Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 872610, 11 pages
doi:10.1155/2012/872610

Research Article

Redsharc: A Programming Model and On-Chip Network for Multi-Core Systems on a Programmable Chip

William V. Kritikos,[1] Andrew G. Schmidt,[1] Ron Sass,[1] Erik K. Anderson,[2] and Matthew French[2]

[1] *Reconfigurable Computing Systems Laboartory, ECE Deptartment, UNC Charlotte, 9201 University City Boulevad, Charlotte, NC 28223, USA*
[2] *Information Sciences Institute, University of Southern California, 3811 North Fairfax Drive, Suite 200, Arlington, VA 22203, USA*

Correspondence should be addressed to William V. Kritikos, will.kritikos@gmail.com

Received 6 May 2011; Revised 21 July 2011; Accepted 28 September 2011

Academic Editor: Claudia Feregrino

The reconfigurable data-stream hardware software architecture (Redsharc) is a programming model and network-on-a-chip solution designed to scale to meet the performance needs of multi-core Systems on a programmable chip (MCSoPC). Redsharc uses an abstract API that allows programmers to develop systems of simultaneously executing kernels, in software and/or hardware, that communicate over a seamless interface. Redsharc incorporates two on-chip networks that directly implement the API to support high-performance systems with numerous hardware kernels. This paper documents the API, describes the common infrastructure, and quantifies the performance of a complete implementation. Furthermore, the overhead, in terms of resource utilization, is reported along with the ability to integrate hard and soft processor cores with purely hardware kernels being demonstrated.

1. Introduction

Since the resources found on FPGA devices continue to track Moore's Law, modern, high-end chips provide hundreds of millions of equivalent transistors in the form of reconfigurable logic, memory, multipliers, processors, and a litany of increasingly sophisticated hard IP cores. As a result, engineers are turning to multi-core systems on a programmable chip (MCSoPC) solutions to leverage these FPGA resources. MCSoPC allow system designers to mix hard processors, soft processors, third party IP, or custom hardware cores all within a single FPGA. In this work, we are only considering multi-core systems with a single processor core, multiple third party IP cores, and multiple custom hardware cores.

A major challenge of MCSoPC is how to achieve intercore communication without sacrificing performance. This problem is compounded by the realization that cores may use different computational and communication models; threads running on a processor communicate much differently than cores running within the FPGA fabric. Furthermore, standard on-chip interconnects for FPGAs do not scale well and cannot be optimized for specific programming models; contention on a bus can quickly limit performance.

To address these issues, this paper investigates Redsharc—an API and common infrastructure for realizing MCSoPC designs. Redsharc's contribution has two parts.

First, introduction of an abstract programming model and API that specifically targets MCSoPC is presented. An abstract API, as described by Jerraya and Wolf in [1], allows cores to exchange data without knowing how the opposite core is implemented. In a Redsharc system, computational units, known as kernels, are implemented as either software threads running on a processor, or hardware cores running in the FPGA fabric. Regardless of location, kernels communicate and synchronize using Redsharc's abstract API. Redsharc's API is based both on a streaming model to pass data using unidirectional queues and a block model that allows kernels to exchange index-based bidirectional data. Section 3 explains the Redsharc API in detail.

Redsharc's second contribution is the development of fast and scalable on-chip networks to implement the Redsharc

API, with special consideration to the hardware/software nature of MCSoPC. The stream switch network (SSN), discussed in Section 4.1, is a run-time reconfigurable crossbar on-chip network designed to carry streams of data between heterogeneous cores. The block switch network (BSN), discussed in Section 4.2, is a routable crossbar on-chip network designed to exchange index data elements between cores and blocks of memory.

To evaluate the proposed approach, the API and infrastructure have been implemented as a software library and a collection of VHDL components with generics. A series of benchmarks were run on a Xilinx ML510 development board to demonstrate Redsharc's performance. A BLAST bioinformatics kernel was also ported to a Redsharc core to demonstrate the full use of the Redsharc API and to show scalability. Results are listed in Section 6.

2. Related Work

The idea of creating an API to abstract communication between heterogeneous computational units has its roots in both the DARPA adaptive computing systems (ACSs) project [2] and PipeRench project [3]. These projects focused on processor to off-chip FPGA accelerator communication and control, they existed prior to the realization of MCSoPC.

More recently, ReconOS [4] and hthreads [5] implemented a programming model based on Pthreads to abstract the hardware/software boundary. They successfully demonstrate the feasibility of abstract programming models in heterogeneous systems. However, their middleware layer requires an impractical amount of FPGA resources. Furthermore, their communication bandwidth is limited due to a reliance on proprietary buses. Redsharc addresses these problems by developing an abstract API better suited for MCSoPC and developing custom on-chip networks, respectively, supporting the API.

Using a message passing interface (MPI) for parallel communication has also been explored on FPGAs. MPI is commonly used in scientific applications running on large clusters with traditional microprocessor-based nodes. Several works, including TMD-MPI [6] and SoC-MPI [7], have implemented some of the standard MPI function calls in an FPGA library. While supporting MPI may be useful when trying to port an existing scientific application to an FPGA cluster, it adds several layers of abstraction that are not necessary for streaming models. Some logic must be present to decode received message, strip out the real data from the message, and deliver that data to the appropriate input of the computational unit. The streaming model allows for direct connection of a stream to computational units, as the stream contains only data. We depend on the streams being properly set up before data is delivered. The goal of Redsharc is to implement as light-weight as possible, so a streaming system was chosen over an MPI or Pthreads-based system.

The stream model is common within FPGA programming. Projects such as [8, 9] use high level stream APIs or stream languages with compilation tools to automatically generate MCSoPC. Unlike Redsharc, these efforts focus on a purely streaming programming model and do not include support for random access memory. Furthermore, these models do not permit the mixing of heterogeneous hard processors, soft processors, or custom FPGA cores as may be needed to meet stringent application demands.

Researchers have also customized streaming networks to communicate between heterogeneous on-chip cores using abstract interfaces. For example, aSOC [10] is notable for communication between heterogeneous computational units, but it targets ASICs instead of FPGAs. Finally, SIMPPL [11] developed an FPGA-based streaming network but do not consider heterogeneous computational units. Moreover, existing on-chip networks only support pure streaming applications and do not include support for random access memory.

