Dynamic Hardware Plugins (DHP): Exploiting Reconfigurable Hardware for High-Performance Programmable Routers

David E. Taylor [*,1], Jonathan S. Turner [1], John W. Lockwood [1], Edson L. Horta [2]

Applied Research Laboratory
Washington University in Saint Louis
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130
USA
(314)935-4845

Abstract

This paper presents the Dynamic Hardware Plugins (DHP) architecture for implementing multiple networking applications in hardware at programmable routers. By enabling multiple applications to be dynamically loaded into a single hardware device, the DHP architecture provides a scalable mechanism for implementing high-performance programmable routers. The DHP architecture is presented within the context of a programmable router architecture which processes flows in both software and hardware. Implementation options are described as well as the prototype testbed at Washington University in Saint Louis which utilizes the partial reconfiguration capability of modern FPGAs.

Key words: Programmable router, reconfigurable hardware, active networking, port processor, FPGA, partial reconfiguration.

* Corresponding author.
Email addresses: det3@arl.wustl.edu (David E. Taylor), jst@cs.wustl.edu (Jonathan S. Turner), lockwood@arl.wustl.edu (John W. Lockwood), edson-horta@ieee.org (Edson L. Horta).
URL: http://www.arl.wustl.edu/det3 (David E. Taylor).
[1] Research supported in part by NSF ANI-0096052 and Xilinx, Inc
[2] Research supported in part by CNPq (Brazil).

1 Introduction

As researchers vigorously develop advanced control and processing schemes for programmable networks [1], there exists a need for a scalable router architecture capable of robust flow-specific processing at optical line speeds without prohibitively high per-port costs. As the quantity and diversity of streaming data and computationally intensive applications continues to increase, router architectures must respond with greater flexibility, processing capacity, and performance. With next-generation routers containing hundreds of ports, processing mechanisms must scale at a reasonable per-port cost.

Existing router architectures that provide sufficient flexibility and per-flow processing employ software processing environments containing multiple Reduced Instruction Set Computer (RISC) cores. Existing high- performance router architectures capable of data processing at optical line speeds employ Application Specific Integrated Circuits (ASICs) to perform parallel computations in hardware. However, these architectures often provide limited flexibility for deployment of new applications or protocols, and necessitate longer design cycles and higher costs than software-based solutions. Clearly, an optimal router architecture must exhibit the flexibility available in software and the performance offered by hardware.

The diversity of networking applications and data flows suggests that dynamically reprogrammable processing environment is needed to cover the potential design space. While some applications performing limited processing at low data rates readily lend themselves to software implementation, a vast array of applications map well to hardware implementation due to high data rates, data regularities, and parallel operations. This implies that a viable solution to the programmable router problem should employ both software and reconfigurable hardware to process data flows. With the development of several multi-RISC core processor architectures and implementations, the problem of providing a scalable software processing environment is well investigated. The problem of adding a flexible and scalable hardware processing environment remains.

Traditionally used for low-volume prototyping and testing purposes, the reconfigurable hardware employed in Field Programmable Gate Arrays (FPGAs) provides a flexible hardware platform. Recently, reconfigurable hardware technology has made several compelling performance advances, identifying it as a possible solution for the programmable router node problem. New reconfigurable hardware devices tout approximately 1 million application logic gates, internal clock rates up to 200 MHz, over 100KB of on-chip memory, and partial-reconfiguration capability [2]. More impressive than the current technical statistics is the rate of progress due to architectural optimizations

and silicon fabrication improvements: usable logic gate count increased by 10 times in two years; system clock frequency doubled in one year; I/O bandwidth quadrupled in two years; block and distributed on-chip memory capacity quadrupled in one year [3]. Reconfigurable hardware devices are clearly positioning themselves as viable options for flexible, high-performance systems.

The Dynamic Hardware Plugins (DHP) architecture employs reconfigurable hardware to provide a flexible hardware processing environment for programmable, multi-port routers. DHP allows multiple hardware applications, or plugins, to be dynamically loaded into a single device and run in parallel, providing a substantial amount of per-flow processing. With dedicated on-chip logic and memory resources provisioned for each plugin as well as arbitrated access to two types of off-chip memory resources, DHP supports a broad spectrum of applications. Results of several case studies of Advanced Encryption Standard (AES) implementations in software, FPGAs, and ASICs are used to show the potential performance and flexibility gains of the DHP architecture for networking applications in programmable routers.

2 Background and Related Work

Several schemes exist for delivering applications to a programmable router. Applications may be deployed at session setup via signalling protocols. Other schemes allow applications to be requested by incoming packets [4] or carried by the packet for execution on the programmable router [5]. With the exception of implementation details, the programmable router architecture discussion is orthogonal to application deployment mechanisms.

The router architecture presented in [6] provides a scalable software processing environment using elements with multiple RISC cores on a single device. This architecture readily lends itself to hardware processing integration and will be used as the departure point for discussing the DHP architecture.

Significant work has already been done in reconfigurable network hardware; specifically, the P4 developed at the University of Pennsylvania [7]. By dynamically switching FPGAs in and out of the datapath, flows can be routed through chains of applications while other applications are loaded into idle FPGAs. This approach does not readily lend itself to implementation in large routers, as the hardware requirements for a single flow of processing are prohibitively impractical. As applications are restricted to the resources provided on the FPGA, this architecture does not provide ample memory resources to cover the design space of potential applications. The Dynamic Hardware Plugins architecture provides more robust processing in a scalable and efficient way, making it more amenable to implementation in large high-performance

routers.

Taking a more global view of related research, Dynamic Hardware Plugins falls under the general auspices of a reconfigurable hardware system. Implementation options for the DHP architecture are discussed in a later section along with the prototype testbed in which the DHP architecture is implemented in a Xilinx Virtex FPGA. FPGA implementation of a system that uses reconfigurability can be done in two ways: Compile-Time and Run-Time Reconfiguration [8]. For Compile-Time Reconfiguration (CTR) the FPGA does not change configuration during the application lifetime. Each application has specific functions that are loaded when the FPGA is started. Some examples of CTR systems are SPLASH [9] and PAM [10]. For Run-Time Reconfiguration (RTR), the FPGA changes configuration while it is operating. RTR can be total (the entire device is reprogrammed) or partial (only part of the device is reprogrammed). Existing CTR platforms have focused on reconfiguration of entire FPGA devices [11][12][13]. The prototype implementation of the DHP architecture is a contribution to recent work in RTR platforms that consider partial reconfiguration [14][15].

3 Programmable Router Architecture

Current routers capable of aggregate forwarding rates of terabits per second and link speeds of 2.4 Gb/s and 10 Gb/s set the standard for high-performance. Programmable routers need to achieve comparable performance to be considered a viable option for commercial applications. The router architecture described in [6] provides a scalable mechanism for processing data flows at router ports. The DHP architecture will be presented as an augmentation of this architecture to include a hardware processing environment.

