A hardware implementation of a signaling protocol

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

ABSTRACT

Signaling protocols in switches are primarily implemented in software for two important reasons. 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, there is an associated performance penalty. 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 we implemented a signaling protocol in reconfigurable FPGA hardware. Our implementation demonstrates the feasibility of 100x-1000x speedup vis-à-vis software implementations on state-of-the-art processors. 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 times.

Keywords: Hardware, Signaling protocols, VHDL, FPGA, SONET/SDH, GMPLS

1. 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\(^1\), User Network Interface (UNI) and Private Network Network Interface (PNNI) signaling protocols in Asynchronous Transfer Mode (ATM) networks\(^2,3\), Label Distribution Protocol (LDP)\(^4\), Constraint-based Routing LDP (CR-LDP)\(^5\) and Resource reServation Protocol (RSVP)\(^6\) in Multi-Protocol Label Switched (MPLS) networks, and the extension of these protocols for Generalized MPLS (GMPLS)\(^7-11\), which supports Synchronous Optical Network (SONET), Synchronous Digital Hierarchy (SDH) and Dense Wavelength Division Multiplexed (DWDM) networks.

Signaling protocols are implemented in the end devices that request the setup and release of connections as well as in the switches of connection-oriented networks. These switches could be circuit-switched, e.g., telephony switches, SONET/SDH switches, DWDM switches, or packet-switched, e.g., MPLS switches, ATM switches, X.25 switches. The end devices requesting the setup/release of connections could be end hosts, e.g., PCs, workstations, or other network switches with interfaces into a connection-oriented network, e.g., Ethernet switches or IP routers with an ATM interface.

Signaling protocol implementations in switches are primarily done 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, signaling protocol implementations are rarely capable of handling over 1000 calls/sec. Correspondingly, call setup delays per switch are in the order of milliseconds\(^12\).

Towards improving performance, we undertook a hardware implementation of a signaling protocol. We used reconfigurable hardware, i.e., Field Programmable Gate Arrays (FPGAs)\(^13,14\) 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 (for example, processing of optional parameters, error handling, etc.) to software.

* haobo_w@photon.poly.edu; phone (718) 260-3384; mv@poly.edu; phone (718) 260-3493; http://kunene.poly.edu/~mv; ramesh@india.poly.edu; phone (718) 260-3596; fax (718) 260-3740; Polytechnic University, 6 Metrotech Center, NY, USA 11201
We modeled the signaling protocol in VHDL* and then mapped onto two FPGAs on the WILDFORCE™ reconfigurable board – a Xilinx® XC4036XLA FPGA with 62% resource utilization and a XC4013XLA with 8% resource utilization. From the timing simulations, we determined that a call can be processed in 6.6µs assuming a 25MHz clock (this includes the processing time for four signaling messages, Setup**, Setup-Success, Release, and Release-Confirm) yielding a call handling capacity of 150,000 calls/sec. Optimizing this implementation will reduce the protocol processing time even further.

The impact of this work is quite far-reaching. By decreasing call processing delays, it becomes conceivable to set up and tear down calls more often leading to a finer granularity of resource sharing and hence better utilization. For example, if a SONET circuit is set up and held for a long duration, given that data traffic using the SONET circuit is bursty, the circuit utilization can be low. However, if fast call setup/teardown is possible, circuits can be dynamically allocated and held for short durations, leading to improved utilization.

Section 2 presents background material on connection setup and teardown procedures and surveys prior work on this topic. Section 3 describes the signaling protocol we implemented in hardware. Section 4 describes our FPGA implementation while Section 5 summarizes our conclusions.

2. 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.

2.1 Background

An end device that needs to communicate with another end device initiates connection setup. When the ingress switch (e.g., switch SW1 in Figure 1) receives such a request, it uses the destination address carried in the Setup message to determine the next-hop switch toward which it should route the connection. This task can be accomplished in different ways. It could be a simple routing table lookup if the routing table is pre-computed. Routing table pre-computation could be done either by a centralized network management station and downloaded to all switches or by a routing process implemented within each switch that processes distributed routing protocol messages and then executes a shortest-path algorithm, such as Bellman-Ford or Dijkstra’s15. Alternately, the signaling protocol processor could perform an on-the-fly route computation upon receipt of a Setup message. Typically switches use a combination of pre-computed route lookups and on-the-fly computation if no pre-computed route exists to meet the requirements of the connection.

