A Flexible Platform for Hardware-Aware Network Experiments and a Case Study on Wireless Network Coding

Xi Deng, Yuanyuan Yang and Sangjin Hong
Department of Electrical and Computer Engineering
New York State Center of Excellence in Wireless and Information Technology
Stony Brook University
Stony Brook, NY 11794, USA

ABSTRACT

In this paper, we present the design and implementation of a general, flexible hardware-aware network platform which takes hardware processing behavior into consideration to accurately evaluate network performance. The platform adopts a network-hardware co-simulation approach in which the NS-2 network simulator supervises the network-wide traffic flow and the SystemC hardware simulator simulates the underlying hardware processing in network nodes. In addition, as a case study, we implemented wireless all-to-all broadcasting with network coding on the platform. We analyze the hardware processing behavior during the algorithm execution and evaluate the overall performance of the algorithm. Our experimental results demonstrate that hardware processing has a significant impact on the algorithm performance and hence should be taken into consideration in the algorithm design. We expect that this hardware-aware platform will become a very useful tool for more accurate network simulations and optimal designs of processing-intensive applications.

Keywords: Network simulation, co-simulation, hardware behavior, hardware-aware, network coding, broadcasting, wireless network.

1. INTRODUCTION

As network capability is improved by new technologies in terms of bandwidth, latency and services, many emerging applications, such as video-on-demand services and voice-over-IP, are now running on networks. Such applications put tremendous demand on network resources and also pose challenges on hardware to process a large volume of traffic at a high speed. In the meanwhile, more and more sophisticated network algorithms, such as header compression, packet aggregation and encryption, are introduced into network protocols to improve network performance, which greatly increases the complexity of hardware processing as well. Although complex hardware processing may be necessary to realize certain network functions, it has a significant impact on network latency, bandwidth and power consumption. This effect may vary among different hardware and network configurations. In general, computational resource limited network devices, including many mobile devices, are more prone to be affected by such complex processing. As a result, hardware-aware algorithm designs and designated hardware components may be required for such devices to optimize the overall performance of the network system. However, traditional network performance evaluation tools usually do not possess hardware-aware capability and it is difficult to use such tools to identify hardware performance bottleneck. Thus, a platform with hardware awareness is needed to study the effect of hardware processing in networks.

Currently, most network algorithms are developed, validated and evaluated by either simulation or emulation. Simulators, generally focusing on network traffic and packet transmissions, provide limited modeling capability for hardware processing. In particular, processing functions are implemented at software level and the hardware behavior is usually simplified as a fixed delay. Such modeling is apparently inadequate and inaccurate when the processing is complex and unpredictable, leading to disparity in network performance evaluation. Alternatively, emulators perform real-time simulations by modeling the nodal processing and network transmissions using physical computers and network systems, thus they can obtain more accurate results on the behavior and performance of the simulated network. In traditional network emulators, the nodes in the simulated network are represented by actual computers on a one-to-one or one-to-many mapping. In the case of one-to-one mapping, where each network node occupies one computer or network device, the processing is performed in hardware. Although this is the most realistic network environment, the network size is constrained by the number of hardware systems, thus the emulation for very large networks is not feasible. In the case of one-to-many mapping, one or more network nodes are simulated as virtual nodes in one computer system, allowing certain scalability on the network size. However, since many virtual nodes share the same hardware utility, such as CPU or memory, hardware processing may not be realistically simulated. Besides, in both cases, the hardware systems used to represent network nodes have fixed hardware configuration and little hardware reconfigurable ability, which does not allow emulators to examine network performance under different hardware conditions. Moreover, some emulations may require certain expensive computational resources, which may not be always available or affordable.

From the above discussions, we can see that simulation provides inadequate supports on processing modeling, while emulation may be expensive and inflexible or not scalable. Thus, neither are suitable as a general platform to study the impact of hardware processing.

In this paper, we present a general, flexible network platform for hardware-aware network simulations. The platform seamlessly integrates a network simulator to simulate the network environment and a hardware simulator to simulate the nodal...
hardware operations above the physical layer. Such integration of two simulators with different simulation levels is generally called co-simulation. In our case, the network-hardware co-simulation provides a complete experimental platform where hardware processing can be easily modeled and examined in a specific network scenario with sufficient flexibility of reconfiguring both the network infrastructure and the underlying hardware architecture. We also present a case study on wireless all-to-all broadcasting with network coding on this platform. Our experimental results reveal that the processing time incurred by network coding has a significant impact on the overall performance, which increases the broadcasting time up to three times in the worst case. We also conduct performance tuning for the broadcast algorithm, which can only be performed in a hardware-aware environment. Due to its easy access to accurate, detailed information on hardware operations, which cannot be captured by the pure network simulation, and its flexibility of reconfiguring hardware components, which the network emulation lacks, we believe the platform will be a very useful tool for network algorithm/protocol designers and network related hardware designers.