As shown in Fig. 1, the programmable router is built around a scalable multistage cell switching fabric as described in [16]. Based on this design, the Switch Fabric may be configured from ten to thousands of ports, each capable of supporting link rates of 2.4 Gb/s. Each physical link attaches to a Transmission Interface (TI) which converts data arriving on the link into a standard format for router input while performing the inverse operation on data destined for the output link. For fiber-optic links, this includes optoelectronic and serial/parallel signal conversion. Between the Transmission Interface and Switch Fabric is the Port Processor (PP). The Port Processor performs all of the flow classification, forwarding, queueing, and processing functions. The Port Processor architecture will be described in the next section. A Control Processor (CP) provides an external control interface and manages the Port Processors. The Control Processor is responsible for maintaining flow classification data structures and filters, as well as binding flows to applications at each Port Pro-

cessor via flow identifiers. In larger systems, the CP may be a shared memory multiprocessor dimensioned to match the processing needs of the specific configuration.

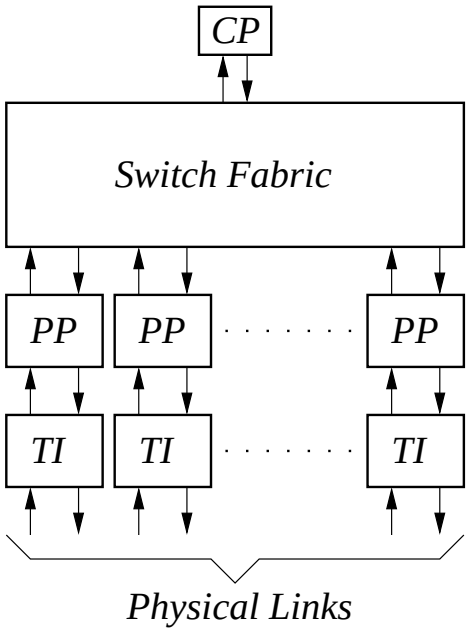

Fig. 1. Programmable router architecture.

4 Port Processor Architecture

The Port Processor provides all of the necessary functionality to forward and process data flows as they pass through the router. The Port Processor architecture is detailed in Fig. 2. The Packet Classification and Queueing (PCQ) element manages the flow of data through three device ports. The TI Port sends and receives data from the Transmission Interface, while the SW Port sends and receives data from the Switch Fabric. Data belonging to flows requiring processing are sent to and received from the processing elements via the Processing Element (PE) Port.

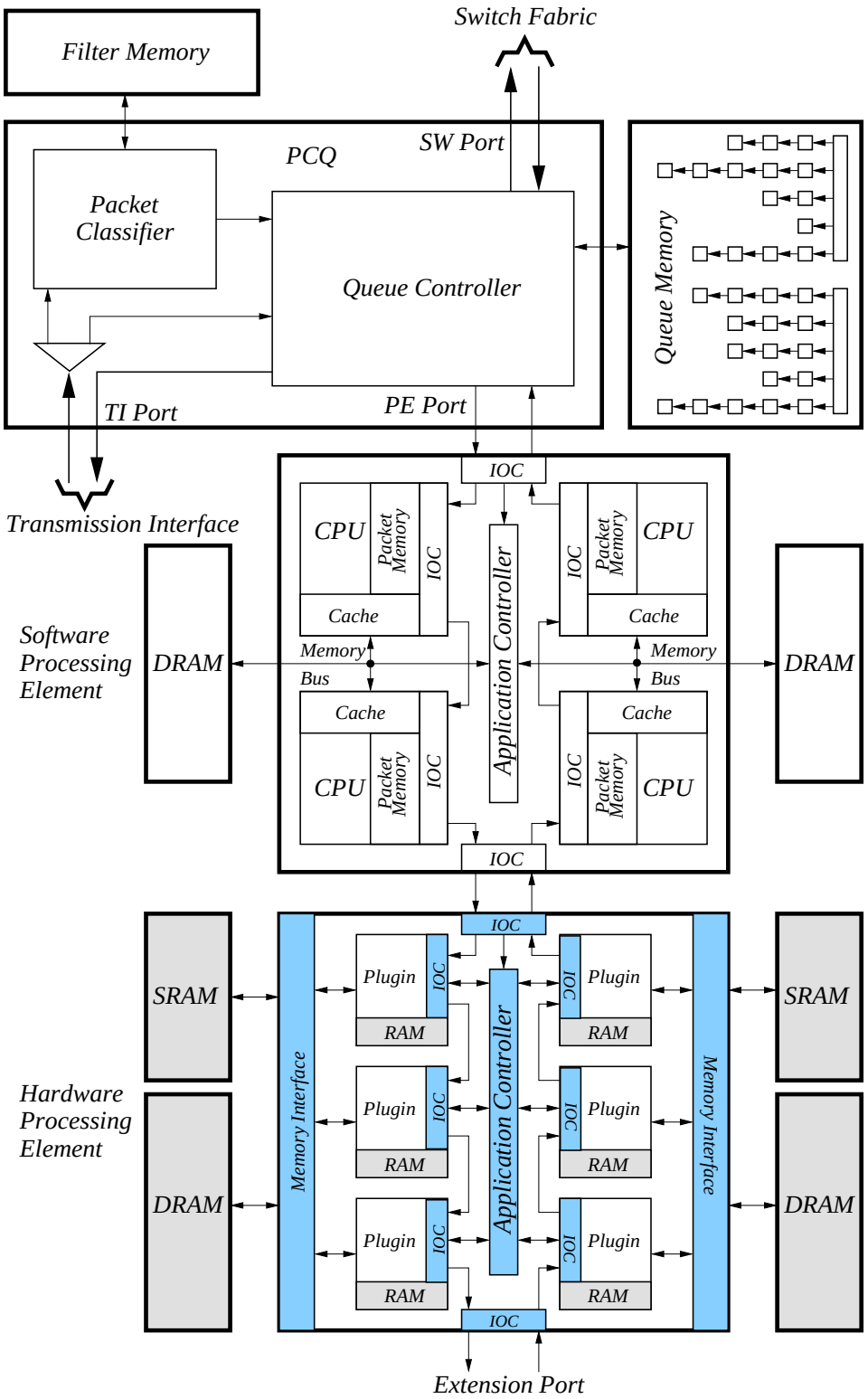

Fig. 2. Port Processor (PP) architecture with Hardware and Software Processing elements. The Hardware Processing Element employs the Dynamic Hardware Plugins (DHP) architecture.

On the PCQ, the Packet Classifier performs a lookup operation on all packets arriving on the TI Port and attaches a flow identifier (flow ID) that identifies the destination of packets and type of processing, if any, that the packet is to receive at the Port Processor. The Packet Classifier maps each packet to a locally significant flow ID that is used to retrieve stored state information at other points in the system. It uses a general packet classification algorithm such as Pruned Tuple Space Search [17]. All data structures and flow IDs are maintained by the central Control Processor of the system.

After classification, packets are sent to the Queue Controller which manages output and application queues. Based on the flow ID, the Queue Controller places the packet on the appropriate queue. Packets not requiring processing are simply placed on the appropriate output queue. Packets requiring processing are placed on the queue associated with the application specified by the flow ID. The Queue Controller schedules packets from the set of application queues for output on the PE Port. Processed packets arriving on the PE Port are placed on the appropriate output queue.

A number of processing elements may reside in a chain at the PE Port of the PCQ. Fig. 2 shows a Software Processing Element followed by a Hardware Processing Element. Note that the quantity and type of processing elements present at a Port Processor may be configured based on traffic demands at a particular port of the router. The Software Processing Element shown in Fig. 2 consists of multiple RISC cores linked by a high speed I/O channel in a ring configuration. Processors are grouped in small clusters for the purpose of sharing access to offchip memory interfaces. This architecture is a refinement of the architecture presented in [6], and is presented here mainly to set the context for the Dynamic Hardware Plugins architecture, which is the focus of this paper.