3. Redsharc API

Developing an abstract API for MCSoPC is not a trivial task. The API must support parallel operations, be realizable in all computational domains, be resource friendly, and be flexible enough to incorporate a large number of application domains. However, these goals can be in conflict at times.

To find the middle ground in these conflicting requirements, Redsharc is based on the stream virtual machine (SVM) API [12]. SVM was originally designed as an intermediate language between high level stream languages and low level instruction sets of various architectures being developed by the DARPA polymorphous computing systems (PCSs) program. The main idea was that multiple high-level languages would map to a common stable architectural abstraction layer that included SVM. That layer would then map to specific polymorphous computing systems such as TRIPS, MONARC, RAW, and others as illustrated in Figure 1 [13–15]. SVM has no preference to the computational model for individual kernels and only specifies how kernels communicate with each other. SVM is primarily based on the relatively simple stream model, but it includes the concept of indexed block data to support random access. These features make it an ideal candidate for porting to MCSoPC. The REDSHARC system implements the same SVM API that would be used on a TRIPS, MONARCH, or RAW architectures while targeting FGPAs instead of ASIC processors. Future work could allow the high level languages shown in Figure 1 to target FPGA-, TRIPS-, MONARCH-, and RAW-based processors.

The Redsharc API recognizes two types of kernels, worker and control. There is only one control kernel in the system and it is responsible for creating and managing streams, blocks, and worker kernels. There may be multiple worker kernels that perform functions on data elements presented in streams or blocks. A stream is a unidirectional flow of data elements between two kernels. A block is an indexed set of data elements shared between two or more kernels. In the current Redsharc implementation, a data element can be of any size 2^n bits where n is an integer and greater than 5.

Applications are divided into kernels. Because both hardware and software kernels utilize the same block and stream API, the initial stages of application development do not need to be concerned with a particular kernel

Table 1: Redsharc control kernel's API calls.

Control's API call	Description
`kernelInitNull (kernel *k)`	Initializes `k` to a default state
`kernelInit (kernel *k, streams *s, blocks *b)`	Configures streams/blocks in array `s/b` for communication with `k`
`kernel Add Dependence (kernel *k1, kernel *k2)`	Makes `k2` dependent on the completion of `k1`
`kernelRun(kernel *k)`	Adds `k` to the list of schedulable kernels
`kernelPause(kernel *k)`	Removes `k` from the list of schedulable kernels
`streamInitFifo(stream *s)`	Initializes `s`
`blockInit(block *b)`	Initializes `b`

Table 2: Redsharc worker kernel's API calls.

Worker's API Call	Description
`void kernelEnd()`	Indicates kernel has completed its work
`void streamPush(element *e, stream *s)`	Pushes `e` onto a stream `s`
`void streamPushMulticast(element *e, stream *s)`	Pushes `e` to multiple streams
`void streamPop(element *e, stream *s)`	Pops top element from `s` and stores value in `e`
`void streamPeek(element *e, stream *s)`	Reads top element from `s` storing value in `e`
`void blockWrite(element *e, int index, block *b)`	Writes `e` to `b` at index
`void blockRead(element *e, int index, block *b)`	Reads `index b` and stores in `e`

being implemented in software or hardware. The system can be viewed as kernels with logical connections as shown in Figure 2. After each kernel is defined, the kernels are implemented using either hardware or software, depending on the suitability of the kernel's task to a software or hardware environment. At that point in time, the system will resemble Figure 3.

3.1. Software Kernel Interface. There are two types of software kernels in Redsharc, worker kernels which perform work for the application, and a single control kernel which sets up the blocks, streams, and which manages the other kernels. Control kernels are always implemented in software. The software kernel interface (SWKI) is implemented as a traditional software library. User kernels link against the library when generating executable files. Different processors (hard or soft) may implement the SWKI in different ways, however, a common approach has been to use a threading model to manage multiple software kernels executing on a single processor.

The control kernel API, in C syntax, is presented in Table 1. The symbols, `kernel`, `stream`, `block`, and `element`, are variable types in Redsharc. The control kernel creates variables of these types and initializes them for use during run-time. In Redsharc, the streams and blocks a kernel communicates with are set at run-time with the `kernelInit` command. A dependency is used in conjunction with a block to allow one kernel to write elements to a block and prohibit any reading kernel from starting until the write is complete. Only the control kernel is aware of each worker kernel's location, either hardware or software. Due to the added complexity control, kernels may only be implemented in software.

The worker kernel API is listed in Table 2. While `kernelEnd` is used to communicate with the control kernel when work is completed, the remainder of the API is dedicated to stream or block communication. In much the same way that a function in an object-oriented language may be overloaded for different variable types, the Redsharc API is independent of data element width, block location, or the transmitting or receiving kernel's implementation.

3.2. Hardware Kernel Interface. The hardware API layer, known as the hardware kernel interface (HWKI), is implemented as a VHDL entity that is included during synthesis. There is one HWKI for each hardware kernel. The HWKI is a thin wrapper that connects hardware kernels to the SSN and BSN. Described in more detail in Section 4, the HWKI implements the Redsharc stream API as a series of interfaces similar to FIFO ports, and the Redsharc block API as a series of interfaces similar to BRAM ports. The use of SVM has simplified the development of hardware kernels. Figure 4 illustrates the HWKI as implemented in VHDL. A kernel developer only needs to know how to pop and push data from an FIFO to use the Redsharc stream switch network. Similarly, all accesses to memory, on-chip, or off-chip, are only a simple BRAM access with additional latency.