The rest of the paper is organized as follows. Section II introduces the related work in network simulation and emulation. Section III gives the detailed description of the platform architecture and the implementation. Section IV describes a case study on the platform that evaluates the performance of wireless all-to-all broadcasting with network coding, and Section V presents the experimental results of the case study. Finally, Section VI concludes the paper.

II. PREVIOUS RELATED WORK

As discussed earlier, two commonly used approaches to evaluating network performance are simulation and emulation. One representative of network simulators is NS-2 [2], which is widely used by the networking research community due to its good modeling capability for a variety of network scenarios and protocols. However, network simulators can provide only very limited modeling capability for hardware processing. NS-2 provides a primitive way to model hardware processing as delays by its timer mechanism. More sophisticated hardware modeling is out of consideration and difficult to implement in NS-2. Another powerful simulator OPNET [1] employs the process editor to model hardware processing using a state-transition diagram approach. However, it cannot model detailed hardware processing either.

In addition, a simulator may be combined with some hardware facility to carry out network emulations for more realistic network modeling. Compared to pure network emulation, such combination aims to take advantage of resourceful network models in the simulator. For instance, NS-2 has been extended to have the emulation facility as described in [8]. The extension enables the simulator to interact with live network traffic, which may be generated by actual hardware. However, this approach requires the users to provide the desired real-time traffic or hardware, thus cannot serve as a general network emulator. In addition, the difficulty in synchronization with real-time traffic prohibits the entire emulation from being performed at high bandwidth.

On the other hand, network emulators employ actual computer systems and networks to provide a more realistic network environment. In the emulation, physical nodes represent the network nodes and a virtual network is created to regulate the network topology and protocols among all the nodes. Emulators for general networks, see, for example, [17], [13], [5], usually adopt real PCs as the physical nodes. Then hardware processing is actually executed rather than simulated, yielding more accurate results. However, the real execution implies that the processing is tightly associated with the hardware configuration of the physical nodes and may become inaccurate if the hardware configuration changes. There are also network emulators for a specific type of networks, such as Emulab [14], Motelab [18] and Emstar [12]. For example, the 802.11a/b/g testbed in Emulab scatters two different types of nodes with real wireless interfaces in an office building, allowing a number of configuration parameters. Compared to Emulab and other similar testbeds, which aim at constructing a real network environment, our proposed platform emphasizes on supporting general, flexible hardware modeling, which is usually neglected in other simulators and emulators.

Finally, a simple network-hardware co-simulation method was adopted in [11] for a specific simulation environment with networked embedded systems. Its main focus is to study the behavior of embedded systems when they are connected to a practical network environment. In such a system, there is a natural partition between the network modeling and the hardware modeling, which makes co-simulation much easier. However, it is difficult to use such an approach for general networks as there may not exist such a clear interface between the network and the hardware. In this paper, we will focus on general network-hardware co-simulation and treat the network traffic and the hardware nodes as an integrated system. In such a system, any network protocols and algorithms can be modeled in hardware so that network performance can be studied in a more realistic environment.

III. PLATFORM IMPLEMENTATION

In this section, we describe the implementation of our network platform. Different from network emulators, in which end-systems are virtual nodes emulated by real computers or specially designed network devices, this platform adopts a hardware simulator to simulate the hardware processing in the network nodes. Meanwhile, the platform utilizes a network simulator to create the desired network topology and manage the network traffic. The interface between the simulated hardware devices and the network is realized through the network-hardware co-simulation which provides the framework of the platform implementation.

In the network-hardware co-simulation, we adopt NS-2 simulator as the network simulator and SystemC hardware simulator as the hardware simulator. NS-2 is a popular software-level network simulator which is capable of creating various networks with flexible configurations; SystemC is a system-level hardware description language which describes the specific hardware without going into the details of the underlying hardware implementation. At the network level, NS-2 manages the global network scenario and provides interfaces to the hardware processing simulated by SystemC. NS-2 treats these interfaces as
special network protocols and thus obtains the hardware processing ability without significant changes on its framework. At the hardware level, SystemC models and simulates the specific hardware processing taking the network traffic its input. In particular, SystemC can either simulate a complete network device in hardware, or simulate some selected hardware components of a device to study the specific protocols or algorithms. This flexibility is very important, as in many situations, hardware modeling of a complete network device may be difficult and time-consuming, while the simulation of some key hardware components of the device is much easier and usually adequate in practice to study the impact of hardware processing.