After determining the next-hop switch toward which the connection should be routed, each switch performs the following four steps:

1. Check for availability of required resources (link capacity and optionally buffer space) and reserve them.
2. Assign “labels” for the connection. The exact form of the “label” is dependent on the type of connection-oriented network in question. For example, in SONET/SDH switches, the label identifies a time slot, while in ATM networks, it is a Virtual Path Identifier/Virtual Channel Identifier (VPI/VCI) pair.

* VHDL stands for VHSIC Hardware Description Language, where VHSIC stands for Very High Speed Integrated Circuits
** Here we use a generic name for the message, i.e., Setup. Different signaling protocols call this message by different names, e.g., Label request message in LDP.
3. Program the switch fabric to map incoming labels to outgoing labels. This will allow user data bits flowing on the connection after it is set up to be forwarded through the switch fabric based on these configurations. We refer to this configuration information as a \textit{Switch-Mapping} table.

4. Set control parameters for scheduling and other run-time algorithms. For example, in packet switched networks, if weighted fair queueing is used in the switch fabric to schedule packets, the computed equivalent capacity and buffer space allocated for this connection are used to program the scheduler. Even in circuit-switched networks, such as a SONET network, there could be certain parameters. An example is the transparency requirement for how the SONET switch handles bytes in the overhead portions of the incoming and outgoing signals.

In a classical connection setup procedure as illustrated in Figure 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 step should be performed in the forward direction so that resources are reserved as the setup proceeds, but the last three steps could 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.

After connection setup, user-plane data arriving at a switch is forwarded by the switch hardware according to the \textit{Switch-Mapping} table. Upon completion of data exchange, the connection is released with a similar end-to-end release procedure. Typically release messages are also confirmed. Switches processing the release messages free up bandwidth, optionally buffer, and label resources for usage by the next connection.

To support the above-described connection setup and release procedures, signaling messages with parameters in each message, some mandatory and some optional, are defined in a typical signaling protocol. In addition, other messages to support notifications, keep-alive exchanges, etc. are also present in signaling protocols.

With regards to implementation, we illustrate the internal architecture of a switch (unfolded view) in a connection-oriented network in Figure 2. The user-plane hardware consists of a switch fabric and line cards that terminate interfaces carrying user data. In packet switches, the line cards perform network-layer protocol processing to determine how to forward packets. In circuit switches, the line cards are typically multiplexers/demultiplexers. 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 resident on the microprocessor. The routing process handles routing protocol messages and manages routing tables. Network Interface Cards (NICs) are shown in the control-plane unit. These cards are used to process the lower layers of the signaling protocols on which the signaling messages are carried. For example, in SS7 networks, the NICs process the Message Transfer Part (MTP) layers, which are the lower layers of the SS7 protocol stack. In optical networks, the expectation is that an out-of-band IP network will be used to carry signaling messages between switches. In this case, the NICs may be Ethernet cards. It is also possible to carry the signaling messages on the same interface as the user data. An example occurs in ATM networks where signaling messages are carried on VCI 5 within interfaces that carry user data on other virtual channels. Management-plane processing is omitted from this figure, e.g., Management Information Bases (MIBs), agents, etc. Also, all the software processes required for initialization, maintenance of the switch, error handling, etc., and various other details are not shown.

We note that the signaling hardware accelerator unit shown in Figure 2 is part of our proposal and not typical in current-day switches. The illustration in Figure 2 shows that the processing of signaling messages is comparable to packet processing in a packet router, where a Setup message comes in on one interface and is “forwarded” on another interface;
in reality, many actions are performed on the Setup message, which makes the signaling protocol engine more complex than a simple router.