4. Redsharc's Networks on a Chip

The stream switch network (SSN) and block switch network (BSN) were necessitated by performance and scalability. Programming models that are built on top of existing general-purpose on-chip networks such as IBM's CoreConnect [16] must translate higher level API calls to lower level network procedures often with a large overhead. For example, even a simple `streamPeek()` operation may require two bus read transactions, the first to check if the stream is not empty, the second to retrieve the value of the top element. In previous work, Liang et al., [10] showed custom on-chip networks

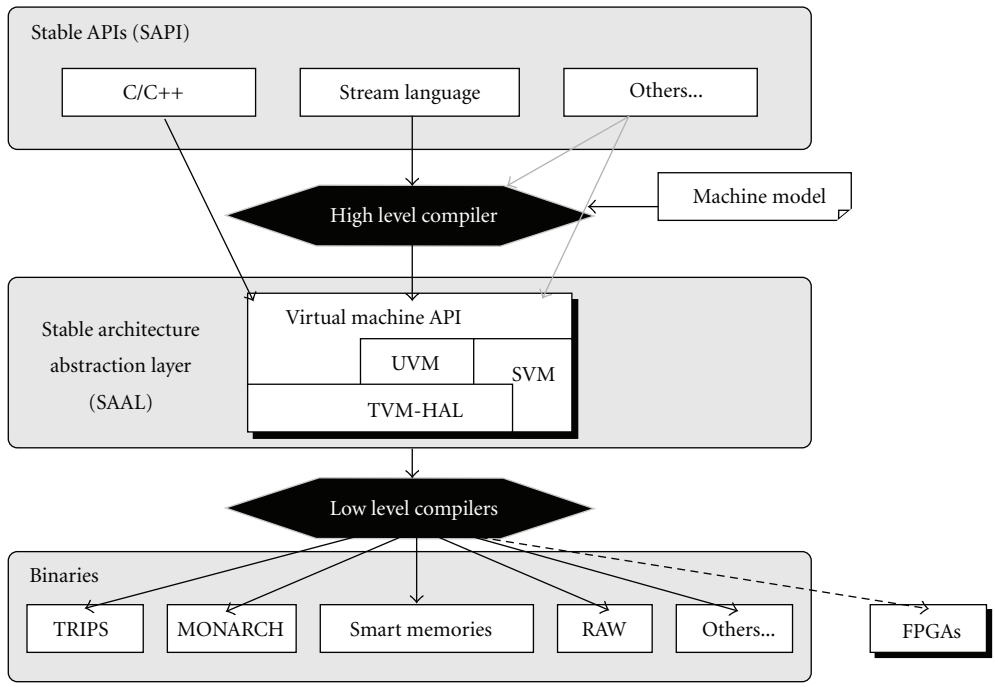

Figure 1: Layers in PCS program.

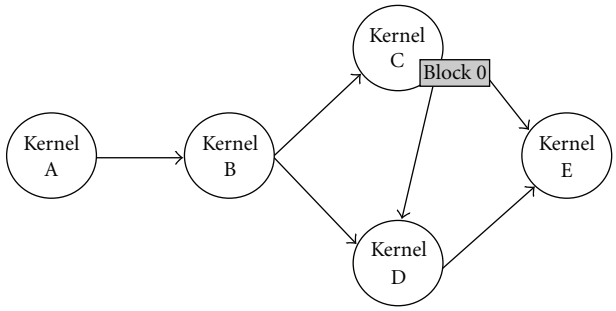

Figure 2: Logical view.

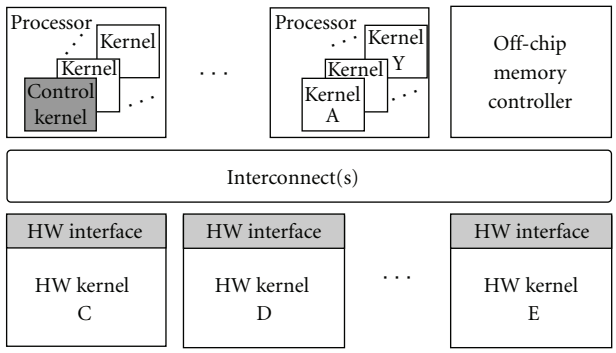

Figure 3: Physcial view.

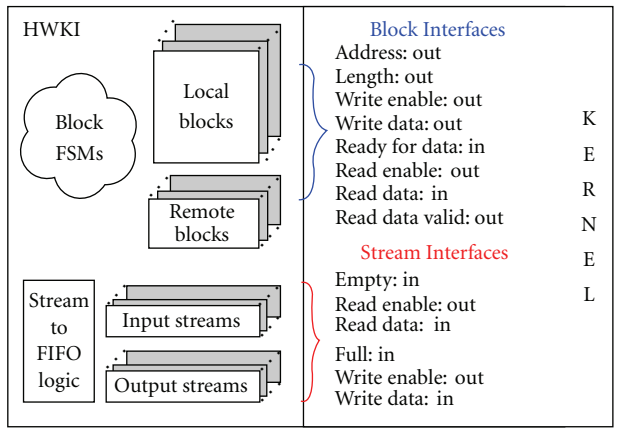

Figure 4: HWKI.

can outperform general-purpose networks. Therefore, the SSN and BSN were designed and implemented specifically to support the Redsharc API and thereby improving scalability and performance.

4.1. Stream Switch Network. The Redsharc stream switch network is a dedicated on-chip network designed to transport streams of data between communicating kernels. The SSN connects to all hardware kernels via the HWKI and the processor using DMA controllers. Software kernels communicate with the SSN through the SWKI which uses DMA descriptors to send and receive data. The SSN is composed of a full crossbar switch, configuration registers, FIFO buffers, and flow control logic.

The SSN's crossbar switch is implemented using a parametrized VHDL model which sets the number of inputs and outputs to the switch at synthesis time. Each output port of the crossbar is implemented as large multiplexers in the FPGA fabric. The inputs and outputs of the switch are buffered with FIFOs allowing each hardware kernel

to run within its own clock domain. Data is transferred from a kernel's output FIFO to another kernel's input FIFO whenever data is available in the output FIFO and room exists in the input FIFO. The flow control signals used in the SSN are based on the LocalLink specification [17]. The SSN's internal data width is set at synthesis time and defaults to 32 bits. To fulfill the Redsharc API requirement that each stream has a variable width, the FIFOs on the edges of the SSN translate the stream's data from 32 bits to the kernel's data width requirement. The SSN runs by default at 200 MHz, however, this system clock can be reduced to improve the timing score for very dense designs.

Figure 5 shows the streams and kernels in an SSN system. The input streams are shown on the left side of the figure flowing into hardware kernels, with the output streams on the right. Hardware kernels may have any number of input and output streams. Also shown are the streams connecting the processor's DMA controllers ports which connect directly to the switch, along with the switch configuration registers which are accessible from the system bus.