5 DHP Architecture

The Hardware Processing Element in Fig. 2 utilizes the Dynamic Hardware Plugins (DHP) architecture to add flexible hardware processing capability to the Port Processor. DHP employs reconfigurable hardware to allow multiple applications to be dynamically loaded into hardware plugins and run in parallel on a single device. Data flows may pass through permutations of hardware plugins, allowing for substantial amounts of per-flow processing. In order to support a broad spectrum of applications, each plugin possesses dedicated on-chip logic and memory resources as well as access to two types of arbitrated off-chip memory resources.

In order to facilitate the current architectural discussion and a later discussion

of implementation options, the DHP architecture is divided into two major parts: hardware plugins and infrastructure. Hardware plugins are the hardware components that may be dynamically reconfigured to support new applications. Infrastructure consists of the static control and datapath components of the DHP architecture. The infrastructure components collectively route packets to plugins and I/O ports, dynamically reconfigure hardware plugins, interface to external memory devices and arbitrate access among the contending pool of applications. The following subsections discuss the major divisions of the DHP architecture and their associated components in detail.

5.1 Infrastructure

The infrastructure, denoted by the shaded blocks in Fig. 2, is the required collection of static control and datapath components to support dynamic, modular hardware applications. The infrastructure provides common services to hardware plugins and hides details of memory device timing. By providing a standard interface for plugins, the infrastructure provides the equivalent of an API to allow hardware developers to more easily design modular applications that work together.

5.1.1 Data I/O and Flow Control

As shown in Fig. 2, the DHP architecture arranges hardware plugins in a slotted ring with each ring interface labelled as an Input Output Controller (IOC). A ring architecture was chosen in preference to a bus because rings can be operated at higher clock frequencies than buses due to their simple point-to-point connections and the resulting reduction in capacitive loading. The ring is better in this context than a crossbar since it allows a single plugin to make use of the full ring bandwidth if necessary. A crossbar capable of providing similar bandwidth to each plugin requires substantially more processing resources. While rings do add latency to data transfers, a suitable hardware implementation can keep these latencies to well under a microsecond in typical configurations. In order to keep up with a link rate of 2.4Gb/s, the ring must have a minimum bandwidth of 4.8 Gb/s to allow hardware plugins to process both ingress and egress data flows at the link rate. A 32-bit wide ring operating at 200 MHz provides a raw bandwidth of 6.4 Gb/s, providing sufficient extra bandwidth to handle internal overheads and keep contention low.

Note that an IOC is provided for each hardware plugin while two IOCs interface to upstream and downstream elements. The upstream IOC may interface to another processing element or directly to the PCQ. The downstream IOC

interfaces only to other processing elements. The ring protocol transfers fixed size units with a busy/idle bit in the first word of each transmission slot. The first word also includes a flow control bit vector with one bit for each IOC on the chip. An IOC sets its bit to signal congestion. A second bit vector is used to enable fair access to the ring. Each plugin with data queued for transmission on the ring sets its bit and paces its transmissions on the ring based on the number of bits set by other plugins. Additional fields in this word identify a ring and slot number of the destination application for the packet. The ring number identifies a unique processing element in the chain, while the slot number specifies the hardware plugin containing the destination application. For packets requiring processing by more than one application, the ring and slot number fields are modified to address the next application. Upon completion of a packet, applications identify the correct ring and slot number of subsequent applications via locally available state information.

As shown in Fig. 2 the upstream IOC contains an additional port to the Application Controller. When new applications are to be loaded into the hardware plugins, the upstream IOC must pass control messages and application data to the Application Controller. While a hardware plugin undergoes reconfiguration, the associated IOC passes data to the next IOC in the ring. This mechanism allows applications to be dynamically loaded into hardware plugins without interrupting the flow of data through the processing ring.

5.1.2 Application Controller

The Application Controller manages the dynamic reconfiguration of hardware plugins to support new applications. Hardware applications arrive as bitfiles to the Application Controller. Bitfiles specify the logic operations, signal routing, and on-chip memory configuration for the hardware application. As bitfiles may be loaded from local memory or remotely over the network, the Application Controller must assemble, buffer, and ensure the correctness of the bitfile prior to loading it into the hardware plugin. Bitfile integrity can be maintained via checksums and reliable transport protocols.

Prior to reconfiguring the hardware plugin, the Application Controller initiates a handshake with the application to prevent data and flow state loss. If the application is not idle, it must stop accepting packets and finish processing current packets. Applications may define appropriate breakpoints for reconfiguration based on the type of flow processing it is performing. Control messages may be sent from applications to the PCQ to halt packet forwarding at breakpoints. For deployment of application revisions, applications may copy flow state to off-chip memory for the new revision to use once it has been loaded into the hardware plugin. Once the application has ensured that no data or relevent flow state will be lost, it returns a handshake to the Application Con-

troller. At this point, the IOC routes all arriving packets to the next IOC in the ring. The Application Controller then loads the new application into the hardware plugin by writing the application bitfile to the reconfigurable logic.

The amount of time required for plugin reconfiguration depends on the size of the plugin and the complexity of the application. Current FPGA technologies do not place a strong emphasis on high reconfiguration speeds. However, as discussed in a later section the time required to configure a current generation FPGA with a complex application such as an encryption cipher requires on the order of 5 ms. While this time is not so long as to make DHP impractical with current technology, the current programming rates of 66MB of configuration data per second must increase for next generation technology to be suitable for use in programmable routers. As designers continue to develop systems that demand high-speed device configuration [18], it is likely that FPGA vendors will need to respond with faster reconfiguration mechanisms.

Once all configuration data is loaded into the hardware plugin, the Application Controller initiates a localized reset to the hardware plugin. The Application Controller waits for a handshake from the application. Once the application is initialized and ready, it completes the handshake with the Application Controller. The Application Controller responds with a control message to the CP, which updates the descriptor table used by the Packet Classifier. The IOC then routes packets with matching descriptors to the application.

5.1.3 Memory Interfaces

In order to cover the design space of potential hardware applications, DHP provides access to two types of off-chip memory resources. Banks of Synchronous Random Access Memory (SRAM) provide storage for per flow state and computations requiring low-latency accesses, while banks of Dynamic Random Access Memory (DRAM) provide ample resources for memory intensive applications. The Memory Interfaces shown in Fig. 2 arbitrate access among the hardware plugins while insulating applications from device-specific timing specifications.

The type of hardware technology used to implement the hardware processing element limits the number of pins available for interfacing to off-chip memory devices. Current devices are capable of supporting two SRAM devices and two DRAM devices; therefore, this configuration will be used for the purpose of this discussion. Due to the wide array of memory devices and technologies available, the type of SRAM and DRAM devices employed in a particular system will likely be a function of size, speed, and cost constraints. Systems running applications that require high-bandwidth access to large amounts of memory may employ DRAM technologies, such as Rambus, to meet perfor-

mance constraints [19]. Implementation of such complex memory interfaces requires more on-chip hardware resources and will be discussed in the Implementation section. Other system designers may wish to reduce cost by using Synchronous Dynamic Random Access Memory (SDRAM) devices.