2.2 Prior work

There are many signaling protocols as listed in Section 1. In addition, many other signaling protocols have also been proposed in the literature\textsuperscript{16-25}. Some of these protocols such as Fast Reservation Protocol (FRP)\textsuperscript{25}, fast reservation schemes\textsuperscript{18 19}, YESSIR\textsuperscript{16}, UNITE\textsuperscript{20} and PCC\textsuperscript{21} have been designed to achieve low call setup delays 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 because upgrading the signaling protocol implementation entails a complete redesign of the ASIC. More recently, Molinero-Fernandez and Mckeown\textsuperscript{26} 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 inside switches, it becomes comparable to a signaling protocol for connection setup/release. They are implementing this technique in FPGAs.

3. SIGNALING PROTOCOL

In this section, we describe the signaling protocol that we implemented in hardware. It is not a complete signaling protocol specification because our assumption is that all aspects of the signaling protocol other than those described below will be implemented in the software signaling process shown in Figure 2. Therefore, often in this description, we will leave out details that are handled by the software.

3.1 Signaling messages

We defined a set of four signaling messages, Setup, Setup-Success, Release, and Release-Confirm. Figure 3 illustrates the detailed fields of these four messages.

![Figure 3: Signaling messages](image)

The Setup message is of variable length while the other three messages are of fixed length. The Message Length field specifies the length of the message. The Time-to-Live (TTL) field is used to avoid routing loops. It is initialized by the sender to some value and decremented by every switch along the end-to-end path. If the value reaches 0, a TTL expired error is recognized, error handling is in the part of the protocol implemented in software. The Message Type field is used to distinguish the different messages. The Connection Reference is used to identify a connection locally. The Source IP Address and Destination IP Address specify the end hosts of the connection. The Previous Node’s IP Address specifies the previous node along the connection. The reason we included this field is that the lower layers of the protocol on which these signaling messages are carried may not indicate the sender of the message, but a switch would need to know the downstream switch’s identity in order to process the Setup. The Bandwidth field specifies the bandwidth requirement of the connection. The Interface/timeslot pairs are used to identify the “labels” assigned to the connection, which are used to program the switch fabric. Since there may be an odd number of interface/timeslot pairs, 16-bit Pad Bits field is used to make all messages 32-bit aligned. The Checksum field covers the whole message.

In Setup-Success message, the Bandwidth field records the allocated bandwidth. In Release and Release-Confirm messages, the Cause field explains the reason of release. Some fields are common to all messages, such as Message Length, Message Type and Connection Reference. These fields are in the same relative position for all messages. Such an arrangement simplifies hardware design.
### 3.2 State transition diagram for a connection at a switch

![State transition diagram](image)

Figure 4: State transition diagram

In connection-oriented networks, each connection goes through a certain sequence of states at each switch. The state of each connection must be maintained at each switch. In our protocol, we define four states, **Setup-Sent**, **Established**, **Release-Sent** and **Closed**. Figure 4 shows the state transition diagram of a connection at a switch. Initially, the connection is in the **Closed** state. When a switch accepts a connection request, it allocates a connection reference to identify the connection, reserves the necessary resources including the labels, programs the switch fabric, marks the state associated with the connection as **Setup-Sent** after sending the **Setup** message to the next switch on the path. When the switch receives a **Setup-Success** message for a particular connection, which means all switches along the path have successfully established the connection, then the state of the connection is changed to **Established**. **Release-Sent** means the switch has received the **Release** message, freed the allocated resources, and sent the outgoing **Release** message to the next node. When the switch receives the **Release-Confirm** message, the connection is successfully terminated, and the state of the connection returns to **Closed**.