Hardware kernels are presented with standard FIFO interfaces as their Redsharc stream API implementation. While abstracted from hardware developers the FIFOs are directly part of the SSN.

Software kernels communicate with the SWKI library. The SWKI in turn communicates with hardware kernels by interacting with the DMA controllers. The DMA controllers can read or write data from off-chip memory into a LocalLink port. The SWKI's stream API is written to send a pointer and length to the DMA engine for the amount of data to send or receive and the location of that data. An interrupt occurs when the DMA transaction has finished. Using the DMA controllers, the processor is more efficient when there is a large amount of data to push from software to hardware or vice versa, which is often the case with streaming applications. The SWKI is also responsible for configuring the SSN's switch to connect the processor with the receiving or transmitting hardware kernel.

An advantage of the crossbar switch is that multicast stream pushes are possible by simply having several output ports reading data from the same input port. The control kernel can change the switch configuration at run-time to modify application functionality or for load balancing optimizations.

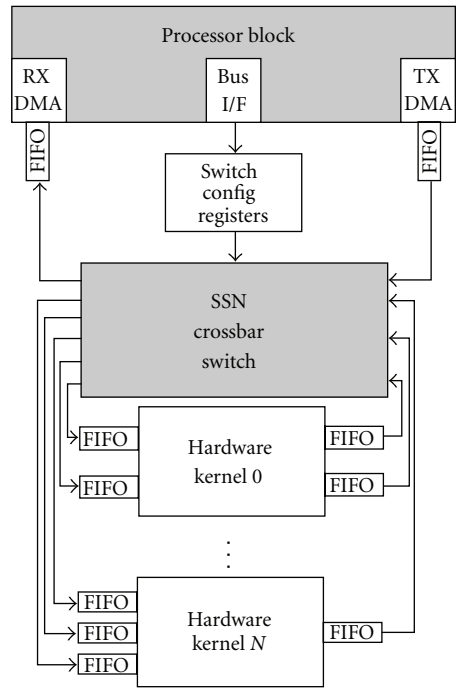

Figure 5: Stream switch network.

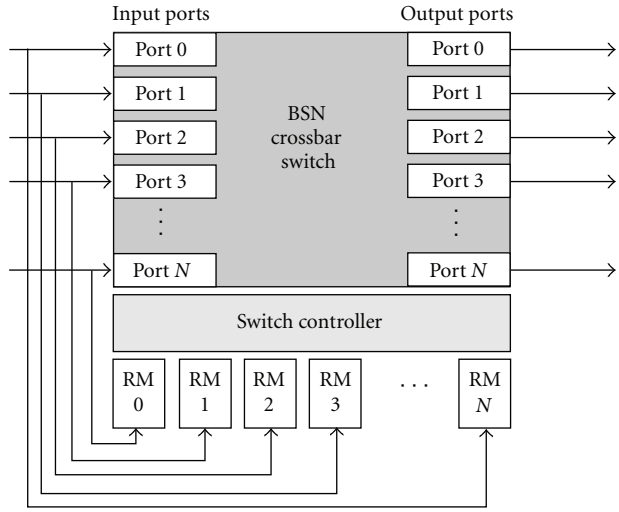

Figure 6: Block switch network's router consisting of a full crossbar switch, routing modules per each input port, and a single switch controller used to configure the inputs to the requested outputs.

4.2. Block Switch Network. The purpose of the block switch network is to implement the low level communication needed by the Redsharc block API. Redsharc blocks are created at synthesis time but allocated at run time by the control kernel. Blocks may be specified to be on-chip as part of an HWKI or a section of off-chip volatile memory. The BSN is implemented as a routable crossbar switch and permits access from any kernel to any block.

Figure 6 shows the BSN's router. The router consists of a full crossbar switch, switch controller, and routing modules. When a kernel requests data, the routing module decodes the address request and notifies the switch controller. The switch controller checks the availability of the output port and, if not in use, configures the switch. This configuration can take as few as 2 clock cycles. The configuration will stall if the desired output port is currently in use by another input until the first port finishes the transmission. The input ports are shown on the left side of the figure which are connected to the output of the hardware kernels. The output ports on the right connect to the input port of the hardware kernels. Hardware kernels can consist of any number of local, remote and off-chip memory ports, giving the kernel a higher degree of parallelism with memory access.

"Local" blocks, commonly implemented as BRAM, are accessible to a hardware kernels with low latency (≈2

clock cycles). They are instantiated as part of a hardware kernel's HWKI. A "remote" block is on-chip memory that is located in a different hardware kernel's HWKI, but accessible through the BSN. This is made possible through dual-ported BRAMs. One port is dedicated to the local hardware kernel, the second port is dedicated to the BSN. "Off-chip" blocks are allocated in volatile off-chip RAM (e.g., DDR2 SDRAM). The BSN communicates directly with the memory controller. While hardware kernels still share this connection for block communication, requests are serialized at the last possible point helping to improve overall performance.

The BSN abstracts away the different block types to provide hardware developers with a common interface via the HWKI. The HWKI block interface is an extension of the common BRAM interface with the addition of "ready for data," "valid," "request size," and "done." The added signals offer the ability to burst larger amounts of data in and out of memory. The use of the ready for data and valid signals is necessary since requests to remote and off-chip memory may take an undetermined amount of time.

The BSN also abstracts away data (element) sizes from the kernel developer. In conventional bus-based systems, the developer may need to access 32-bit data from one location and 128-bit data from a second location. As a result, the developer must be aware of the amount of data needed for each request type. In the Redsharc system, the BSN, gives the developer the exact data size needed and handles the internal transfers accordingly. With the BSN a developer can still transfer 128-bit data, but instead of actively transmitting the four 32-bit words, only a single transaction is required. The BSN still transfers all 128-bits; however, it does so internally as a burst of four 32-bit words.

Another advantage is the block location can be moved (e.g., from a local block to a off-chip block) at synthesis time based on available resources without requiring the kernel developer to redesign their code. For example, if the design requires a significant amount of BRAMs to be used elsewhere in the system, the data can be placed in off-chip memory. To the kernel developer the interface remains fixed and only the connection within the HWKI changes.