To allow for flexibility in selecting external memory devices, the Memory Interfaces provide a standard interface to hardware plugins insulating them from device-specific timing and control signalling. The Memory Interfaces provide each plugin with independent access to both memory types, hence applications are free to utilize both types of off-chip memory resources in parallel. Hardware plugins gain access to off-chip memory via a simple grant/request handshake. The Memory Interface services requests in a round-robin fashion. Once access is granted, applications may issue read, write, burst read, and burst write commands. Starvation avoidance is achieved by plugins monitoring the status of the grant/request signals. When other plugins contend, the plugin currently accessing memory must release memory at the conclusion of the current transaction.

5.2 Hardware Plugins

Hardware plugins provide applications with the reconfigurable logic and memory resources to process data flows. In this context, hardware plugins are the physical hardware structures that may be configured to implement various networking applications. The reconfigurable logic resources include logic gates, lookup tables, flip-flops, multiplexors, demultiplexors, and signal routing matrices. On-chip Random Access Memory (RAM) may be configured to implement queues and multi-port memories.

In order to design modular applications for use in the DHP, a standardized hardware plugin interface is necessary. Like an API for software, hardware plugins must interface to a static set of ports for data I/O, control, and external memory. As shown in Fig. 3, the hardware plugin interface includes off-chip SRAM and DRAM interfaces, IOC interface, and Application Controller interface. Each application may also define its own interface to on-chip RAM.

The interface to off-chip DRAM includes grant and request signals for the arbitration handshake, memory command signals, address lines, and tri-state data lines. Similarly, the off-chip SRAM interface includes grant and request signals, memory command signals, and address lines. However, this interface employs separate input and output data lines to allow for pipelined memory reads and writes. For low-latency state and data storage, applications may define unique interfaces to on-chip RAM. Reconfigurable hardware technology

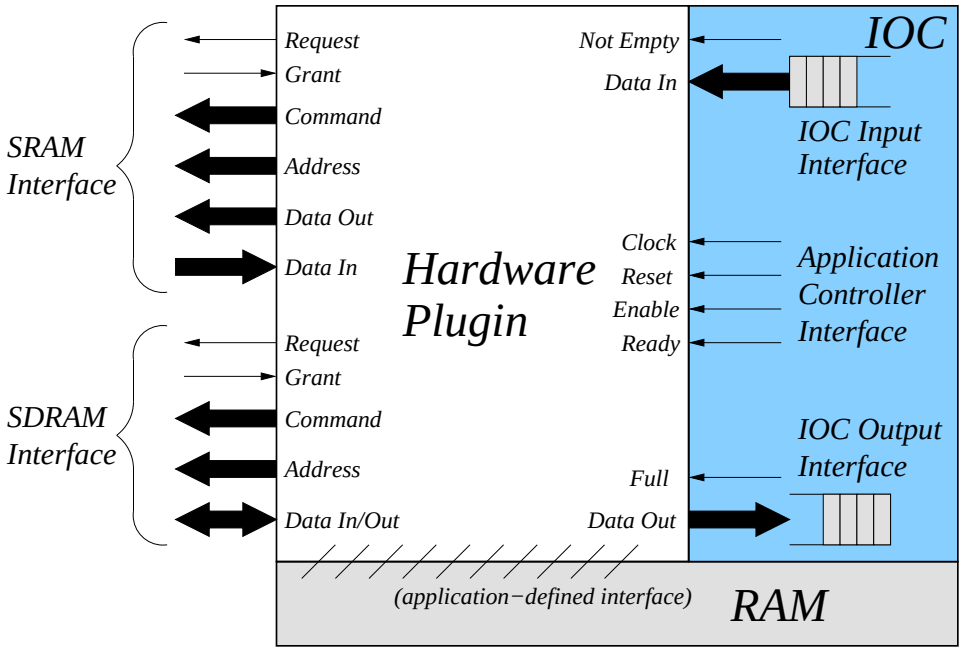

Fig. 3. Hardware Plugin interface with static interfaces to infrastructure components. Applications define interfaces and configurations for on-chip memory.

allows these resources to function as multi-port memories, queues, and large register files.

The IOC interface includes input and output queue interfaces. The input queue interface employs a "not empty" status flag, while the output interface uses a "full" status flag. To keep the design uniform, the queue data paths are the same width as the ring. The Application Controller interface provides applications with a system clock, local reset, and enable/ready signals for the reconfiguration handshake with the Application Controller. Applications may use subdivided or multiplied versions of the system clock to suit design needs.

6 Applications & Performance

While the focus of this paper is not a performance comparison of hardware and software applications, it is important to identify the types of applications that benefit from the DHP architecture. Any computationally intensive application operating on streaming data at high rates is a likely candidate. Potential applications also need to contain operations that may be performed in parallel or pipelined. Purely sequential computations cannot take full advantage of the inherent benefits of hardware implementations.

One of the most widely used applications which is also crucial to the growth of the Internet as a commercial tool is encryption. Since every byte must be

manipulated in order to properly encrypt a data block, encryption is a computationally expensive application. Due to the nature of the computations performed, encryption is highly amenable to hardware implementation. Results of case studies of the new Advanced Encryption Standard (AES) will be used to illustrate the potential performance of DHP for networking applications.

In order to select an algorithm for AES, the National Institute of Standards and Technology (NIST) and several independent research groups analyzed the security and performance of the finalist algorithms for both software and hardware implementations [20][21][22]. Based on these analyses, Rijndael was selected as the algorithm for AES [23]. In order to provide a baseline performance comparison of software and FPGAs, the authors of [21] implemented and analyzed an iterative version of the Rijndael algorithm that provided encryption, decryption, and key-scheduling for 128-bit keys operating over 10 rounds on 128-bit data blocks in a Xilinx FPGA. This implementation achieved a throughput of 353 Mb/s, providing a factor of 11.15 speedup over comparable software implementations that achieved 31.64 Mb/s. This implementation would occupy approximately 20% of the available resources of the largest current generation FPGA, a Xilinx Virtex 3200E, and would require on the order of 5 ms for device configuration. While these results show significant performance gains, NIST cited case studies of ASIC implementations of the Rijndael algorithm achieving throughputs of 5.16 Gb/s, a factor of 163 speedup over software [20]. This level of performance was achieved through fully pipelined architectures as opposed to the iterative architectures used in [21]. Fully pipelined architectures require significantly more resources, making them impractical for use in current generation FPGAs and the DHP architecture. However, for implementation in programmable routers a higher performance FPGA implementation of AES is required.

The authors of [22] implemented several architectural variants of the Rijndael algorithm in an FPGA. Their analysis focused solely on encryption throughput, operating under the assumption that key-scheduling delays can be masked by a suitable parallel implementation. This analysis is relevant to the programmable router discussion, as throughput is the metric of interest and it is likely that encryption and decryption will occur in separate hardware plugins. In this analysis, the authors found that a 5-stage partial-pipeline with a single-stage sub-pipeline architecture of Rijndael algorithm achieved a throughput of 1.94 Gb/s. While this implementation required nearly twice the amount of device resources as the iterative implementation in the aforementioned study, it occupies less than 40% of the largest current generation FPGA.