Fig. 1 illustrates the overview of our platform. Similar to most network simulators, the platform requires a script provided by the user to specify the network environment. Such specification includes network topology, network traffic, protocols used by the user to specify the network environment. Such specifica-

Fig. 1. The overview of the platform

Time synchronization is a critical issue in the co-simulation. As both NS-2 and SystemC are event-driven simulators, each simulator is managed by the scheduler, which schedules and executes all the events in the order of time. In the co-simulation, the events must be scheduled and executed by a global scheduler with a global timer. However, it is difficult to implement a global scheduler because the schedulers of the two simulators have different specifications on the events. A feasible solution is to let two schedulers schedule their own events and in the meanwhile maintain the time synchronization between them.

Besides the events originally generated in the two simulators, the co-simulation incurs another type of new events, that is, the events of sending messages between the two simulators. For presentational convenience, we classify the events in the co-simulation into two categories: Type I events and Type II events. Type I events are the original events in the two simulators and Type II events are the events of sending messages between the two simulators. By definition, a Type II event is generated when a simulator decides to send a message to the other simulator, during the execution of a Type I event. Each event is associated with a time stamp indicating the exact time this event is executed, called the time of the event. Specifically, the time of a Type II event is assigned the current simulation time when this event is generated. The event with the earliest time in the co-simulation is called the executable event and there could be multiple executable events. When an executable event is executed, the simulation time advances to the time of this event. Notice that any Type II events are executable events because the time of Type II events is the current simulation time and must be the earliest. In each simulator, the scheduler maintains the list of events locally generated, among which the earliest Type I event is called the schedule event and its time is called the schedule time of the simulator.

At the beginning of the co-simulation, both simulators have only Type I events. After exchanging the schedule time between the two simulators, the simulator with earlier schedule time starts the simulation while the other simulator pauses and waits for the messages. Executable Type I events are then sequentially executed until there are no more such events, which means that the schedule time of this simulator surpasses either the schedule time of the other simulator or the time of Type II events. Then all the Type II events (if any) are executed and the messages are sent to the other simulator. Type I events are executed first because that each execution of Type II events leads to a message transmission between the two simulators and we can minimize the communication cost by putting all the Type II events together. After executing Type II events, the current simulator completes its execution cycle and pauses while the other simulator begins its execution cycle by receiving the messages. The execution cycles will be performed in turn among two simulators until no more events are left in either simulator. Then the co-simulation terminates. Table 1 gives the pseudo code of the event scheduling algorithm.

As shown in Table 1, the same scheduling algorithm is executed concurrently in both NS-2 and SystemC. In the algo-
TABLE 1  
SCHEDULING ALGORITHM IN BOTH SIMULATORS

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>( t_1 = \text{schedule_time(this)}; )</td>
</tr>
<tr>
<td>2.</td>
<td>( t_2 = \text{schedule_time(other \text{ simulator})}; )</td>
</tr>
<tr>
<td>3.</td>
<td>( \text{wait}(t_2); )</td>
</tr>
<tr>
<td>4.</td>
<td>while ((\text{exist_event})) {</td>
</tr>
<tr>
<td>5.</td>
<td>( \text{update}(t_1); )</td>
</tr>
<tr>
<td>6.</td>
<td>while ((t_1 &gt; t_2) \land (t_1 &gt; \text{time}(\text{Type II events}))) {</td>
</tr>
<tr>
<td>7.</td>
<td>( \text{send}(\text{Type II events}); )</td>
</tr>
<tr>
<td>8.</td>
<td>( \text{send}(t_1); )</td>
</tr>
<tr>
<td>9.</td>
<td>( \text{wait}(t_2); )</td>
</tr>
<tr>
<td>10.</td>
<td>( \text{get_executable_event}(e); )</td>
</tr>
<tr>
<td>11.</td>
<td>( \text{execute}(e); )</td>
</tr>
</tbody>
</table>

As mentioned earlier, Type II events send messages between the two simulators. How to efficiently communicate between the two simulators executed in two processes is another critical issue in the co-simulation.

Since NS-2 and SystemC are executed in two processes, their communication is accomplished through an interprocess queue. Fig. 3 illustrates the communication model between NS-2 and SystemC, where the entire procedure can be divided into three phases: message buffering, message transmission and message handling.

As mentioned in the last subsection, Type II events are generated during the execution of Type I events. The creation of Type II events implies a message transmission between the two simulators. As shown in Table 1, instead of starting the message transmission immediately, the scheduler will wait until all the executable Type I events are executed. Therefore, a message will be temporarily stored in the buffer whenever a Type II event is generated during the execution.

Message transmission occurs when there are no more executable Type I events in the scheduler. Since the two simulators reside in different processes, messages cannot be sent directly. We use the message queue as the interprocess communication mechanism so that messages can be conveniently and efficiently sent and managed. The message queue is responsible for receiving messages from one simulator and sending messages to the other simulator.