### 3.3 Data tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Index</th>
<th>Return/Written value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routing table</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Destination address</td>
<td>Next node address</td>
</tr>
<tr>
<td>CAC table</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>Return/Written value</td>
</tr>
<tr>
<td></td>
<td>Next node address</td>
<td>Total bandwidth</td>
</tr>
<tr>
<td>Conn. table</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>Return value</td>
</tr>
<tr>
<td></td>
<td>Neighbor address</td>
<td>Neighbor interface#</td>
</tr>
<tr>
<td>State table</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>Return/Written value</td>
</tr>
<tr>
<td></td>
<td>Own connection reference</td>
<td>Connection reference</td>
</tr>
<tr>
<td></td>
<td>Previous</td>
<td>Next</td>
</tr>
<tr>
<td>Switch mapping table</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>Return/Written value</td>
</tr>
<tr>
<td></td>
<td>Own connection reference</td>
<td>Sequential offset(0 to BW-1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Interface#</td>
</tr>
</tbody>
</table>

Figure 5: Data tables used by the signaling protocol

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 Figure 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 Figure 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 A[9:0].
D[9:0]. We define a generic Switch-Mapping table to emulate this programming interface, with the connection reference as the index, the incoming interface/timeslot pair and the outgoing interface/timeslot pair as the fields.

3.4 Discussion

Aspects of signaling protocols that make it difficult for hardware implementation include the maintaining of state information, the usage of timers, the need to initiate messages from a switch instead of simply forwarding messages (e.g., a release message aborting a connection setup if resources are not available), the Tag-Length-Value (TLV) structure used to carry parameters within messages instead of fixed location fields, choices specific to parameters (e.g., values with global or local significance), and most importantly, the current drive toward generalizing protocols with goal of making them applicable to a large variety of networks.

Starting with the last reason first, consider the evolution of LDP. It has evolved from LDP to CR-LDP to CR-LDP with extensions for GMPLS networks, such as SONET/SDH and DWDM. This complex protocol is now targeting almost all connection-oriented networks both packet-switched and circuit-switched. This drive impacts almost all fields in parameters within messages. For example, the address field identifying the destination address of the connection allows for different address families, IP, telephony E.164, ATM End System Addresses, etc. Next, with regards to choices made for specific parameters, consider a simple parameter such as a connection identifier or connection reference. Most signaling protocols have this parameter. If this is chosen to be globally unique, then connection related data tables need to be searched with a much larger key than if this is chosen to be locally significant. Next, the TLV structure was designed for flexibility, allowing protocol designers to add parameters in arbitrary order. But this construct makes parameter extraction in hardware a complex task. Finally, with regards to state information, signaling protocol engines have to maintain the states of a connection as shown in Figure 4. While the type of state information is quite different, the notion of maintaining some state information is already in practice in IP packet and ATM cell forwarding engines for policing purposes. Other aspects that complicate signaling protocols are the support for a variety of procedures, such as third-party connection control and multiparty connection control.

The signaling protocol described in this section is limited to the part implemented in hardware. Thus, the specification of error handling, aborting setups for lack of resources, checking timers, handling connections more complex than simple two-party connections, etc. have been delegated to the remaining part of the protocol implemented in software. Our approach is to define a large enough subset of the protocol that a significant percentage of users’ requirements can be handled with this subset. Infrequent operations are delegated to the slower software path. Nevertheless, there are many aspects of the complex CR-LDP-like protocols that we have omitted here. Examples include TLV processing, handling larger parameters (such as global connection references, called “label-switched path identifier” in CR-LDP), handling many choices such as the different types of addresses, etc. We are currently implementing CR-LDP for SONET networks in VHDL for an FPGA implementation. This is an NSF-sponsored project \(^27\). At the end of that experiment, we hope to answer the question of whether a complex signaling protocol such as CR-LDP can be implemented in this mode of handling frequent operations in hardware and infrequent operations in software, or whether simpler lightweight signaling protocols targeted for specific networks need to be defined, as we have done here.

4. FPGA-BASED IMPLEMENTATION OF SIGNALING PROTOCOL
To demonstrate the feasibility and advantage of hardware signaling, we implemented a signaling hardware accelerator in FPGA. We used the WILDFORCE™ multi-FPGA reconfigurable computing board shown in Figure 6, which consists of five XC4000XLA series Xilinx® FPGAs, one XC4036XLA (CPE0) and four XC4013XLA (PE1-PE4). These five FPGAs can be used to implement 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 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.

Figure 7 illustrates our prototype implementation. We use CPE0, PE1, FIFO0, FIFO1 and DPRAM. The CPE0 implements the signaling hardware accelerator state machine, the 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.