Based on these results, a single hardware plugin could encrypt 80% of the traffic carried on an OC-48 link. Achieving this level of performance in software would require distributing the computation over 60 RISC cores, an exorbitant amount of resources for a single application operating on a single link. These

results clearly strengthen the case for employing reconfigurable hardware in reprogrammable routers. Unlike ASIC implementations, the DHP architecture allows for new encryption standards such as AES to be deployed in a matter of milliseconds.

New streaming data services such as audio and video bridging for video conferencing also provide ideal hardware plugin applications. Multi-service routing and multicast support are also ideal candidates for hardware implementation. With the proliferation of Hardware Description Languages (HDLs) as common tools for designing hardware applications, many applications currently implemented in ASICs can be easily ported to DHP implementation.

7 Implementation

Due to strides in current FPGA technology, the Dynamic Hardware Plugins architecture can be implemented in a single FPGA. As device speeds and densities continue to increase, the quantity and performance capabilities of hardware plugins available on a single device will likewise increase. Providing dynamic, modular plugins surrounded by static control structures in a single device physically translates to partially reprogramming a running FPGA at the port of a router. This is a non-trivial task that is the focus of ongoing research at Washington University in Saint Louis. A significant part of the solution involves new CAD tools capable of targeting specific regions of a device, producing partial reprogramming bitfiles, reserving logic and routing resources, and locking signals for static plugin interfaces. While many of these capabilities exist in one form or another within current CAD tool suites, execution of this task requires an enormous amount of effort. This significant area of research will be examined in the following section.

Due to the homogeneous nature of reconfigurable logic, FPGA implementation of static infrastructure components provides lower efficiency and performance than a comparable ASIC implementation. Another limitation lies in off-chip data transfers. The types of memory devices available to system designers are limited by those conforming to I/O standards supported by FPGA vendors. Adding high-performance memory interfaces, such as Rambus, could provide substantially better performance. While an FPGA is a readily available device for implementing the DHP architecture, the challenges and inefficiences encourage the pursuit for a better solution.

An intriguing implementation option for the DHP architecture is a mixed ASIC/FPGA device with regions of reconfigurable logic embedded into application specific silicon. By hand-crafting the IOC ring, Application Controller, and Memory Interfaces in ASIC technology, greater I/O performance could

be achieved for on- and off-chip data transfers and memory transactions as well as faster plugin configuration. Given the same die size, this would also result in a more area-efficient infrastructure implementation providing more silicon area for reconfigurable logic; hence, more logic and memory resources per hardware plugin or more plugin slots per device. Such implementation options are the focus of ongoing research in this area.

8 Prototype Testbed

In order to prototype the Dynamic Hardware Plugins architecture operating in a Port Processor of a multi-port programmable router, several research systems designed and built at Washington University in Saint Louis are used in combination [24]. The WUGS 20, an 8 port ATM switch providing 20 Gb/s of aggregate throughput, is used for the Switch Fabric. This switching core is based upon a multi-stage Benes topology, supports up to 2.4 Gb/s link rates, and scales up to 4096 ports for an aggregate throughput of 9.8 Tb/s [25].

The Smart Port Card (SPC) is used to prototype the software processing element [26]. It employs an embedded microprocessor, memory, and custom network interface ASIC, the APIC, to process network data flows in software. Data flows requiring processing are identified by their ATM Virtual Circuit Identifier (VCI). The APIC writes the data from these flows directly to system memory via the PCI bus. Once in memory the embedded microprocessor processes the data, then triggers the APIC to transmit.

The Field Programmable Port Extender (FPX) is used to prototype the Dynamic Hardware Plugins architecture [27][28]. It employs two FPGAs, one acting as the Network Interface Device (NID) and the other as the Reprogrammable Application Device (RAD). The RAD FPGA has access to two 1MB Zero-Byte Turnaround (ZBT) SRAMs and two 64MB SDRAM modules. A diagram of the FPX is shown in Fig. 4. Both the SPC and FPX are implemented on Printed Circuit Boards (PCBs) of the same form factor as the WUGS transmission interfaces. Hence, each port of the WUGS may be fitted with different FPX/SPC combinations. A photograph of an FPX is shown in Fig. 5. A photograph of an FPX in the WUGS is shown in Fig. 6.

As these research systems were designed to be sufficiently general for use in diverse research areas, mapping the DHP architecture to this testbed requires a partitioning of infrastructure components across several devices. Fig. 7 details the mapping of the DHP architecture onto the prototype testbed at Washington University. At a coarse level, the SPC functions as the software processing element while the FPX functions as the hardware processing element and Packet Classification and Queueing (PCQ) element. In this context, all ingress

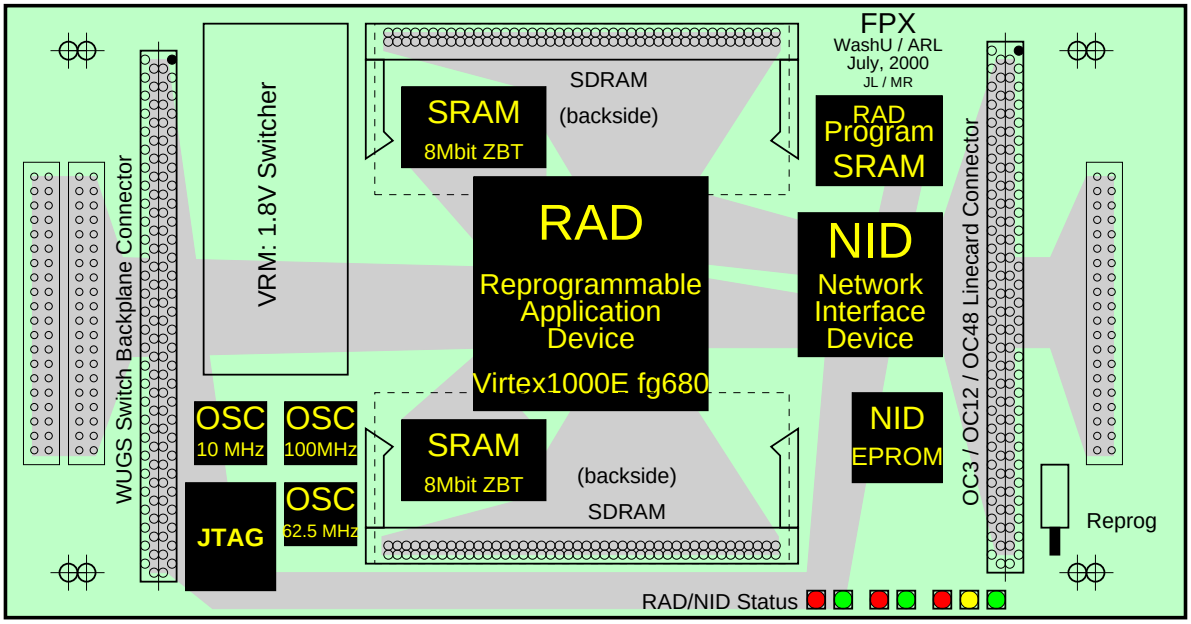

Fig. 4. Diagram of the Field Programmable port eXtender (FPX) used to prototype the Dynamic Hardware Plugins (DHP) architecture.

