A hardware-accelerated implementation of a signaling protocol

Haobo Wang, Malathi Veeraraghavan and Ramesh Karri
Polytechnic University, New York

haobo_w@photon.poly.edu, mv@poly.edu, ramesh@india.poly.edu

Abstract: Signaling protocols are primarily implemented in software for two reasons: protocol complexity and the requirement for flexibility. However, even with state-of-the-art processors, software implementations of signaling protocol are rarely capable of handling over 1000 calls/sec. Correspondingly, call setup delays per switch are in the order of milliseconds. Towards improving performance for high-speed networks, we implemented a signaling protocol in reconfigurable FPGA hardware. Our implementation demonstrates the feasibility of 100X and potentially 1000X speedup vis-à-vis its software counterpart. The impact of this work can be quite far-reaching by allowing connection-oriented networks to support a variety of new applications, even those with short call holding time.

I. INTRODUCTION

Signaling protocols are used in connection-oriented networks primarily to set up and release connections. Examples of signaling protocols include Signaling System 7 (SS7) in telephony networks [1], the Private Network Node Interface (PNNI) signaling protocol in Asynchronous Transfer Mode (ATM) networks [2], Label Distribution Protocol (LDP) [3], Constraint-based Routing LDP (CR-LDP) [4] and Resource reServation Protocol (RSVP) [5] in Multi-Protocol Label Switched (MPLS) networks, and the extension of these protocols for Generalized MPLS (GMPLS) [6]-[8], which supports Synchronous Optical Network (SONET), Synchronous Digital Hierarchy (SDH) and Dense Wavelength Division Multiplexed (DWDM) networks.

Signaling protocols are primarily implemented in software. There are two important reasons for this choice. First, signaling protocols are quite complex with many messages, parameters and procedures. Second, signaling protocols are updated often requiring a certain amount of flexibility for upgrading field implementations. While these are two good reasons for implementing signaling protocols in software, the price paid is performance. Even with the latest processors, software signaling protocol implementations are rarely capable of handling over 1000 calls/sec. Correspondingly, call setup delays per switch are in the order of milliseconds.

Towards improving performance, we undertook a hardware implementation of a signaling protocol. We used reconfigurable hardware, i.e., Field Programmable Gate Arrays (FPGAs) [9]-[13] to solve the inflexibility problem. These devices are a compromise between general-purpose processors used in software implementations at one end of the flexibility-performance spectrum, and Application Specific Integrated Circuits (ASICs) at the opposite end of this spectrum. FPGAs can be reprogrammed with updated versions as signaling protocols evolve while significantly improving the call handling capacities relative to software implementation. As for the challenge posed by the complexity of signaling protocols, our approach is to only implement the basic and frequently used operations of the signaling protocol in hardware, and relegate the complex and infrequently used operations to software.

Our VHDL implementation has been mapped into two FPGAs on the WILDFORCE reconfigurable computing board – a Xilinx XC4036 FPGA with 62% resource utilization and a XC4013 with 8% resource utilization. From the timing simulations, we determined that a call can be processed in 6.6\(\mu\)s assuming a 25MHz clock yielding a call handling capacity of 150,000 calls/sec. Optimizing this implementation (for example, using pipelining) will increase the call handling capacities even further.

The impact of this work can be quite far-reaching. It is especially important in high-speed networks because call setup delay overhead becomes a larger percentage of the total delay. By decreasing call setup delays, we can improve utilization in high-speed connection-oriented networks.

Section II presents background material on connection setup and teardown procedures and surveys prior work on this topic. Section III describes the signaling protocol we implemented in hardware. Section IV describes our FPGA implementation while Section V summarizes our conclusions.

II. BACKGROUND AND PRIOR WORK

In this section, as background material, we provide a brief review of connection setup and release. We also describe prior work on this topic.

A. Background

This work is sponsored by a NSF grant, 0087487, and by NYSTAR (The New York Agency of Science, Technology and Academic Research) through the Center for Advanced Technology in Telecommunications (CATT) at Polytechnic University.

1 This work is sponsored by a NSF grant, 0087487, and by NYSTAR (The New York Agency of Science, Technology and Academic Research) through the Center for Advanced Technology in Telecommunications (CATT) at Polytechnic University.

