# PCI-PipeRench and the SWORDAPI: A System for Stream-based Reconfigurable Computing

Ronald Laufer, R. Reed Taylor, Herman Schmit
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA, 15213 USA
{rel, rt2i}@andrew.cmu.edu, herman@ece.cmu.edu

**ABSTRACT: Reconfigurable hardware accelerators have been shown to be flexible and efficient in stream-based applications. In this paper, we discuss the design of PCI-PipeRench and the SWORDAPI. PCI-PipeRench is a coprocessor utilizing the PipeRench architecture which includes on-chip control and data buffering to interface with a host processor over a PCI bus. SWORDAPI calls resemble standard C file control functions, and allow developers to create applications independent of underlying reconfigurable hardware details. In addition, the SWORDAPI provides a cosimulation environment so that verification can be performed using unmodified application source with a hardware simulator. Efficient utilization of the bus is of critical importance in the design of such a system; various methods used to address this issue are presented.**

## 1 Introduction

An increasing number of people are using their computers to handle high-bandwidth streaming media such as live video, real-time encryption, and digital signal processing. The computational load associated with streaming data is different from the load presented by more “traditional” computations: it tends to be bandwidth-intensive, to be comprised of repetitive operations, and to lack traditional data locality.

Many reconfigurable hardware devices have been proposed and designed in response to these nontraditional needs, but in order for reconfigurable hardware to be applied to these problems, it must be interfaced with a host processor. This paper discusses the use of the PipeRench reconfigurable fabric[9] as a coprocessor attached to a traditional processor via a PCI bus. The paper also introduces an API that can be used to interface both with the PipeRench hardware and with simulators created to test the design.

In a well-balanced system, a coprocessor should make good use of its bus by being able to meet or exceed the bandwidth that the bus provides. The throughput of a PipeRench coprocessor far exceeds the bandwidth of a 33 MHz PCI bus. As a result, the performance of the system is determined almost entirely by the degree to which the bus is used efficiently. The design described in this paper is motivated in large part by this need for efficiency. One important part of PCI bus performance is obtained through the use of a packet interface. The packets transferred can carry either configuration or data, allowing the system to use the same bus for both control and data.

The coprocessor and API rely upon the transfer of these configuration and data packets. This system is well suited to stream-based computing and is similar to well understood network architectures. Concepts from the networking community allow us to extend the principle of layered architectural abstraction, exemplified by hardware virtualization in PipeRench. The coprocessor and API allow the use of the same application C code and the same PipeRench configuration bitstream with any present or future PipeRench device without recompilation. The result is the SWORDAPI (Streaming-Word Oriented Reconfigurable Device API), and the PCI-PipeRench coprocessor.

Section 2 of this paper gives a brief review of the PipeRench architecture and the design considerations inherent in interfacing it with the PCI bus. Section 3 details PCI-PipeRench, which was designed for use with the SWORDAPI. Section 4 explains the packet format which the SWORDAPI uses to establish communications between reconfigurable devices and the host processor. Section 5 outlines the function calls that the SWORDAPI provides to application developers and discusses how the API’s modular nature facilitates expansion to multiple simulators and devices. Section 6 shows how the SWORDAPI resembles a layered network protocol stack. Section 7 discusses related systems, and Section 8 presents our conclusions.

## 2 PipeRench Fabric and the PCI Bus

PipeRench is an instance of the class of Pipeline Reconfigurable Fabrics discussed fully in [11]; a brief overview is given here. Pipeline Reconfigurable Fabrics are FPGA-like devices, divided into reconfigurable pipeline stages. Each stage consists of programmable combinational functionality, flexible routing interconnect, and a collection of registers. These stages are separately configurable, and identical to one another. As a result, an application which has been broken up into pipeline stages can be mapped to the hardware stages, and the configuration of any stage of the application can be loaded into any stage of the hardware. These stages are known as *stripes*; the stages of the application are called *virtual stripes,* and the hardware stages they are loaded into are called *physical stripes.*

At runtime, virtual stripes are loaded in sequence, one stage ahead of the incoming data, until the physical stripes are full. Then, the virtual stripe that was resident longest in the fabric is swapped out, and replaced with the next virtual stripe. The physical fabric "scrolls down" the virtual pipeline repeatedly until all the data has traversed all of the pipeline. The application can exceed the physical resources of the device, and still run, with reduced throughput.[5] This allows for forward compatibility: future devices with more physical stripes can run the identical configuration with increased performance, and no recompilation. This method also makes compilation of applications faster, as the place and route algorithm does not have to be optimal, since there is no hard constraint on the size of the pipeline.