Software kernels access blocks through the SWKI. If the block is located off-chip, the SWKI routes the request through the PPC's memory port (avoiding bus arbitration). If the block is located within the BSN, the request is translated to memory mapped commands to the BSN on the PLB.

While blocks are conceptualized as either being stored in on-chip block RAMs or in off-chip RAM, many designs also need access to a set of small control or "one off" registers to interface with peripheral or kernels. We group these together as special purpose registers. For example, a design may need access to one or more of these registers in order to set some initial parameters, such as sequence length or number of transfers. Rather than dedicating an entire block, a designer may want to have the more familiar register access within the kernel. The BSN enables access to these registers by both the control kernel and other hard and soft kernels in the system. A designer can specify at synthesis the number of registers which are to be allocated for the kernel. To the hardware kernel interface, the registers are additional elements with a specific address range. However, to the kernel, the registers are directly accessible without performing block transfers.

5. Implementation

The Redsharc infrastructure spans several IP cores and incorporates multiple interfaces. Nonetheless, by utilizing VHDL generics and by careful organization, Redsharc can be synthesized for multiple FPGA families. Specifically, this is possible because of four key principles. First, it simplifies hardware and software kernel development through an abstract programming model that has been implemented in a conventional software and hardware kernel interface (SWKI and HWKI, resp.). Second, Redsharc can use both hard and soft processing cores to control kernel execution. Third, by design, Redsharc enables low-latency and high-bandwidth access to streams of data between hardware and software kernels through the stream switch network (SSN). Last, having a separate network for data transfers to random access block memory (through the block switch network, BSN) reduces resource contention.

5.1. Kernel Integration. Among its many contributions, Redsharc enables rapid kernel integration into new and existing systems. Through the use of the hardware kernel interface (HWKI), a kernel developer can exclusively focus on the design of the kernel rather than how to access streams and blocks of data. Complexities associated with bus transactions and memory controllers need not be considered by the kernel developer. Instead, the Redsharc infrastructure simplifies these accesses to mere FIFO and BRAM interfaces. The HWKI is then responsible for translating these requests into the more complex stream and block transfers. This is especially important as systems migrate from on FPGA to the next because modifying low-level kernels to access memory should be the last concern for a system designer.

5.2. System Integration. Separate from kernel development is system integration. Redsharc emphasizes the difference in order to enable designers to more efficiently construct large systems comprising of several kernels that are executing in both hardware and software. The goal is to allow the rapid assembly of such systems without significant involvement from individual kernel developers. In fact, so long as the kernel developer adheres to FIFO- and BRAM-like interfaces, the system designer is free to assemble and modify the system.

Ultimately, a Redsharc system is comprised of one or more processors, memory controllers, and kernels. The processors can be hard or soft IP, such as the PowerPC, MicroBlaze, or Nios processor. While performance and/or available resources may dictate which processor to use, the Redsharc system has been designed to be completely processor and FPGA vendor agnostic. The LocalLink interface standard used in the SSN and BSN can be replaced with a similar point to point streaming protocol with flow control. The internal LocalLink channels in the SSN and BSN would not need to be modified to accept a new streaming protocol,

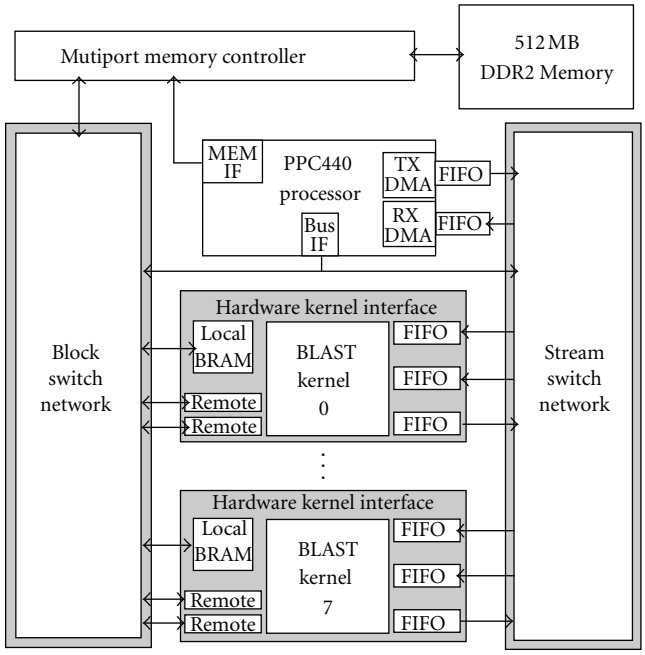

Figure 7: PowerPC BLAST system.

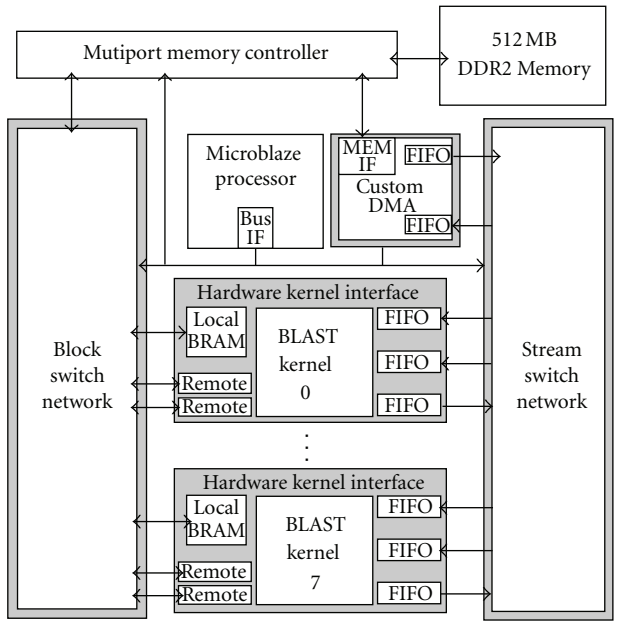

Figure 8: MicroBlaze BLAST system.