2VHDL stands for VHSIC Hardware Description Language, where VHSIC stands for Very High Speed Integrated Circuits.
Connection setup and release procedures, along with the associated message exchanges, constitute the signaling protocol. Setting up a connection at a switch consists of six steps:

1. Parsing received signaling messages.
2. Determining the next-hop switch toward which the connection should be routed.
3. Checking for the availability of and reserving required resources.
4. Assigning “labels” for the connection. The exact form of the “label” depends on the type of connection-oriented network. For example, in SONET/SDH networks, the label identifies a timeslot.
5. Programming the switch fabric to map incoming labels to outgoing labels.
6. Updating the connection state information, constructing an outgoing signaling message and transmitting it.

In a classical connection setup procedure as illustrated in Fig. 1, the setup progresses from the calling end device toward the called end device, and the success indication messages travel in the reverse direction. In this scenario, the first three steps should be performed in the forward direction so that resources are reserved as the setup proceeds, while the remaining steps can be performed as signaling proceeds in the forward direction or in the reverse direction. Other variants of this procedure are possible such as reverse direction resource reservation [5]. Upon completion of data exchange, the connection is released with a similar end-to-end release procedure.

We illustrate a typical architecture for switches in a connection-oriented network in Fig. 2. The user-plane hardware consists of a switch fabric and line cards that terminate incoming/outgoing lines carrying user traffic. The control-plane unit consists of a signaling protocol engine, which could have a hardware accelerator as we are proposing, or be completely implemented in the software residing on the microprocessor.

There are many signaling protocols as listed in Section I. In addition, many other signaling protocols have also been proposed in the literature [14]-[18]. Some of these protocols such as Fast Reservation Protocol (FRP) [18], fast reservation schemes [15][16], YESSIR [14], and PCC [17] have been designed to achieve low call setup delay by improving the signaling protocols themselves. FRP is the only signaling protocol that has been implemented in ASIC hardware. Such an ASIC implementation is inflexible. More recently, Molinero-Fernandez and Mckeown [19] are implementing a technique called TCP Switching in which the TCP SYNchonize segment is used to trigger connection setup and TCP FINish segment is used to trigger release. By processing these segments inside switches, the TCP SYN/FIN procedures become comparable to a signaling protocol for connection setup/release. They are implementing this technique in FPGAs.

### III. Signaling Protocol

In this section, we describe the signaling protocol we implemented in hardware. It is not a complete signaling protocol specification. Our approach is to define a subset large enough that a significant percentage of user requirements can be handled with this subset. Infrequent operations are delegated to the software signaling process, as shown in Fig. 2. Therefore, often in this description, we will leave out details that are handled by the software.

#### A. Signaling messages

![Fig. 3. Signaling messages](image)

We defined four signaling messages, Setup, Setup-Success, Release, and Release-Confirm. Fig. 3 illustrates the detailed fields of these messages.
The Message Length field specifies the length of the message. The Time-to-Live (TTL) field has the same function as in IP networks. The Message Type field distinguishes four different messages. The Connection Reference identifies a connection locally. The Source IP Address, Destination IP Address, and Previous Node’s IP Address specify the end hosts and the previous node respectively. The Bandwidth field specifies the bandwidth requirement of the connection. The Interface/timeslot pairs are used to program the switch fabric. If there are an odd number of interface/timeslot pairs, a 16-bit pad is needed because the message is 32-bit aligned. The Checksum field covers the whole message.

In Setup-Success message, Bandwidth field records the allocated bandwidth. In Release and Release-Confirm messages, Cause field explains the reason of release. The Message Length, Message Type and Connection Reference fields are common to all messages and allocated in the same relative position. Such an arrangement simplifies hardware design.

B. State transition diagram of a connection

![State transition diagram of a connection](image)

Each connection passes through a certain sequence of states at each switch. In our protocol, we define four states, Setup-Sent, Established, Release-Sent and Closed. Fig. 4 shows the state transition diagram. Initially, the connection is in the Closed state. When a switch receives a Setup message, if resources are available, the switch accepts the request, sends the message to the next switch on the path, and marks the state of the connection as Setup-Sent. When a switch receives a Setup-Success message, which means all switches along the path have successfully established the connection, the state of the connection changes to Established. Release-Sent means the switch has received the Release message, freed the allocated resources, and sent the message to the previous node. When the switch receives the Release-Confirm message, the connection is successfully terminated, the state of the connection returns to Closed.

C. Data tables

There are five tables associated with the signaling protocol, namely, Routing table, Connection Admission Control (CAC) table, Connectivity table, State table, and Switch-mapping table, shown in Fig. 5. The Routing table is used to determine the next-hop switch. The index is the destination address; the fields include the address of the next switch and the corresponding output interface. The CAC table maintains the available bandwidth on the interfaces leading to neighboring switches. The Connectivity table is used to map the interface numbers used at neighboring switches to local interface numbers. This information will be used to program the switch fabric.