To complete the message transmission, some additional information should be attached to the actual message. The message format used in the co-simulation includes four fields: \( mid, length, time \) and data. \( mid \) specifies the destination module this message should be delivered to, where a module corresponds to a functional unit in the network node. To correctly receive the messages, every module in SystemC must have a corresponding module in NS-2 and the two modules are assigned the same \( mid \) before the simulation. \( length \) indicates the length of field data, facilitating the receiving of messages. \( time \) records the time when the message is created. When the simulator on the receiving side receives this message, the simulation time will be advanced according to \( time \). data contains the actual message stored as a string. The receiver is responsible for converting the string to the proper data format.

The receiving procedure does not require any buffer. The messages received will be immediately delivered to the corresponding module indicated in \( mid \). Since both simulators are event-driven, message delivery is considered as the execution of an event.

C. Module-Scheduler Interface Design

In both NS-2 and SystemC, the scheduler schedules the events, while other functional modules execute the events. The synchronization and communication are mainly accomplished by the scheduler, and message buffering and message handling require the cooperation between the scheduler and functional modules. The interface between the scheduler and functional modules must be extended to provide extra functions for the co-simulation. In particular, the scheduler should provide the store function for functional modules to temporarily store the messages in the local buffer while functional modules are required to provide corresponding handlers to handle the receiving of messages. Due to the differences in the frameworks of two simulators, we describe the interfaces in NS-2 and SystemC separately next.

NS-2 Interface

In NS-2, the communication among different modules is realized by calling functions send/recv which send/receive data packets between the modules. Two similar functions sendmsg/recvmsg are provided to process the co-simulation messages. Function sendmsg is provided and implemented by the scheduler. Once a module has a Type II event, it first creates a formatted message, then calls function sendmsg and pauses.
the execution until the scheduler calls function recvmsg, which is provided by the module itself. Function recvmsg is not added directly to the functional modules. Instead, it is added as a virtual function in the base class, from which all the functional modules in NS-2 inherit, as the scheduler needs to call this function in different modules. The implementation of recvmsg is module dependent, thus, each co-simulation module needs to define its own implementation of recvmsg.

SystemC Interface

In SystemC, a special module ns_module is created to communicate between the scheduler and other modules. In the simulation, ns_module can be considered as a regular module which runs NS-2 simulator, thus the communication between the two simulators becomes the communication between ns_module and other modules. In SystemC, channels are the communication media and modules use ports to access the channels. In the implementation, two special ports send/recv are created as the interface between ns_module and other modules.

ns_module communicates with NS-2 through functions sendmsg/recvmsg. Different from NS-2, SystemC does not allow the immediate processing of the received message. Therefore, when function recvmsg is called by the scheduler, ns_module will store the message in its buffer and send a processing request to the scheduler. When the request is granted, ns_module will send the message to other modules through port recv. When other modules send a message to ns_module through port send, ns_module will call function sendmsg to send the message to the buffer. The implementation details of these interface functions are SystemC-dependent and are omitted in this paper due to limited space.

D. Summary

In the above, we have described the implementation outline of our platform. The implementations of the above three major issues compose the framework of the co-simulation. To run a network-hardware co-simulation on the platform, users simply provide the desired network and hardware models by a script. The primary features of the platform are summarized as follows.

1. The platform provides powerful modeling capability for both network and hardware through the co-simulation of NS-2 and SystemC.
2. The platform adopts NS-2 user interface, facilitating the easy use by a large number of users familiar with NS-2.
3. The platform provides great flexibility in hardware modeling such that the users can flexibly partition simulated functions between hardware and software.
4. The platform is executed purely in software, without any requirement on specific hardware support.

IV. CASE STUDY OF WIRELESS ALL-TO-ALL BROADCASTING USING NETWORK CODING

The co-simulation of NS-2 simulator and SystemC simulator provides a configurable environment for studying network behaviors with hardware awareness. In this section, we present a case study on the platform to demonstrate how it can be used and what kind of information can be obtained related to hardware processing.

We consider all-to-all broadcasting in a wireless network that uses network coding in packet transmissions. Network coding is a promising generalization of routing that allows a network node to generate output packets by encoding its received packets to reduce bandwidth consumption in the network and improve network throughput. While network coding leads to improved throughput, due to the extra work in coding/decoding packets, it inevitably increases the processing complexity of network nodes as well. However, this type of processing overhead is largely ignored in conventional network simulators, often leading to inaccurate performance results.

As a case study, in this section we examine a network algorithm that implements wireless all-to-all broadcasting with network coding on our platform by taking hardware processing into consideration. As will be seen from our experimental results, hardware processing has a significant impact on the broadcasting performance, doubling the broadcasting time in the worst case. We also show that with the network and hardware traces generated by the platform, performance tuning can be performed to effectively alleviate such performance degradation.

Next we first introduce the all-to-all broadcasting scheme based on network coding and analyze its hardware processing by decomposing it into different functional units. Then we present the hardware modeling of intra-node processing on the platform. We will give the experimental results in Section V.