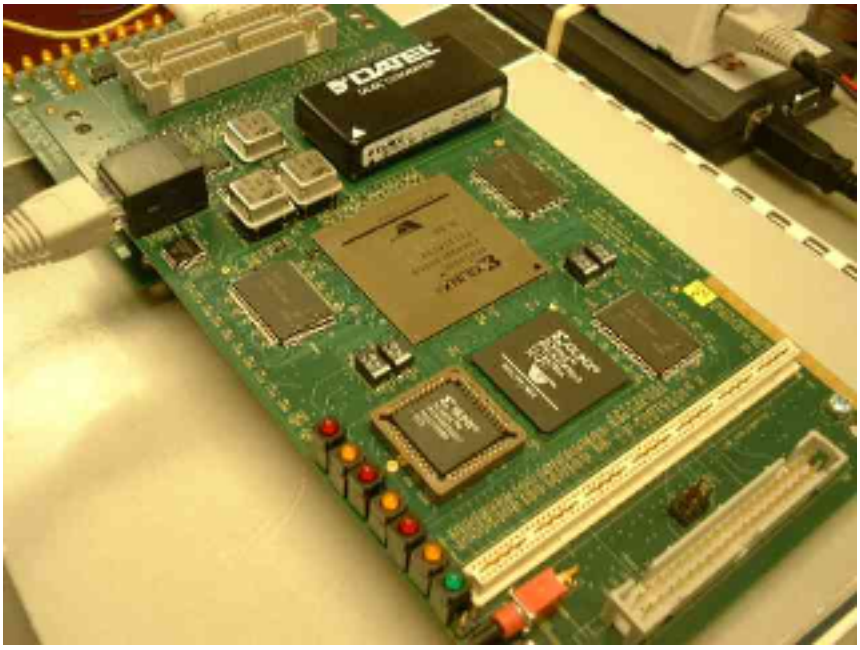

Fig. 5. Photograph of the Field Programmable port eXtender (FPX)

traffic must be sent to the FPX for classification and queueing; hence, cells arriving from the Line Card must tunnel through the APIC. Upon arrival at the FPX, the NID switches the ATM cells containing packets of the data flow to the PCQ port of the RAD. Utilizing the ZBT SRAM, the Packet Classifier performs a lookup on the packet header fields and assigns locally unique flow identifiers. These flow identifiers are subsequently used by the Queue

Controller to allocate and schedule queues in the SDRAM. The Control Cell Processor parses control cells for on-chip control register and off-chip memory updates for PCQ initialization and control. Data flows requiring processing in the SPC are sent back through the NID to the APIC where packet contents are copied directly to system memory via the PCI bus. The embedded microprocessor then processes the packets and triggers the APIC for transmission upon completion. Data flows requiring hardware processing are sent to the NID, switched to the DHP port of the RAD, and placed on the IOC ring for processing.

As the FPX does not provide mechanisms for partial self-configuration, the Application Controller is implemented in the NID FPGA. Applications are loaded into the RAD FPGA by sending control cells containing configuration bitfiles to the NID FPGA. The Application Controller parses the control cells to extract the bitfile and buffers the data in an off-chip SRAM. Upon receipt of the complete bitfile and a program command, the Application Controller programs the RAD FPGA using the SelectMAP programming interface [29]. This interfaces provides the fastest available mechanism for configuring the RAD FPGA. In order to update or re-configure a single hardware plugin, a specific region of the RAD FPGA must be reprogrammed while allowing the remaining regions to continue operating. This requires the use of the partial reconfiguration capability of the Xilinx Virtex FPGA, along with custom CAD tools to generate the correct configuration bitfiles.

Fig. 6. Prototype environment for the DHP architecture; photograph of an FPX in a WUGS with the line card removed for visibility.

8.1 PARBIT

In order to partially reconfigure an FPGA, it is necessary to isolate a specific area inside the device and download the configuration bits related to that area. A tool called PARBIT [30] has been developed to easily transform and restructure bitfiles to enable implementation of dynamically loadable hardware applications.

8.1.1 Virtex Architecture

PARBIT leverages knowledge of the Xilinx Virtex FPGA architecture, programming mechanisms, and partial reconfiguration mechanisms in order to perform these tasks [31]. The Virtex architecture contains three types of logic resources: Configurable Logic Blocks (CLBs), BlockRAM, and Input/Output Blocks (IOBs). The CLBs contain lookup tables (LUTs) and logic resources that can be programmed to implement user-defined functions. The IOBs (Input/Output Blocks) may be configured to conform to various I/O standards

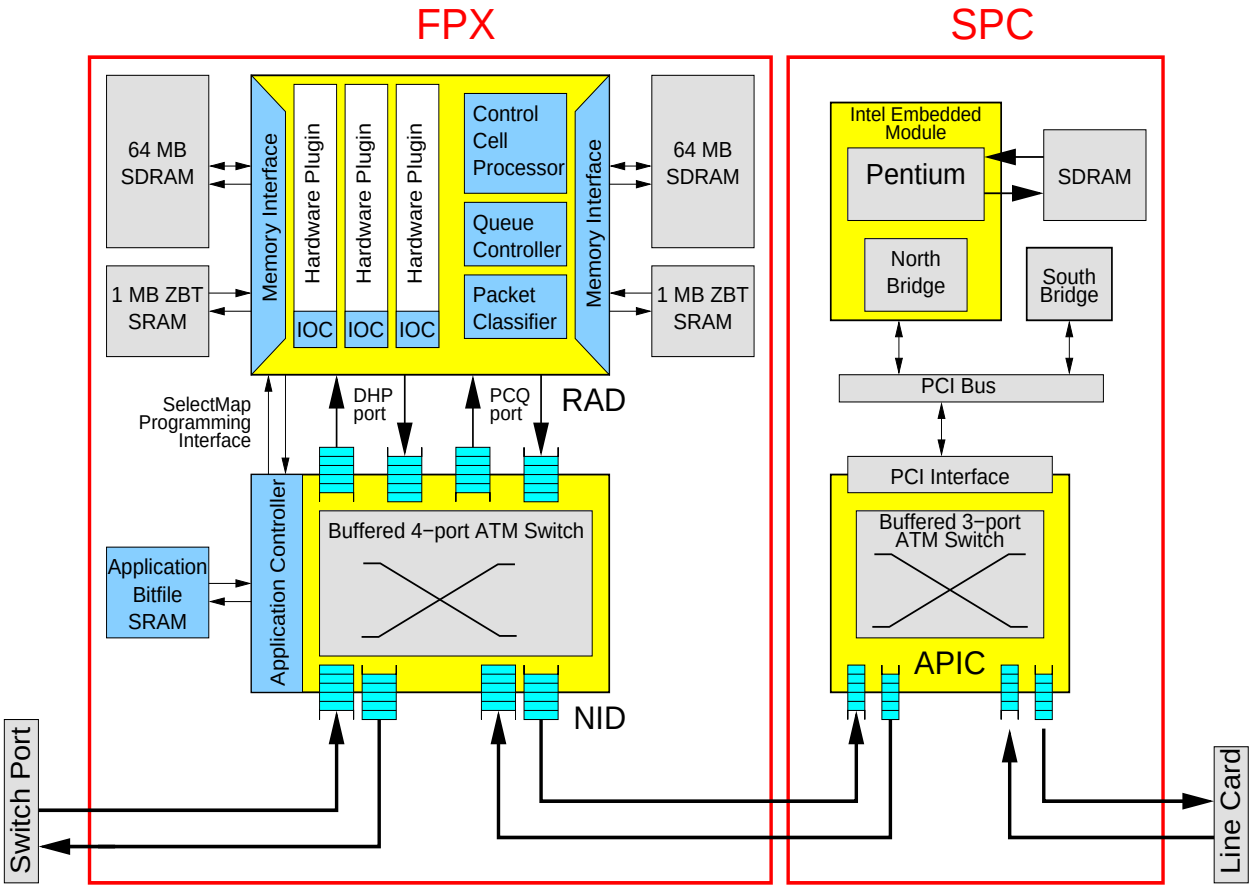

Fig. 7. Block diagram of the prototype implementation of the DHP architecture in the Washington University testbed.