The State table maintains the state information associated with each connection. The connection reference is the index into the table. The fields include the connection references and addresses of the previous and next switches, the bandwidth allocated for the connection, and most importantly, the state information as defined in Fig. 4.

Switch fabrics, such as PMC-Sierra’s PM5372, Agere’s TDCS6440G and Vitesse’s VSC9182, have similar programming interfaces. For example, VSC9182 has an 11-bit address bus A[10:0] and a 10-bit data bus D[9:0]. The switch is programmed by presenting the output interface/timeslot number on A[10:0] and the input interface/timeslot number on D[9:0]. We define a generic Switch-mapping table to emulate this programming interface, with the connection reference as the index and the incoming interface/timeslot pair and the outgoing interface/timeslot pair as the fields.

IV. FPGA-BASED IMPLEMENTATION

To demonstrate the feasibility and advantage of hardware signaling, we implemented a signaling hardware accelerator in FPGA. We used the WILDFORCE™ reconfigurable computing board shown in Fig. 6. It consists of five XC4000 series Xilinx® FPGAs, one XC4036 (CPE0) and four XC4013 (PEn). These five FPGAs can be used for user logic while the crossbar provides programmable interconnections.
between the FPGAs. In addition, there are three FIFOs on the board, and one Dual Port RAM (DPRAM) attached to CPE0. The board is hooked to the host system through the PCI bus. The board supports a C language based API through which the host system can dynamically configure the FPGAs and access the on-board FIFOs and RAMs.

For our prototype implementation, we used CPE0, PE1, FIFO0, FIFO1 and DPRAM. The CPE0 implements the signaling hardware accelerator state machine, State and Switch-mapping tables, FIFO0 controller, and DPRAM controller. The DPRAM implements the Routing, CAC and Connectivity tables. FIFO0 and FIFO1 work as receive and transmit buffers for signaling messages. PE1 implements the FIFO1 controller and provides the data path between CPE0 and FIFO1.

In the following subsections, we describe the sizes of data tables selected for our hardware implementation, and the state transition diagram of the hardware accelerator. We also present two novel approaches for managing resources such as timeslots and connection references.

A. Parameters

Our implementation supports 5-bit addresses for all nodes, 16 interfaces per switch, a maximum bandwidth of 16 OC1s per interface, and a maximum of 32 simultaneous connections at each switch. We were primarily limited by the size of the on-board DPRAM on our off-the-shelf prototype board, and hence were forced to use these small values for address sizes, etc. The FPGA implementation itself did not limit these parameters. Hence, we can easily increase the values of these parameters when designing a customized board.

In recent years, there has been significant progress in fast table lookup in both research literature and commercial products [20][21]. Lookup/classification co-processors are widely available, such as Silicon Access Networks’ IAP, PMC-Sierra’s ClassiPI, Solidum’s PAX1200, etc. These chips can easily process 65M lookups/sec [22]. In our prototype implementation, we assumed that routing table lookups can be offloaded to an external co-processor, and used an equivalent three-memory access duration to simulate a routing table lookup.

B. State transition diagram of the signaling hardware accelerator

Fig. 7 shows the detailed state transition diagram of the signaling hardware accelerator. When a signaling message arrives, it is temporarily buffered in FIFO0. The hardware accelerator then reads the messages from FIFO0 and delimits the messages according to the Message Length field. The Checksum field is verified. The State table is consulted to check the current state of the connection. Based on the Message Type field, the hardware accelerator processes messages accordingly. The processing of the Setup message involves checking the TTL field, reading the Routing table to determine the next switch and corresponding output interface, updating the CAC table, reading the Connectivity table to determine the input interface, allocating a connection reference to identify the connection, allocating timeslots and programming the Switch-mapping table. The Setup-Success message requires no special processing. The processing of the Release message involves updating the CAC table, releasing the timeslots reserved for the connection. When processing the Release-Confirm message, the allocated connection reference is freed and thus, the connection is terminated. After processing any message, the State table is updated. The new message is generated and buffered in FIFO0 temporarily, and then transmitted to the next switch on the path.

