE Design & Test of Computers*, vol. 19, no. 4, July-Aug. 2002, pp. 59-69.
6. A. Clouard et al., "Towards Bridging the Gap between SoC Transactional and Cycle-Accurate Levels," *Proc. Design, Automation, and Test in Europe Designer Forum*, DATE Conf. Secretariat, Edinburgh, UK, 2002, pp. 22-29.
7. E. Kohler et al., "The Click Modular Router," *ACM Trans. Computer Systems*, vol. 18, no. 3, Aug. 2000, pp. 263-297.
8. J.L. Hennessy et al., *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, Calif., 1990.

Pierre G. Paulin is the director of the System-on-Chip Platform Automation Group at STMicroelectronics, Ottawa, Ontario, Canada. His research interests include design automation technologies for multiprocessor systems, embedded systems, and system-level design. Paulin has a BSc and an MSc in engineering physics and electrical engineering from Laval University, Quebec City, and a PhD in electronics engineering from Carleton University, Ottawa. He is a member of the IEEE.

Chuck Pilkington is a senior staff engineer in the System-on-Chip Platform Automation Group at STMicroelectronics. His research interests include hardware modeling and system software for high-performance parallel processing. Pilkington has a BSc in physics from the University of Waterloo, Ontario, and an MSc in electrical engineering from the University of Toronto.

Essaid Bensoudane is a research-and-development engineer in the System-on-Chip Platform Automation Group at STMicroelectronics. His research interests include system simulation and high-level system design and analysis. Bensoudane has a BSc from l'Institut Polytechnique de Grenoble, France, and an MSc in automation and systems engineering from l'École Polytechnique, Montreal.

■ Direct questions and comments about this article to Pierre G. Paulin, STMicroelectronics, Central R&D, SoC Platform Automation, 16 Fitzgerald Rd., Suite 100, Nepean, Ontario, K2H 8R6, Canada; pierre.paulin@st.com.