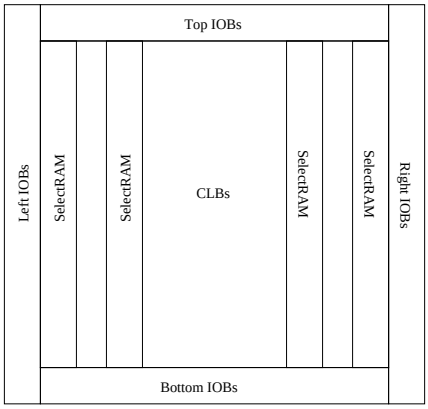

Fig. 8. Architecture of the Xilinx Virtex FPGA.

in order to interconnect the logic to off-chip resources. A block diagram of the Xilinx Virtex architecture can be seen in Fig. 8.

Configuration memory that specifies the use of on-chip resources is divided in vertical slices, called columns. There are five types of columns:

- Center - controls the global clock pins;
- IOB (Input/Output Block) - controls the configuration for the left and right side IOBs;
- CLB (Configurable Logic Block) - controls one column of CLBs and two IOBs above and bellow these CLBs. Each column has "n" rows, with one CLB per row;
- Block SelectRAM Interconnect - defines the interconnection of each RAM column;
- Block SelectRAM Content - defines the contents of each RAM column;

To configure the Virtex FPGA, a series of bits, divided into fields of commands and data, are loaded into the device. The data field can program the contents of each configuration column. The minimum amount of a column that can be reconfigured is a vertical slice, one-bit wide, called a frame. Fig. 9 shows the configuration columns for one specific device, the Xilinx Virtex-E 1000 (XCV1000E).

8.1.2 Operation

PARBIT allows arbitrary block regions of a compiled FPGA design to be re-targeted into any similar size region of the FPGA. It is possible to define an area inside the CLB columns of the chip, without the top and bottom IOB configuration bits. The tool generates the partial bitfile containing the area selected by the user from the original bitfile. This file will be used to reconfigure the FPGA.

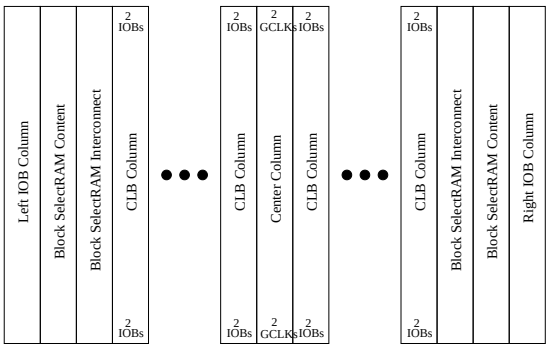

Fig. 9. Configuration of columns in the Xilinx Virtex-E 1000 (XCV1000E).

A target bitfile is used by PARBIT to copy the configuration bits that are inside a column specified by the user, but outside the partial reconfigurable area. This happens due to the fact that one frame takes all the rows of a column and the partial reconfigurable area is smaller than a whole column.

To generate a partial configuration bitfile, the tool utilizes the original bitfile, the target bitfile, and parameters given by the user. These parameters include the physical coordinates of the logic implemented on the FPGA, the coordinates of the target area for partial reprogramming, and the programming options. For application revisions and modifications, PARBIT also reallocates logic within the target partial reconfigurable area. The tool calculates new values for the configuration address registers [31] and modifies the bitfile, such that the original area can be reallocated in another region of the FPGA.

8.1.3 Using PARBIT to Implement DHP

Applications targeted to hardware plugins on the FPX are first built with the standard design methodology of running CAD tools to compile, place, and route logic into a fixed region of an XCV1000E or XCV2000E FPGA. After generating the source bitfile, PARBIT is run to transform the source file into a partial bitfile. This bitfile contains the application to be implemented in a hardware plugin of the DHP architecture.

PARBIT reads the region of the chip that contains the hardware plugin from the original bitfile. This file contains the partial reconfiguration area that will be loaded into the device. The hardware plugin must always remain in same location. In order to do this, the user must confine the hardware plugin between static CLB coordinates in the FPGA: start column, end column, start row and end row. The original bitfile, if viewed from an FPGA chip editor, has the format shown in Fig. 10.

The target design specifies empty areas to load the block design generated in the previous step. Each one of these areas is defined by two coordinates (Row,

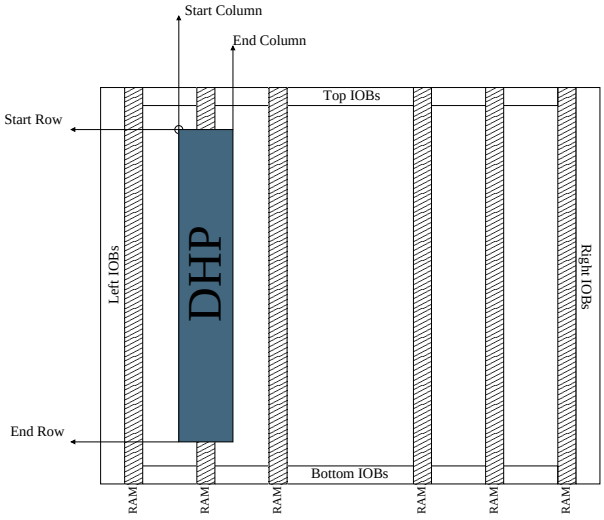

Fig. 10. DHP - Original Bitfile

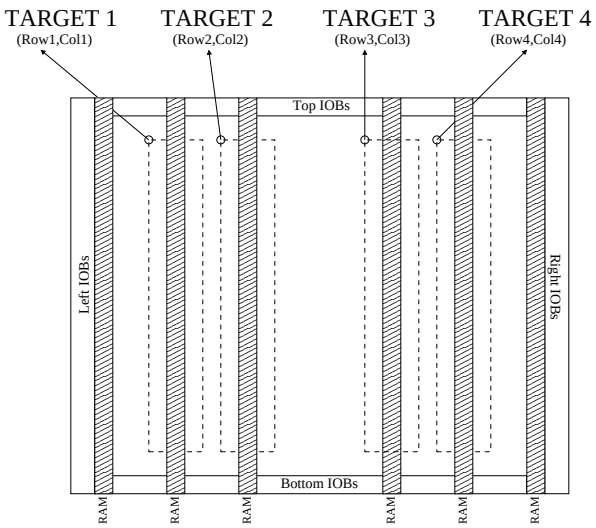

Fig. 11. DHP - Target Bitfile

Col), as shown in Fig. 11. The area surrounding the empty target locations is used by the DHP infrastructure.