Figure 7: Implementation of signaling protocol on WILDFORCE™ board

In the following subsections, we describe the design consideration about routing table, and the state transition diagram of the hardware accelerator. We also present two novel approaches for managing timeslots and connection references.

4.1 Routing table look up

In recent years, there has been significant progress in fast table lookup in both research literature and commercial products. Lookup co-processors are widely available, such as Silicon Access Networks’ iAP, SiberCore Technologies’ Ultra-9M, Netlogic Microsystems’ NSE4256, MOSAID Semiconductor’s DC9288, etc. These chips can easily process up to 100 million lookups/sec. In our prototype implementation, we assumed that routing table lookups can be offloaded to an external co-processor, and used equivalent three memory accesses to emulate a routing table lookup.

4.2 State transition diagram of the signaling hardware accelerator

Figure 8 shows the detailed state transition diagram of the signaling hardware accelerator. When a signaling message arrives, it is temporarily buffered in FIFO0. The signaling hardware accelerator 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 signaling 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, and 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 FIFO1 temporarily, and then transmitted to the next switch on the path.
4.3 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.

Figure 9 illustrates our implementation of a timeslot manager. Each entry in the timeslot table is a bit-vector, corresponding to an output interface with the bit-position determining the timeslot number and the bit-value determining availability of the timeslot (‘0’ available, ‘1’ used). The priority decoder is used to select the first available timeslot. When an interface number is provided by the signaling state machine to the timeslot manager, the bit-vector corresponding to the interface is sent into the priority decoder and the first available timeslot is returned. Then the bit corresponding to the timeslot is marked as used (from 0 to 1) and the updated bit-vector is written back to the table. In the example shown in Figure 9, 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.

4.4 Managing the connection references

A connection reference is used to identify a connection locally. It is allocated when establishing a connection and de-allocated 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, 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. Figure 10 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 Figure 10, 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 Figure 10.

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.

4.5 Simulation

We developed a prototype VHDL model for the signaling hardware accelerator, used Synplify® for synthesizing the design and Xilinx® Alliance for the placement and routing of the design. CPE0 (Xilinx® XC4036XLA FPGA) uses 62% of its resources while PE1 (XC4013XLA) uses 8% of its resources.

We performed timing simulations of the signaling hardware accelerator using ModelSim® simulator. The simulation results are shown in Figure 11-Figure 14. From the timing simulation of the Setup message (Figure 11), 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.
Figure 12: Timing simulation for Setup-Success message

Figure 13: Timing simulation for Release message

Figure 14: Timing simulation for Release-Confirm message
Processing Setup-Success (Figure 12), Release (Figure 13) and Release-Confirm (Figure 14) messages consumes about 70 clock cycles total since these messages are much shorter (2 32-bit words versus 11 32-bit words for Setup) and require simpler processing. A detailed breakdown of the clock cycles consumed to process each of these signaling messages is shown in Table 1.

<table>
<thead>
<tr>
<th></th>
<th>Setup</th>
<th>Setup Success</th>
<th>Release</th>
<th>Release Confirm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock cycles</td>
<td>77-101</td>
<td>9</td>
<td>51</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 1: Clock cycles consumed by the various messages

Assuming a 25 MHz clock, this translates into 3.1 to 4.0 microseconds for Setup message processing and about 2.8 microseconds for the combined processing of Setup-Success, Release and Release-Confirm message. 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. 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 selectively duplicating the data path to further improve the throughput.

5. CONCLUSIONS

Implementation of 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, 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 the significant performance gains of hardware implementation of signaling protocols, i.e., call handling within a few µs. Overall, this prototype implementation of a signaling protocol in FPGA hardware has demonstrated the potential for 100x-1000x speedup vis-à-vis software implementations on state-of-the-art processors. Our current work is implementing CR-LDP for SONET networks in hardware.

ACKNOWLEDGMENTS

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.

We thank Reinette Grobler for helping specify the signaling protocol, and Brian Douglas and Shao Hui for the initial prototype VHDL model of the protocol.

REFERENCES


* Based on a worst-case search through a four option routing table