C. Managing the available timeslots

The management of timeslots and connection references is easy in software through simple array manipulations. However, this poses a challenge in hardware implementations. Our solution is to use a priority decoder.
While our actual implementation only used 32 connections, this is impractical when there are a large number of connections. However, this approach can be generalized if more than 4096 connections are to be handled. A straightforward implementation of a connection reference manager is a bit-vector combined with a priority decoder. The priority decoder finds the first available bit-position (a bit marked as '0'), and updates the bit-vector is written back to the table. We can correspond to the connection reference is reset to 0 and the updated bit-vector is written back to the table. The example shown in Fig. 8, the timeslot manager was asked to find a free timeslot on interface 3. It returns timeslot 14 and marks it as 'used.' De-allocating a timeslot follows a similar pattern but the timeslot number is needed as an input in addition to the interface number in order for the timeslot manager to free the timeslot.

A connection reference is used to identify a connection locally. It is allocated when establishing a connection and deallocated when terminating it. A straightforward implementation of a connection reference manager is a bit-vector combined with a priority decoder. The priority decoder finds the first available bit-position (a bit marked as '0'), sends its index as the connection reference and updates the bit as used (a bit marked as '1'). However, this approach is impractical when there are a large number of connections. While our actual implementation only used 32 connections per switch (see Section IV), we designed the connection reference manager to handle $2^{12}$ simultaneous connections, which requires a bit-vector with 4096 entries. This is too large for the simple priority decoder implementation as used for timeslots.

Our improvement to this basic approach is to use a table with 256 entries of 16-bit vectors to record the availability of a total of 4096 connection references. Fig. 9 illustrates this approach. With 4096 connections, we need a 12-bit connection reference. The first 8 bits of the connection reference correspond to the table pointer, while the remaining 4 bits correspond to the first available connection reference from among the 16 pointed to by the table pointer. The connection reference manager starts with the table pointer set to 0. If any of the 16 connection references corresponding to this row of the table are available (i.e., a bit position is 0), the priority decoder will identify this index and write the output connection reference as a concatenation of the 8-bit table pointer and the 4-bit index extracted. In the example shown in Fig. 9, the 12th bit in the first row is a 0. Therefore it outputs the connection reference number 12. The bit-position is marked as used as illustrated with steps 5–7 of Fig. 9.

De-allocating follows a similar approach; the bit corresponding to the connection reference is reset to 0 and the updated bit-vector is written back to the table. We can parallelize this approach by partitioning the table into several smaller tables, each with a pointer and a priority decoder, forming several smaller managers. All these managers work concurrently. A round-robin style counter can be used to choose a connection reference among the managers. Thus, this approach can be generalized if more than 4096 connections are to be handled.

We developed a prototype VHDL model for the signaling hardware accelerator, used Synplify® tool for synthesizing the design and Xilinx® Alliance tool for the placement and routing of the design. CPE0 (XC4036) uses 62% of its resources while PE1 (XC4013) uses 8% of its resources.

We performed timing simulations of the signaling hardware accelerator using ModelSim® simulator. The simulation results for the Setup message are shown in Fig. 10. It can be seen that while receiving and transmitting a Setup message (requesting a bandwidth of OC-12 at a cross connect rate of OC-1) consumes 12 clock cycles each, processing of the Setup message consumes 53 clock cycles. Overall, this translates into 77 clock cycles to receive, process and transmit a Setup message. We estimate a maximum of 101 clock cycles if multiple (four) routing table lookups are needed.
Assuming a 25 MHz clock, this translates into 3.1 to 4.0 cycles, respectively, which adds to 70 clock cycles. Thus, a complete setup and teardown of a connection consumes about 6.6 microseconds. Compare this with the millisecond-based software implementations of signaling protocols [23]. We are currently optimizing the design to operate at 100 MHz thereby reducing the processing time even further. We are also exploring pipelined processing of signaling messages by selective duplication of the data path to further improve the throughput.

V. CONCLUSIONS

Implementing signaling protocols in hardware poses a considerably larger number of problems than implementing user-plane protocols such as IP, ATM, etc. Our implementation has demonstrated the hardware handling of functions such as parsing out various fields of messages, maintaining state information, managing resources such as timeslots and connection references, writing resource availability tables and switch mapping tables, etc., all of which are operations not encountered when processing IP headers or ATM headers. We also demonstrated that hardware implementation of signaling protocols can result in significant reductions of call handling delays down to a few microseconds. This is especially important in high-speed networks that offer low latencies because call setup delays become a larger percentage of the total overhead. This preliminary implementation of our signaling protocol in VHDL thus shows quite promising results with the potential of achieving a 100X-1000X speedup vis-à-vis software implementations on state-of-the-art processors. Our current work is to implement CR-LDP for SONET networks in hardware.

ACKNOWLEDGMENTS

We thank Reinette Grobler for specifying the signaling protocol, Brian Douglas and Shao Hui for the first VHDL implementation of the protocol.

REFERENCES