A. Wireless All-to-All Broadcasting with Network Coding

Network coding was first introduced by Ahlswede et al. in [4], where independent information flows are combined to better utilize network bandwidth and achieve higher throughput. Later, linear coding [6] and random coding [7] were introduced to provide near-optimal performance and make network coding more practical. Recently, this promising technique has been extended to wireless networks [9], [10], [15], [16], as the broadcast nature and limited bandwidth of wireless channels make them a perfect place to apply network coding. In this case study, we consider wireless all-to-all broadcasting with network coding. All-to-all broadcasting is the most bandwidth consuming communication operation that can reveal network performance in the most stressed scenario. Since the main purpose of this case study is to examine the processing impact on the network algorithm, not to design a new network coding algorithm, we will adopt the algorithm introduced in [10], where random network coding is used to achieve high efficiency on all-to-all broadcasting. This algorithm is the first distributed algorithm which can be easily applied to a wireless network environment. We consider a basic scenario, where every node in the wireless network sends a packet to all other nodes. As will be seen in Section V, this scenario is sufficient to illustrate the benefit of network coding and its hardware processing impact.

Consider a wireless ad hoc network with n nodes. Let \( x_i \) denote the packet that node \( i \) needs to send. With network coding, the packet sent or received by a node is a linear combination of the packets such that \( y = \sum_{i=1}^{n} g_i x_i \), where \( g_i \) is the coding coefficient on \( x_i \) and vector \( g = \{g_1, g_2, \ldots, g_n\} \) is the coding vector of packet \( y \). In each transmission, the coding vector is randomly assigned over a finite field and sent along with packet \( y \). If the coding vector associated with the incoming packet is linearly independent of all the coding vectors received earlier,
the packet is called innovative. The packets can be decoded when a node receives \( n \) innovative packets.

In our experiment, we use a 2D grid network topology for convenience. The transmission range is defined such that each node can only send packets to its neighbors in the grid. The transmission mode is half-duplex and a node sends only one packet in each transmission. Each node maintains a packet buffer of size \( n \) to receive the innovative packets and stores its own packet in the packet buffer initially. The transmissions are performed in a time-slotted manner, starting at the beginning of each time slot. To avoid transmission collision, nodes are scheduled to transmit in collision-free time slots. When a node is scheduled to transmit, it sends a random linear combination of all the packets in the packet buffer along with the coding vector. The transmission continues until all nodes decode the \( n \) original packets. Table 2 gives the pseudo code of the intra-node procedure for both transmitting and receiving operations.

**TABLE 2**

**INTRA-NODE PROCEDURE IN ALL-TO-ALL BROADCASTING**

<table>
<thead>
<tr>
<th>Transmitting procedure:</th>
<th>Receiving procedure:</th>
</tr>
</thead>
<tbody>
<tr>
<td>while TRUE</td>
<td>while packets are not decoded</td>
</tr>
<tr>
<td>if transmission scheduled in this time slot</td>
<td></td>
</tr>
<tr>
<td>Generate a coding vector randomly</td>
<td></td>
</tr>
<tr>
<td>Encode the packet</td>
<td></td>
</tr>
<tr>
<td>Send the packet</td>
<td></td>
</tr>
<tr>
<td>end if</td>
<td>if the packet is innovative</td>
</tr>
<tr>
<td>Wait until next time slot</td>
<td>Store the packet in the buffer</td>
</tr>
<tr>
<td>end while</td>
<td>end if</td>
</tr>
<tr>
<td></td>
<td>if the buffer has ( n ) packets</td>
</tr>
<tr>
<td></td>
<td>Decode the packets</td>
</tr>
<tr>
<td></td>
<td>end if</td>
</tr>
<tr>
<td></td>
<td>end while</td>
</tr>
</tbody>
</table>

There are two issues that need to be considered here. One is how to obtain a collision-free transmission schedule, and another is how long a time slot should be.

We first consider the transmission schedule. Besides collision-free transmissions, the schedule should also possess fairness and efficiency. In other words, every node in the network should have an approximately equal opportunity to transmit and the transmissions in a time slot should saturate the transmission channels without collisions. Next we give a transmission schedule which allows every node in the grid to transmit once in every five consecutive time slots.

We first consider an infinite 2D grid network. Denote the nodes by two coordinates \((x, y)\). Consider transmission assignment I in Fig. 4(a) where nodes \((2 i + j, i - 2 j)\) are transmitters and \(i\) and \(j\) are integers. It can be proved that in such an assignment, there will be no transmission collisions and every node is either a transmitter or a receiver. We call such an assignment perfect assignment. Similarly, we can construct other four perfect assignments where nodes \((2 i + j, i - 2 j + 1)\), \((2 i + j + 1, i - 2 j)\), \((2 i + j, i - 2 j - 1)\) and \((2 i + j - 1, i - 2 j)\) are transmitters, respectively. It is easy to see that each transmitter in these four perfect assignments corresponds to a receiver in assignment I. Therefore, each node must be a transmitter in exactly one of the five assignments. The transmission schedule that repeats these five transmission assignments can guarantee every node in the grid to transmit once in every five consecutive time slots in a collision-free manner. Thus, fairness and efficiency are achieved. In our experiments, we apply the schedule in a bounded grid network, where transmissions on the border are trimmed off. Fig. 4(f) shows such an assignment in a \( 5 \times 5 \) network.