PipeRench further divides each stripe into N Processing Elements (PEs). Each PE consists of a barrel shifter, a 3 input LUT, replicated to be B bits wide, a number of B bit registers, and carry and zero detect logic. A B-bit word crossbar is provided in each stripe to connect the inputs of each PE to any of the registered outputs of the previous stripe's PEs or any registered or unregistered output of the current stripe's PEs, The utilization the compiler obtains, and hence the performance of the application, is dependent on the selection of N and B. Figure 1 shows performance results obtained in [8].

The next prototype of PipeRench, PCI-PipeRench, will contain 16 physical stripes, with 16 8-bit PEs per stripe. This prototype is designed to interface to a host processor via a 33MHz, 32-bit PCI Bus. Although PCI is the current standard for desktop PC buses, and thus an attractive target for reconfigurable devices, the I/O needs of PipeRench do not easily match the performance of PCI.

There are three major qualities of PipeRench that make it difficult to interface to PCI:

1. **Higher Frequency:** The operating frequency targeted by PCI-PipeRench is 100MHz, approximately three times the speed of PCI. This will make PCI a bottleneck for many of the operations that map well to PipeRench. Also, bus performance can be highly variable due to traffic from other devices on the bus.
2. **Wide Datapath:** Another drawback to PCI is its narrow width. A key feature of PipeRench is its ability to operate on wide data words. This wide datapath allows PipeRench applications to exploit parallelism.
3. **Bursty Data:** PipeRench's virtualized configuration compensates somewhat for the speed bottleneck, but also complicates the interface. The I/O needs of the fabric vary depending on what parts of the application are currently swapped in. This results in a bursty I/O demand that will not coincide with the semi-constant stream of PCI.

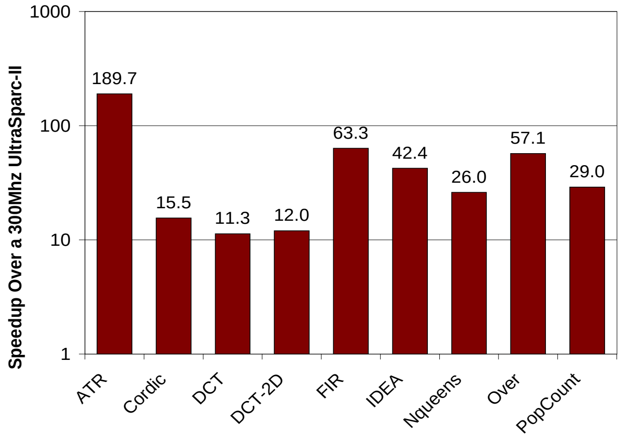

Fig. 1. Projected speedup 8-bit PE, 8 Register per PE, 144-bit wide 28-stripe 100MHz PipeRench vs. UltraSparc-II running at 300Mhz

These concerns are addressed in PCI-PipeRench. In addition, PCI-PipeRench was designed with a multi-chip system in mind, allowing virtual pipelines to expand across multiple chips to increase performance if an appropriately narrow point in the dataflow graph can be found at which to split the pipeline. The design of PCI-PipeRench addresses the mismatch between PCI and PipeRench, and makes PipeRench easily integrated with current products. While it has become clear that PCI is still a bottleneck, pipelined reconfigurable fabrics will become much more efficient when the fabric is located closer to the host processor in future systems.

## 3 PCI-PipeRench

PCI-PipeRench is intended to be integrated into a PC system via the PCI bus. PCI is a standard 32-bit, 33MHz bus interface, capable of direct memory access (DMA). The chip will be used on a custom card containing an off the shelf PCI interface chip, and two 32-bit FIFOs. Figure 2 shows the structure of the PCI-PipeRench chip. The configuration controller is based on the one described in [5]. The configuration controller handles configuration of the stripes and storing and restoring state from swapped out stripes. This section will explain the elements used to compensate for the discrepancies between PipeRench and PCI described in Section 2, and explain other design considerations of PCI-PipeRench.