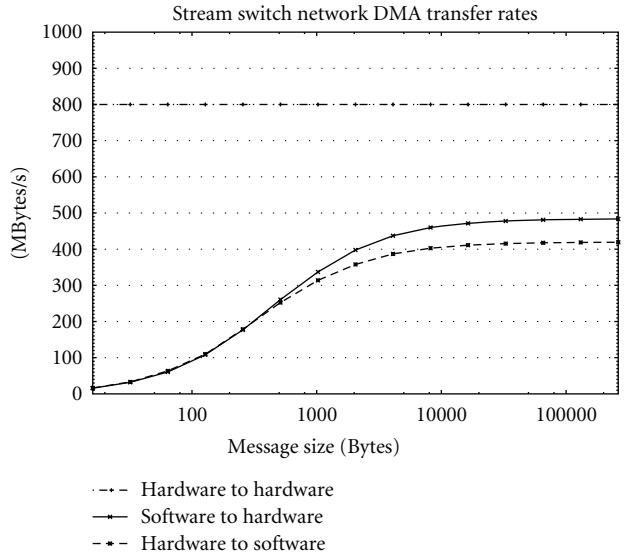

Figure 9: Stream bandwidth across hardware and software with 200 MHz SSN clock.

only the specific interfaces that connect between the SSN and BSN switches with the processors or system bus of the system.

The only requirement of the processor is that it needs to have software-addressable registers to configure the SSN and BSN networks. This is easily accomplished over a system bus interface. The BSN needs direct access to the external memory controller for high efficiency. Both Xilinx and Altera FPGAs provide such an interface for user logic. In this paper, we have implemented Redsharc using two processor systems, a PowerPC 440 hard processor and MicroBlaze soft processor. Both systems were implemented on a Xilinx ML510 development board which is based on a Virtex 5 FX 130T FPGA. Figures 7 and 8 show these two systems.

5.3. Memory Interface. While a variety of interfaces to memory exist, Redsharc is best suited to be directly connected to on-chip and off-chip memory controllers. Specifically, both the SSN and BSN have separate interfaces to the memory controller to enable independent access, reducing contention, and simplifying arbitration for the individual resource.

The SSN is connected to the processor core via two FIFOs for stream communication and a set of registers to configure the SSN. In this implementation, we have used DMA to transfer a buffer from software, in DDR2 main memory, to or from the SSN FIFOs. The PowerPC 440 system uses the DMA controllers embedded in the processor IP block. The MicroBlaze system uses a custom DMA controller since no embedded DMA controller exists in the current MicroBlaze implementation.

The BSN requires a connection to external memory to access blocks stored off-chip. In this Redsharc implementation, we use the Xilinx Multi-Port Memory Controller (MPMC) to connect the BSN directly to external memory. While the current HDL will only interface with the MPMC, adapting the BSN to use a different memory controller would only require changes to one component of the BSN.

6. Results

A Redsharc system with between-one and eight BLAST kernels has been implemented to measure the performance,

scalability, and resource utilization of the Redsharc abstract API and infrastructure. The characteristics, measured on a Virtex 5 FX 130T FPGA, are presented in this section. Xilinx ISE Design Suite version 11.4 was used for all experiments for FPGA implementation.

6.1. Network Performance. The SSN's performance is measured by the bandwidth between two kernels. While the HWKI abstracts these issues from the hardware developer, these metrics are dependent on (1) width of the stream, (2) width of the SSN, (3) clock frequency of the SSN and kernels. Between two hardware kernels, running at 200 MHz and a 32 bit data width, the SSN's bandwidth is 800 MB/s. When the receiving or transmitting kernel is software, the bandwidth is limited by the throughput of the DMA controller and the message size. Figure 9 shows the measured bandwidth for different stream lengths between hardware kernels and software kernels. The SSN performs best with large message sizes.

The BSN's performance is measured by the latency and bandwidth of a read operation. Similar to the SSN, the Redsharc API abstracts synthesis-time configurations that may affect the bandwidth of a specific system. The settings that affect BSN's bandwidth are (1) location and width of the data, (2) operating frequency of the hardware kernel and BSN (clock domain crossing adds overhead), and (3) possible contention at the remote block. Table 3 provides an overview of the BSN performance for the three types of data locality, given both the BSN and hardware kernels on the same 100 MHz clock. The BSN performs very favorable compared to the PLB which, in previous work, has a measured peek performance of 25 MB/s [18] for a 32 bit read transaction and a 200 MB/s for 1024 bit transaction.

6.2. Kernel Performance. In order to demonstrate Redsharc's scalability, a BLAST bioinformatics kernel was implemented. BLAST is a bioinformatics application that performs DNA comparisons. BLAST compares one short DNA sequence, called the query, against a long database of DNA sequences, and produces a stream of indexes where the two sequences match. The database of DNA sequences is 35 KBytes. The BLAST query is encoded into the 8 KByte local block. Researchers have implemented BLAST on FPGAs to demonstrate impressive speedups [19]; however, previous work [20, 21] has shown that bandwidth requirements for BLAST limit the scalability such that a common bus interconnect is insufficient for transferring databases to each BLAST core as well as being used for off-chip memory lookups. Specifically, each BLAST core can sustain database input rates of 3.2 GB/s. Common bus implementation, such as the processor local bus (PLB), offers sufficient bandwidth (12.8 GB/s) although there are limitations on the burst size and the number of concurrent transfers. For these reasons and due to the aforementioned previous research investigating conventional bus-based implementations, this work focuses on analyzing the scalability of BLAST cores within the Redsharc system.

The BLAST hardware kernel used in these experiments is a reimplementation of the BLAST algorithm with two input streams: one for the input database and one for the length of each sequence in the database. One output stream is used for the matching results. Three blocks (one local and two off-chip) store query information. Each BLAST kernel added to the system can run one query in parallel. Furthermore, it is possible to use the multicast capabilities of the SSN to broadcast the same database to all of the running BLAST kernels.

Table 3: BSN latency and bandwidth for each block type with 100 Mhz BSN clock.

Block type	32 b data width	1024 b data width
Local	2 cc (200 MB/s)	2 cc (6400 MB/s)
Remote	16 cc (25 MB/s)	47 cc (272 MB/s)
Off-chip	25 cc (16 MB/s)	56 cc (228 MB/s)

For comparison, a constant work size of eight queries and one database is used to test the system. The system with a single BLAST kernel must sequentially evaluate the queries, while the system with eight kernels can evaluate them in parallel. This test will show if the Redsharc system can scale with increasing hardware kernels. Table 4 shows the execution time results of the system when executing with a PowerPC processor. The performances of the BLAST application when implemented on a microblaze based system are shown in Table 5.