Fig. 4. Transmission assignments in a 2D grid network, where (a)-(e) show the five assignments for an infinite grid network, and (f) shows an assignment for a bounded \( 5 \times 5 \) grid network. Transmitters are labeled with their coordinates and the arrows represent the transmission directions from transmitters to receivers.

We now consider the length of a time slot. The time slot should be sufficiently long so that a packet can be successfully received within a time slot. Otherwise, the packets transmitted in two consecutive time slots may collide. Note that network coding will incur some extra time in each time slot. When the hardware processing time for network coding is ignored, the minimum length of a time slot can be the packet transmission time which is calculated by the length of the packet divided by the network bandwidth. When the hardware processing is considered, the time slot length should be at least the sum of the encoding time and the transmission time. The encoding time during the broadcasting varies and depends on the number of packets stored in the buffer at the time of encoding. Therefore, to completely avoid collisions, the time slot length should be at least equal to the sum of the maximum encoding time and the transmission time. However, as will be seen in Section V, a few collisions may not affect the completeness of the broadcasting and it is unnecessary to set the time slot length to this maximum value. Instead, a suitable time slot length may be determined by examining the broadcasting performance under different time slot lengths and picking the one that achieves the best performance. We call this process performance tuning.

To obtain more realistic performance of the broadcasting algorithm, we need to consider the intra-node hardware behavior. In the next subsection, we decompose the intra-node processing into separate functions and describe the main operations in each function.

**B. Function Decomposition of Intra-node Processing**

In the all-to-all broadcasting algorithm, the intra-node processing can be divided into the following five phases based on
their functions:
1. Receiving phase. The incoming packets are stored in a temporary buffer.
2. Checking phase. This phase checks whether the packets in the buffer are innovative. If a packet is innovative, it will be stored in the receiving buffer; otherwise, the packet will be disposed immediately. (Recall that an innovative packet has a coding vector independent of the coding vectors already received. The dependency check is accomplished by checking whether the rank of the matrix composed of the coding vectors increases when the incoming coding vector is added to the matrix.)
3. Encoding phase. When a node is scheduled to transmit, the encoding routine will create a new encoded packet by performing a random linear combination among all the packets in the receiving buffer of the node.
4. Transmitting phase. The prepared packet will be broadcast by the node.
5. Decoding phase. This phase starts when \( n \) innovative packets are received, where \( n \) is the network size. The decoding routine is similar to the dependence checking routine except that Gaussian Elimination is performed on both coding vectors and the packet data.

Among these five phases, it is easy to see that most of the processing time is spent on the checking phase, encoding phase and decoding phase. In the decoding phase, the decoding routine is executed only once when all the \( n \) packets can be decoded and thus has little impact on the broadcast time. Therefore, we will mainly consider the checking and encoding routines. To quantify the processing time of these two routines, we assume that the routines are executed on a network processor without any specific hardware support. Then the processing time depends on two parameters, the CPU frequency \( f_{cpu} \) and the network bandwidth \( B_m \) in the network processor. Also, we define \( P \) to be the packet length, using Byte(B) as the unit. The length of the coding vector then is \( nB \).

In the checking routine, the coding vectors in both the incoming packet and the receiving buffer should be loaded into the network processor, which requires \( c \cdot n \) time where \( c \in \{1, 2, \cdots, n\} \) represents the number of coding vectors. The processor then computes the matrix rank by Gaussian Elimination. Assuming that each operation takes one cycle, the entire computation takes at most \( \frac{c \cdot n^2}{B_m} \) time. After the matrix computation is done, the innovative packet requires additional \( \frac{n^3 + nP}{B_m} \) time to be written into memory. Thus, the maximum execution time of a checking routine is

\[
T_{\text{check}} = n^2 + \frac{n^3 + nP}{B_m} + \frac{n^3}{f_{cpu}}.
\]

In the encoding routine, all packets in the receiving buffer should be loaded into the network processor with at most \( \frac{n(nP)}{B_m} \) time. The computation of encoding a packet takes \( \frac{n^2 + nP}{B_m} \) time. Thus, the total time of an encoding routine is at most

\[
T_{\text{encode}} = n(n + P)(\frac{1}{B_m} + \frac{2}{f_{cpu}}).
\]

Take a commonly used network processor, RouterBOARD [3] with \( f_{cpu} = 266 MHz \) and \( B_m = 800 Mb/s \), as an example. Assume \( n = 25 \), \( P = 100 \) and network bandwidth is \( 54 Mb/s \) as adopted in 802.11g. Then the processing time of the checking routine is approximately \( 20 \mu s \), and the time for encoding is approximately \( 28 \mu s \). On the other hand, when hardware processing is not considered, the time slot length can be as short as \( 18 \mu s \), which is the packet transmission time. We can see that the processing time is far more than negligible when the nodes adopt this network processor.