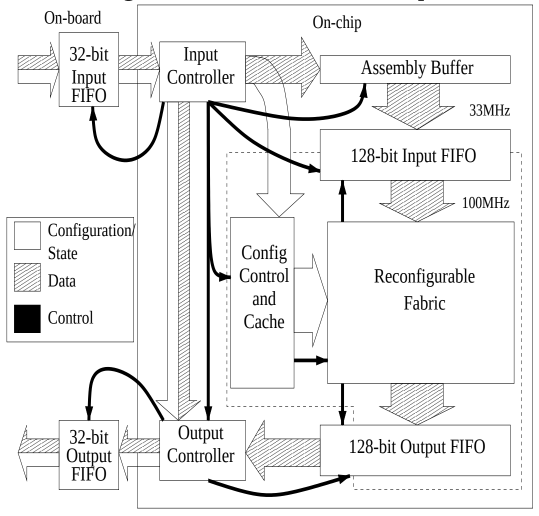

Fig. 2. PCI-PipeRench Structure

### 3.1 Dealing with Higher Frequency

The differing clock frequencies are handled by FIFOs with input and output ports that can be clocked at different frequencies. This allows PCI-PipeRench to take advantage of its own higher frequency without disturbing the PCI Bus. The input port of the 128-bit Input FIFO and the output port of the 128-bit Output FIFO, along with the input and output controllers and assembly buffer are in the clock domain of the PCI Bus and the remainder of the chip is in the 100 MHz PipeRench clock domain. Since the PCI bus must use large DMA transfers to obtain its maximum throughput, deep 32-bit off-chip FIFOs are used to buffer large transfers.

### 3.2 Dealing with the Wide Datapath

In order to support the wide datapath of PipeRench, PCI-PipeRench includes an assembly buffer to concatenate between one and four PCI words into one PipeRench input word. The flexibility of the assembly buffer allows applications with inputs narrower than PipeRench's input bus to utilize the full bandwidth of PCI, without having to pad the inputs to 128 bits. This buffer is controlled by the input controller and can be configured with a flexible control word to assemble in various patterns. The pattern is specified in the I/O controller configuration word (see Section 3.4) by three fields:

1. Initial mask (4 bits)
2. Shift size (2 bits)
3. Initial shift count(2 bits)

The 128-bit assembly buffer is divided into four 32-bit slots. Each cycle, the incoming 32-bit word is written into the slot corresponding to 1's in the 4-bit mask variable. If the 2-bit shift count variable is greater than zero, the mask is shifted left by the shift size, and the shift count is decremented. This process is repeated until the shift count reaches zero signaling that the assembled word is complete. It is put into the 128-bit input FIFO and the mask and shift count are reset to their initial values.

On the other end of the fabric, the output controller formats data in the output FIFO into an output packet, in a similar fashion to the assembly buffer; the only difference is that the assembly buffer has the capability of a one-to- many mapping to the buffer slots[1], and the output buffer obviously cannot do a many-to-one mapping from the 128-bit Output FIFO to the 32-bit Output FIFO.

### 3.3 Dealing with the Bursty Data

When an input stripe is swapped in, the input FIFO is rapidly drained, and when an output stripe is swapped in, the output FIFO is rapidly filled. As long as the bus is not busy, the input FIFO is filled and the output FIFO drained at the rate and availability of PCI. The larger these FIFO buffers are, the more decoupled the I/O demands can be from what PCI provides.

Each time the input and output virtual stripes are swapped in, they only remain resident for a number of cycles equal to the number of physical stripes in the fabric. In between those times, the rest of the pipeline is processing, and the I/O stripes are swapped out, allowing the bus to load inputs and drain outputs for a number of cycles equal to the virtual stripes minus the physical stripes. Thus, the wide FIFOs need only be as deep as the number of physical stripes. PCI-PipeRench has 16 physical stripes, so total capacity of all the on-chip FIFOs is only 256 bytes (small enough to have a negligible impact on die area).

### 3.4 Other Design Considerations

In order to process PCI-PipeRench packets (see Section 4) the chip also includes input and output controllers. The input controller decodes incoming packets and provides central control to the rest of the chip. The output controller constructs outgoing packets as shown in Section 3.2.