Both systems show no speedup in the time to load the queries into the blocks or read the result data back from the kernels. These operations are entirely sequential and offer no possible speedup. The 100 MHz Microblaze processor takes significantly longer than the 400 MHz PowerPC processor in these sequential steps. A nearly linear speedup is observed in both systems in the time spent comparing the database to the query. Note that with eight queries running in parallel, the BSN must handle the increasing load for the query information stored in off-chip memory blocks while the SSN is also reading the database from off-chip memory. This contention for the single off-chip memory resource prevents totally linear scaling.

6.3. Resource Utilization. This subsection presents the sizes of the BSN, SSN, and HWKI, the three critical hardware components that comprise a Redsharc system. Table 6 shows the resource utilization of a single BLAST kernel and its associated HWKI. The most used resources are the lookup tables. The remote block interface components of the HWKI appears to use an extraordinary amount of resources. This is due to one of the remote blocks having a data width of 448 bits, the natural data width of a single lookup of the query information stored in the off-chip memory block. The HWKI handles expanding the data from the 32 bit memory interface into the 448 bit data width, a job that would normally have to be done by the application internally.

Figure 11 and Table 8 show the number of lookup tables used in the one, two, four, and eight kernel systems used in these tests. The figure shows that even at eight kernels, the Redsharc infrastructure, the BSN and SSN, uses fewer resources than the kernels. It also shows that the SSN is

Table 4: Performance of one to eight BLAST cores running in a PowerPC 440 Redsharc system. 100 MHz SSN, BSN, and BLAST Kernel Clocks.

Cores	Load queries	Speedup	BLAST exec.	Speedup	Read results	Speedup	Total time	Total speedup
1	1643.94 μs	1x	9013.1 μs	1x	49.49 μs	1x	10706.53 μs	1x
2	1641.16 μs	1x	4512.16 μs	2x	41.74 μs	1.19x	6195.06 μs	1.73x
4	1639.78 μs	1x	2265.38 μs	3.98x	38.73 μs	1.28x	3943.90 μs	2.71x
8	1638.65 μs	1x	1134.11 μs	7.95x	36.4 μs	1.36x	2809.16 μs	3.81x

Table 5: Performance of one to eight BLAST cores running in a Microblaze Redsharc system. 100 MHz SSN, BSN, and BLAST Kernel Clocks.

Cores	Load queries	Speedup	BLAST Exec.	Speedup	Read results	Speedup	Total time	Total speedup
1	4309.3 μs	1x	6767.1 μs	1x	83.2 μs	1x	11159.6 μs	1x
2	3959.2 μs	1.1x	3512.1 μs	1.9x	77.4 μs	1.1x	7548.7 μs	1.5x
4	3783.3 μs	1.1x	1820.5 μs	3.7x	75.1 μs	1.1x	5678.9 μs	2.0x
8	3695.3 μs	1.2x	947.6 μs	7.1x	74.4 μs	1.1x	4717.3 μs	2.4x

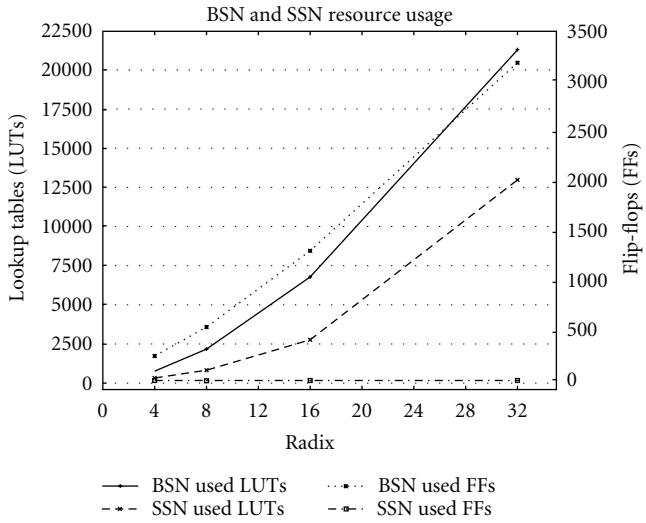

Figure 10: Block switch and stream switch network resource utilization in terms of lookup tables (LUTs) and flip-flops (FFs).

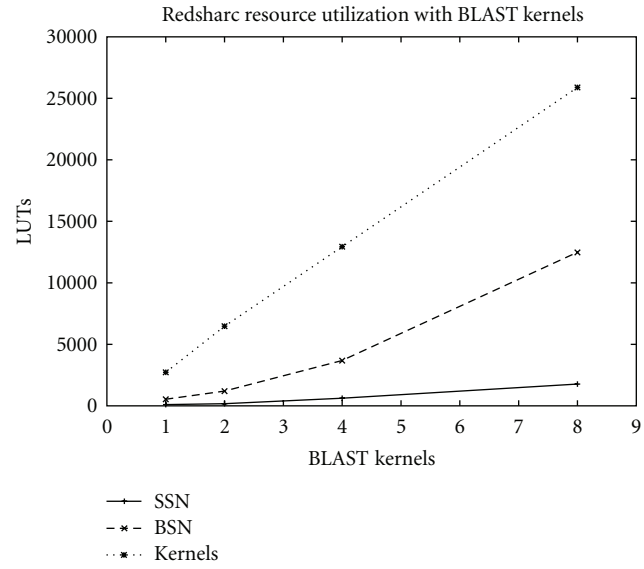

Figure 11: Redsharc system LUT utilization.

Table 6: Resource utilization of a single BLAST Kernel on a Virtex 5 FPGA.

Kernel component	LUT	FF	Carry Chain	BRAM
Local block IF	262	239	118	2
Remote block IF	1708	1077	198	0
Stream IF	1	1	0	3
BLAST application	756	471	105	0
Total	2727	1788	421	5

Table 7: BSN and SSN LUT and FF resource utilization.