C. Hardware Modeling for Intra-node Processing

The intra-node processing is implemented in a processing agent written in SystemC on our platform. Each network node in the broadcasting algorithm has an instance of such an agent, which serves as a hardware network processor. Although accurate simulation of the complete algorithm execution is feasible in SystemC [19], for the purpose of revealing the effect of hardware processing, it is sufficient to consider only the major functions in the algorithm. As discussed earlier, checking and encoding are the two major functions. Therefore, we set up a simplified network processor model to simulate the execution of checking and encoding routines.

As shown in Fig. 5, the simplified network processor consists of five units. Instead of simulating a CPU that executes all the routines, we employ two functional units, the checking unit and the encoding unit to implement the checking and encoding routines respectively. Thus the CPU unit is simplified to only simulate the computational delay of the two routines. During the execution, the checking and encoding units record the operation cycles of computation and send a request to the CPU unit for the delay calculation. The CPU unit then calculates and simulates the computational delay based on CPU frequency and the number of cycles. If the encoding and decoding routines send the request simultaneously, which causes a contention, the CPU unit will process the requests sequentially on a first-come-first-serve basis. Besides these units, the memory unit performs load/store operations and simulates memory latency whenever memory is accessed. The timer unit looks up the scheduling table at the beginning of each time slot and notifies the encoding unit to start the encoding routine when the node is scheduled to transmit.

V. EXPERIMENTAL EVALUATION

In this section, we present the experimental results for the wireless all-to-all broadcasting algorithm with network coding and analyze the effect of hardware processing on its overall performance.

A. Experiment Setup

The all-to-all broadcasting algorithm is implemented in a 2D grid wireless ad hoc network. We examine two network scenarios: a \( 5 \times 5 \) grid and a \( 6 \times 6 \) grid. The network environment is simulated in NS-2. The distance between neighboring nodes is \( 100m \) and each node can broadcast only to its neighbors in the grid. The bandwidth is set to \( 54 Mb/s \) while the MAC
layer is simplified by assuming sending packets without collision avoidance (recall that the collisions are eliminated by the collision-free scheduling algorithm). A connecting agent above the MAC layer is created to represent a network node in NS-2. This agent is only responsible for receiving and transmitting packets from/to its MAC layer while the actual processing of the packet is simulated in the processing agent written in SystemC. The performance metric is the finish time of the all-to-all broadcasting when every node in the network decodes all the packets.

B. Simulation Results of Hardware Processing Impact

To illustrate the impact of hardware processing, we compare the results to the case when hardware processing is not considered. In this case, to minimize the finish time, the transmissions are scheduled immediately after the last round of transmissions are completed, that is, the time slot length is set to the transmission time of a packet. Table 3 lists the finish time with different packet lengths when hardware processing time is set to zero. In the following experiments, the finish time is the ratio of the actual time to the corresponding time in Table 3 for the same packet length.

We now examine the performance when considering hardware processing time. The experiment is conducted under four different hardware configurations which are the combinations of two parameters: CPU frequency $F$ and memory bandwidth $B$, with $F = 500 MHz$, $1 GHz$, and $B = 800 Mb/s$, $1.6 Gb/s$. These values are comparable to the typical values of current generation network processors, such as RouterBOARD 433 series [3].

B.1 Impact of Time Slot Length

As discussed in Section IV-A, a suitable time slot length cannot be predetermined when hardware processing is considered. Hence, we performed the simulation using a gradually increased time slot length. Fig. 6 shows the broadcasting finish time with different time slot lengths when packet length $P$ is $100B$ and $1400B$, respectively. For each line in the figure, the shortest time slot length is the minimum length with which the broadcasting can succeed, while the longest time slot length is the threshold from which the finish time increases monotonically as the time slot length increases.

We can see that hardware processing time leads to a substantial degradation of the broadcasting performance: the finish time is increased by about 15% to 150% in the experiments. Besides, different time slot lengths cause different degree of degradation: as the time slot length increases, the finish time generally drops at first and then rises. This pattern indicates that the performance is more severely degraded when the time slot length is too close to the minimum or too long. On one hand, shorter time slots lead to more collisions and tend to increase the total number of time slots used to complete the broadcasting; on the other hand, longer time slots tend to reduce the collisions but increase the total time for a fixed number of time slots. Therefore, a suitable time slot length should be a trade-off between these two aspects.