The operation of these controllers is configured with one configuration word associated with each application in the configuration cache. Since each line of the configuration cache is the width of one stripe's configuration, there is ample space in this configuration word to program the controller and for future expansion of on-chip resources, such as a scratchpad RAM or more sophisticated I/O control. The advantage of this approach is that the entire configuration for an application, including I/O controller configuration, can be sent in a single DMA bus transfer.

## 4 PCI-PipeRench packets

PCI-PipeRench has a packet based interface that serves the following purposes:

- Bundling data and configuration into large blocks that can take advantage of DMA
- Inter-chip routing to a specific PipeRench chip in a system
- Intra-chip routing to the appropriate part of the chip (fabric, configuration, initial state registers)
- Handling streams of dynamic length

While most PCI interface chips provide mechanisms for sending individual signals between the board and the host processor, these signals ultimately must be decoded on the board from the PCI datastream and are not standardized. To take advantage of the full bandwidth of the bus, and maintain portability to other platforms or chipsets, no signals external to the 32-bit datastream are used in the interface. Data sent across the bus is divided into packets. The preamble of a packet includes its length and instructions to the on-chip data controllers, removing the need for external signals.

Each packet's preamble consists of a 32-bit "header" and a

1. This is accomplished by using an input mask with more than one bit set to 1.

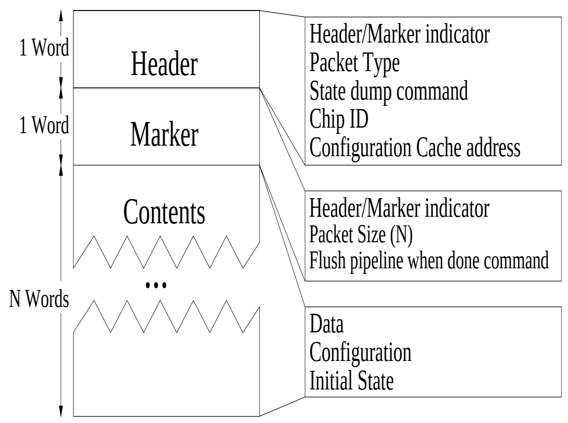

Fig. 3. Structure of a Typical Packet

32-bit “marker”. In general, the header contains instructions on what to do with the enclosed data, and the marker contains a description of the data. These words are considered separate to facilitate multichip systems, where several chips must perform different operations on the same data, and multipacket streams, where the same operation must be performed on multiple packets of data.

The contents of a typical packet are shown in Figure 3. The fields of the preamble are as follows:

Headers and Markers:

- **Header/Marker Indicator:** tells the input controller whether this word is a Header (1) or a Marker (0)

Headers

- **Packet Type:** Packets can be of types: Data to be processed (0), Configuration (1), Initial State (2)or State Dump (3).
- **Chip ID:** For multiple chip systems, identification of the chip that this header is meant to control.
- **Configuration Cache address:** For Configuration and State packets**,** the address at which to write the incoming data. For Data packets, the address of the configuration to be run on the incoming data. For State Dump packets, the address at which to begin the state dump.

Markers:

- **Packet Size:** Number of content words to follow or how many state words to dump.
- **Flush command:** Bit to indicate whether(0) or not(1) to automatically drain the pipeline after this packet.

This packet interface can support a number of reconfigurable chips chained together. In this case, the output of each chip is connected directly to the input of the next one, and the ends of the chain are connected to the bus interface. *Bare packets* with only a marker and no header are possible, as a device can move them around without having an instruction to deal with them. Bare packets are used to send data to a chip further down the chain. The destination chip downstream is sent a header telling it to expect data. The Chip ID field tells each chip whether to keep the header or pass it along. Then, any number or bare packets are sent through the chain. When the input controller receives a bare packet, it checks to see if it is currently processing a header. If so, it is assumed that this incoming packet belongs to the header it has received. If not, this packet must be for a device further on the chain, and is passed on untouched. Eventually it will reach the destination chip, and since that chip will have received a header, it knows the packet is meant for it. In this way, if only some of the chips are being used, the others will act as a mostly transparent bus. A small latency cost will be incurred as the stream is passed through the unused chips.