Radix	BSN LUTs	BSN FFs	SSN LUTs	SSN FFs
4	456	240	140	0
8	1432	522	576	0
16	5169	1267	2272	0
32	20536	3104	11198	0

scaling lineally with the number of kernels, while the BSN has some exponential growth. Figure 10 and Table 7 illustrate the BSN and SSN's usage of lookup tables and flip-flops for any system. The term radix in this figure refers to the number of ports on each switch. For example, each BLAST kernel has three port connections to the BSN and two to the SSN. Note that the SSN is purely combinatorial and as a result has no flip-flops. The BSN number includes the routing module logic and switch controller, which increases the resource count. Overall, the resources used consume a small portion of available resources for medium-to-large-scale FPGA devices. While a bus presents a smaller resource footprint, as a trade-off the dual switches provide significant bandwidth that is necessary to satisfy the type of high-performance applications targeted by this research.

Table 8: Redsharc system LUT utilization.

Kernels	SSN	BSN	Kernel logic
1	100	534	2727
2	172	1193	6476
4	615	3672	12942
8	1770	12470	25886

The HWKI supports access to variable number of streams and blocks with variable data element sizes. As such, we present the resources required for each additional stream or block and assume 32-bit data widths for all ports. For the SSN, only an LUT is required for each input and output port to drive the Xilinx LocalLink signals and the input and output stream FIFOs. The FIFO depth is configurable by the hardware developer so the number of BRAMs is variable. For the BSN, more logic is needed to support local and remote block requests. Each local block requires 176 flip-flops and 300 LUTs whereas each remote block only requires 161 flip-flops and 163 LUTs. These represent a minimal amount of resources needed to support the high-bandwidth memory transactions while maintaining a common memory interface to the hardware kernel.

7. Conclusion

Programming MCSoPC that span hardware and software is not a trivial task. While abstract programming models have been shown to ease the programmer burden of crossing the hardware/software boundary, their abstraction layer incurs a heavy burden on performance. Redsharc solves this problem by merging an abstract programming model with on-chip networks that directly implement the programming model.

The Redsharc API is based on a streaming programming model but it incorporates random access blocks of memory. Two on-chip networks were implemented to facilitate the stream and block API calls. Our results showed that the SSN and BSN have comparable bandwidth to state-of-the-art technology and scales nearly linearly with parallel hardware kernels. Redsharc can be implemented across multiple platforms, with no dependence on a particular FPGA family or processor interface. Ergo, programmers, and system architects may develop heterogeneous systems that span the hardware/software domain, using a seamless abstract API, without giving up performance of custom interfaces.

Disclaimer

The views, opinions, and/or findings contained in this article/presentation are those of the author/presenter and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

References

[1] A. Jerraya and W. Wolf, "Hardware/software interface codesign for embedded systems," *Computer*, vol. 38, no. 2, pp. 63–69, 2005.

[2] M. Jones, L. Scharf, J. Scott et al., "Implementing an API for distributed adaptive computing systems," in *Proceedings of the 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCMM '99)*, pp. 222–230, April 1999.

[3] R. Laufer, R. R. Taylor, and H. Schmit, "PCI-pipeRench and the swordAPI: a system for stream-based reconfigurable computing," in *Proceedings of the 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCMM '99)*, pp. 200–208, April 1999.

[4] E. Lubbers and M. Platzner, "Reconos: an rtos supporting hard-and software threads," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 441–446, August 2007.

[5] D. Andrews, R. Sass, E. Anderson et al., "Achieving programming model abstractions for reconfigurable computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 34–44, 2008.

[6] A. Patel, C. A. Madill, M. Saldaña, C. Comis, R. Pomès, and P. Chow, "A scalable FPGA-based multiprocessor," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '06)*, pp. 111–120, April 2006.

[7] P. Mahr, C. Lörchner, H. Ishebabi, and C. Bobda, "SoC-MPI: a flexible message passing library for multiprocessor systems-on-chips," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 187–192, 2008.

[8] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented fpga computing in the streams-c high level language," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '00)*, pp. 49–56, April 2000.

[9] D. Unnikrishnan, J. Zhao, and R. Tessier, "Application-specific customization and scalability of soft multiprocessors," in *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM '09)*, pp. 123–130, April 2009.

[10] J. Liang, A. Laffely, S. Srinivasan, and R. Tessier, "An architecture and compiler for scalable on-chip communication," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 7, pp. 711–726, 2004.

[11] L. Shannon and P. Chow, "Simplifying the integration of processing elements in computing systems using a programmable controller," in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '05)*, vol. 2005, pp. 63–72, 2005.

[12] P. Mattison and W. Thies, "Streaming virtual machine specification, version 1.2," Tech. Rep., January 2007.

[13] M. B. Taylor, J. Kim, J. Miller et al., "The raw microprocessor: a computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, 2002.

[14] K. Sankaralingam, R. Nagarajan, H. Liu et al., "Exploiting ILP, TLP, and DLP with the polymorphous trips architecture," *IEEE Micro*, vol. 23, no. 6, pp. 46–10, 2003.

[15] R. Rettberg, W. Crowther, P. Carvey, and R. Tomlinson, "The monarch parallel processor hardware design," *Computer*, vol. 23, no. 4, pp. 18–28, 1990.

[16] *128-Bit Processor Local Bus Architecture Specifications*, IBM, Version 4.7 edition.

[17] Xilinx, http://www.xilinx.com/products/design resources/conn central/locallink member/sp006.pdf.

[18] A. G. Schmidt and R. Sass, "Characterizing effective memory bandwidth of designs with concurrent high-performance computing cores," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 601–604, August 2007.

[19] S. Datta, P. Beeraka, and R. Sass, "RC-BLASTn: implementation and evaluation of the BLASTn Scan function," in *IEEE Symposium on Field Programmable Custom Computing Machines (FCCM '09)*, pp. 88–95, April 2009.

[20] S. Datta and R. Sass, "Scalability studies of the BLASTn scan and ungapped extension functions," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '09)*, pp. 131–136, December 2009.

[21] A. G. Schmidt, S. Datta, A. A. Mendon, and R. Sass, "Investigation into scaling i/o bound streaming applications productively with an all-FPGA cluster," *International Journal on Parallel Computing*. In press.