In addition, we can observe that a small difference, say, 5%, in the time slot length may cause a dramatic change in the overall performance. For example, when $P = 1400B$, $F = 500 MHz$, $B = 800 Mb/s$ in the $5 \times 5$ grid, the change in the time slot length from 240 to 250 reduces the finish time by more than 50%. Such sensitivity is difficult to capture on a platform where hardware processing is coarsely modeled.

![Fig. 6. Broadcasting finish time with different time slot lengths considering hardware processing. The metric used is the finish time normalized to the corresponding finish time without considering hardware processing.](image)

B.2 Impact of Hardware Configuration

As can be observed in Fig. 6, there is a tendency that the hardware configuration with higher CPU frequency and higher memory bandwidth yields shorter finish time. In particular, the performance is more sensitive to the CPU frequency. Increasing CPU frequency causes a much sharper decrease of the finish time than increasing memory bandwidth. In some extreme cases, the finish time when $B = 1.6 Gb/s$ is even longer than the time when $B = 800 Mb/s$. This indicates that the computation is the major cause of the performance degradation. Also, we can see that when $P = 1400B$, the finish time largely follows the trend of dropping and then rising. However, when $P = 100B$, the finish time is much more disordered. This phenomenon is mainly due to the overhead from the coding vector, which is relatively high when $P = 100B$.

B.3 Shortest Finish Time by Performance Tuning

Finally, we perform performance tuning on the broadcasting algorithm by adjusting the time slot length to achieve the shortest finish time. For example, in Fig. 6, when $P = 100B$, $F = 1 GHz$ and $B = 1.6 Gb/s$ in the $5 \times 5$ grid, the shortest finish time is found when the time slot length is $22 \mu s$. Fig. 7 shows the shortest finish time after the performance tuning.

We can see that in the $5 \times 5$ grid, when the packet length is longer than $200B$, the finish time drops dramatically and remains stable afterwards, which means that the effect of the coding vector is largely eliminated; on the contrary, the finish time in the $6 \times 6$ grid drops more slowly, revealing a more severe impact of processing time. Besides, as mentioned earlier, the finish time is much more affected by the CPU frequency, leading to a 15% gap between $F = 500 MHz$ and $F = 1 GHz$. For the configuration with the highest CPU frequency and memory bandwidth among the four configurations, the effect caused by
the processing time is about 15%, which apparently cannot be ignored. Finally, it should be mentioned that since an actual network processor is much more complex than our simplified model, the real impact of hardware processing could be even greater.

<table>
<thead>
<tr>
<th>Packet length (Byte)</th>
<th>100</th>
<th>200</th>
<th>300</th>
<th>400</th>
<th>500</th>
<th>600</th>
<th>700</th>
<th>800</th>
<th>900</th>
<th>1000</th>
<th>1100</th>
<th>1200</th>
<th>1300</th>
<th>1400</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 × 5 Grid (ms)</td>
<td>1.14</td>
<td>2.04</td>
<td>2.94</td>
<td>3.84</td>
<td>4.74</td>
<td>5.64</td>
<td>6.54</td>
<td>7.44</td>
<td>8.28</td>
<td>9.18</td>
<td>10.08</td>
<td>10.98</td>
<td>11.88</td>
<td>12.72</td>
</tr>
<tr>
<td>6 × 6 Grid (ms)</td>
<td>1.83</td>
<td>3.17</td>
<td>4.49</td>
<td>5.81</td>
<td>7.13</td>
<td>8.36</td>
<td>9.68</td>
<td>11.00</td>
<td>12.32</td>
<td>13.64</td>
<td>14.96</td>
<td>16.19</td>
<td>17.51</td>
<td>18.83</td>
</tr>
</tbody>
</table>

Fig. 7. The shortest finish time obtained under different packet lengths, normalized to the finish time in Table 3.

From our case study experiment, we can draw the following conclusions:

- Hardware processing time has a great impact on the overall performance of the algorithm.
- Computation is a major part of the processing time in typical network processors.
- The time slot length used in the algorithm can be adjusted to reduce the hardware processing effect. The suitable time slot length depends on the network processor and network bandwidth.

VI. CONCLUSIONS

In this paper, we have presented an efficient, flexible network platform to support the hardware-aware performance study of network algorithms/protocols. Based on the co-simulation of NS-2 and SystemC, the platform provides a network environment where both network traffic and intra-node processing can be well modeled and configured. By running a case study of wireless all-to-all broadcasting with network coding on the platform, we have demonstrated that hardware processing has a great impact on the performance evaluation of network algorithms, which may severely affect the actual performance obtained. We have also shown that the impact from hardware processing can be effectively examined and reduced through performance tuning with the support of the platform. Therefore, we believe such a platform is a very useful tool for studying and developing network algorithms and protocols.

REFERENCES