9 Conclusion

Dynamic Hardware Plugins provides a scalable mechanism for building high-performance, multi-port routers capable of robust per flow processing. As reconfigurable hardware technology continues to offer higher performance via denser logic and memory resources at faster clock rates, the amount and diversity of per flow processing made available by the DHP architecture likewise increases. Implementing networking applications in hardware provides perfor-

mance levels either not achievable in software, or achievable only with significantly more hardware resources and complex control mechanisms. By allowing multiple hardware applications to be dynamically loaded into a single device, the DHP architecture is a flexible, parallel, hardware processing mechanism. As applications are developed, the prototype testbed at Washington University in Saint Louis provides an ideal platform for performance analysis and further research into reconfigurable network hardware.

10 Acknowledgment

The authors would like to thank David Parlour and Xilinx, Inc. for their support and efforts to aid in this research. The authors would also like to thank Naji Naufel for his contributions to the FPX project.

References

[1] D. L. Tennenhouse et al., "A Survey of Active Network Research," in *IEEE Communications Magazine*, pp. 80–86, Jan. 1997.

[2] Xilinx Inc., "Virtex-e 1.8v field programmable gate arrays," Feb. 2000.

[3] Xilinx Inc., "Xilinx unveils new fpga architecture to enable high-performance, ten million system-gate designs," May 2000.

[4] D. Decasper and B. Plattner, "DAN: Distributed Code Caching for Active Networks," in *INFOCOM '98*, 1998.

[5] D. L. Tennenhouse and D. J. Wetherall, "Towards an Active Network Architecture," *Computer Communication Review*, vol. 26, no. 2, 1996.

[6] T. Wolf and J. Turner, "Design issues for high performance active routers," in *Proceedings of International Zurich Seminar on Broadband Communications*, (Zurich, Switzerland), February 2000.

[7] I. Hadzic, W. Marcus, and J. Smith, "On-the-Fly Programmable Hardware for Networks," 1998.

[8] B. L. Hutchings and M. J. Wirthlin, "Implementation approaches for reconfigurable logic applications," in *Field-Programmable Logic and Applications (FPL'1995)* (W. Moore and W. Luk, eds.), (Oxford, England), pp. 419–428, Springer-Verlag, Berlin, Aug. 1995.

[9] D. T. Hoang, "Searching genetic databases on splash 2," in *IEEE Workshop on FPGAs for Custom Computing Machines* (D. A. Buell and K. L. Pocek, eds.), (Los Alamitos, CA), pp. 185–191, IEEE Computer Society Press, 1993.

[10] P. Bertin, H. Touati, and E. Lagnese, "PAM programming environments: Practice and experience," in *IEEE Workshop on FPGAs for Custom Computing Machines* (D. A. Buell and K. L. Pocek, eds.), (Los Alamitos, CA), pp. 133–138, IEEE Computer Society Press, 1994.

[11] J. M. Ditmar, "A Dynamically Reconfigurable FPGA-based Content Addressable Memory for IP Characterization," Master's thesis, KTH- Royal Institute of Technology, Stockholm, Sweden, 2000.

[12] J. D. Hadley and B. L. Hutchings, "Designing a partially reconfigured system," in *Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, Proc. SPIE 2607* (J. Schewel, ed.), (Bellingham, WA), pp. 210–220, SPIE – The International Society for Optical Engineering, 1995.

[13] D. Ross, O. Vellacott, and M. Turner, "An FPGA-based Hardware Accelerator for Image Processing," in *More FPGAs: Proceedings of the 1993 International workshop on field-programmable logic and applications* (W. Moore and W. Luk, eds.), (Oxford, England), pp. 299–306, 1993.

[14] S. McMillan and S. Guccione, "Partial run-time reconfiguration using JRTR," in *Field-Programmable Logic and Applications / The Roadmap to Reconfigurable Computing (FPL'2000)*, (Villach, Austria), pp. 352–360, Aug. 2000.

[15] E. L. Horta and S. T. Kofuji, "The architecture of a reconfigurable ATM switch (RECATS)," in *Workshop de Computação Reconfigurável*, (Marília, SP, Brazil), Aug. 2000.

[16] J. Turner, T. Chaney, A. Fingerhut, and M. Flucke, "Design of a Gigabit ATM Switch," in *In Proceedings of Infocom 97*, Mar. 1997.

[17] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *SIGCOMM 99*, pp. 135–146, 1999.

[18] H. Schmit, "Incremental reconfiguration for pipelined applications," in *IEEE Symposium on FPGAs for Custom Computing Machines* (K. L. Pocek and J. Arnold, eds.), (Los Alamitos, CA), pp. 47–55, IEEE Computer Society Press, 1997.

[19] R. Warmke, "Designing a Multimedia Subsystem with Rambus DRAMs," in *Multimedia Systems Design*, Miller Freeman, Inc., March 1998.

[20] J. Nechvatal, E. Barker, L. Bassham, W. Burr, M. Dworkin, J. Foti, and E. Roback, "Report on the Development of the Advanced Encryption Standard (AES)," in *Computer Security Division Information Technology Laboratory*, National Institute of Standards and Technology, Technology Administration, U.S. Department of Commerce, October 2000.

[21] A. Dandalis, V. Prasanna, and J. Rolim, "A comparative study of performance of AES final candidates using FPGAs," in *Third AES Candidate Conference*, Mar. 2000.

[22] A. Elbirt, W. Yip, B. Chetwynd, and C. Paar, "An FPGA implementation and performance evaluation of the AES block cipher candidate algorithm finalists," in *Third Advanced Encryption Standard (AES) Conference*, 2000.

[23] J. Daemen and V. Rijmen, "AES proposal: Rijndael," in *NIST AES Proposal*, 1998.

[24] J. S. Turner, "Gigabit Technology Distribution Program." http://www.arl.wustl.edu/gigabitkits/kits.html, Aug. 1999.

[25] S. Choi, J. Dehart, R. Keller, J. W. Lockwood, J. Turner, and T. Wolf, "Design of a flexible open platform for high performance active networks," in *Allerton Conference*, (Champaign, IL), 1999.

[26] D. Decasper, G. Parulkar, S. Choi, J. DeHart, T. Wolf, and B. Plattner, "A scalable high-performance active network node," in *IEEE Network, Vol. 13, No. 1*, January/February 1999.

[27] J. W. Lockwood, J. S. Turner, and D. E. Taylor, "Field programmable port extender (FPX) for distributed routing and queuing," in *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2000)*, (Monterey, CA, USA), pp. 137–144, Feb. 2000.

[28] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor, "Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX)," in *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2001)*, (Monterey, CA, USA), pp. 87–93, Feb. 2001.

[29] C. Carmichael, "Virtex configuration and readback." Xilinx XAPP138, Mar. 1999.

[30] E. Horta and J. W. Lockwood, "PARBIT: a tool to transform bitfiles to implement partial reconfiguration of field programmable gate arrays (FPGAs)," Tech. Rep. WUCS-01-13, Washington University in Saint Louis, Department of Computer Science, July 6, 2001.

[31] S. Kelem, "Virtex configuration architecture advanced user's guide." Xilinx XAPP151, Sept. 1999.