Data packets are consumed, and produce a corresponding output packet. The output packet will be a bare packet, with the appropriate length being calculated by the device producing it. In this way, the output of one chip can be passed to the next for more computation. The ratio between the size of the input packet and the size of the output packet is statically determined when the configuration is generated. Unfortunately, the ratio’s static nature makes it difficult to implement variable length compression algorithms; dynamic generation of output packet markers is a possible future improvement on the system.

The “flush” bit allows the interface to handle streams of data where the length of the incoming data is not known at the beginning. For an application like a filter, the user may want to continuously stream data through the filter as it comes in from a sensor or similar source, without having to initially specify how long the entire stream will be. Using this signal, the stream can be broken into multiple packets, and the state of the pipeline is retained between the packets.

Multiple bare packets are sent following a single header, and as long as the last packet bit is inactive, PipeRench will wait for more data. When the stream is complete, the last packet bit is made active, either on the last packet of actual data, or on a zero-length packet afterwards, and the input controller automatically drains the pipeline and discards the current header.

Figure 4 shows an example of multiple chips and the flush feature. At the time this packet is sent, it is assumed that all the Reconfigurable Devices in the system have been sent configuration packets containing the applications that they are about to run. If the on-chip configuration caches are large enough, it is possible that all four chips could contain all four programs. In any case, a header is sent to each chip, telling it which program to begin executing (i.e. which configuration to make active). Each chip will pass along any headers with a Chip ID greater than zero, decrementing the Chip ID. Headers with a Chip ID of zero are processed and consumed. This gives the programmer the advantage of addressing the chips by their order of connection to each other, and not requiring jumpers, switches, or registers to hard-code an address.

The first packet of input data is sent after the headers. Chip0 will begin to process the data, and output results to Chip1, and so on down the chain. When Chip0 has read the

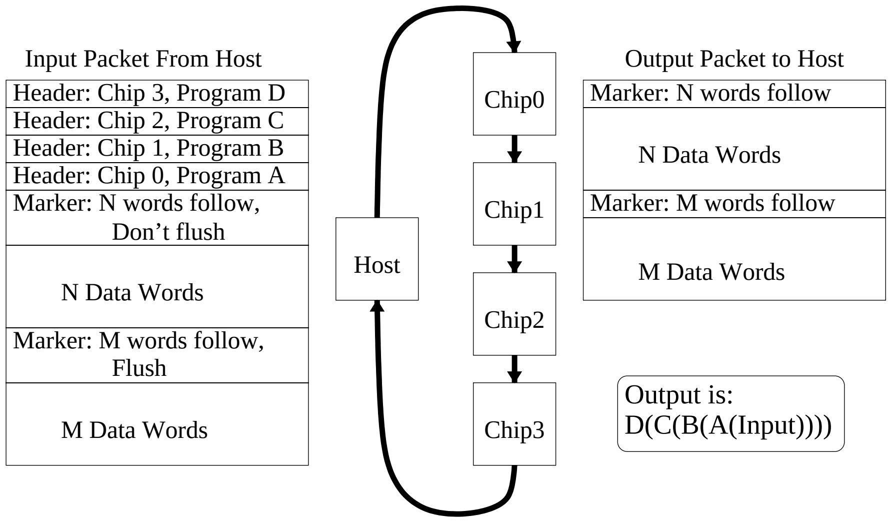
Input Packet From Host
Header: Chip 3, Program D
Header: Chip 2, Program C
Header: Chip 1, Program B
Header: Chip 0, Program A
Marker: N words follow, Don't flush
N Data Words
Marker: M words follow, Flush
M Data Words
Host
Chip0
Chip1
Chip2
Chip3
Output Packet to Host
Marker: N words follow
N Data Words
Marker: M words follow
M Data Words
Output is: D(C(B(A(Input))))

Fig. 4. Packets for multiple PipeRench Chips

last input word of the first packet (the Nth data word) into the pipeline, it stalls, since the Marker indicated that more data was coming. Likewise, the subsequent chips will stall when they run out of data. Chip0 will not have completed its output packet (since there is still data in the pipeline) and since that serves as Chip1's input packet, Chip1 will simply wait until Chip0's output packet finishes. When the next bare packet of input arrives, processing will continue, and Chip0 will drain its pipeline after the (N+M)th data word, as ordered by the second Marker. All the final output data will then propagate back to the host. This system further extends the concept of hardware virtualization, with multiple PipeRenches taking the place of a single large one.

Due to PCI bandwidth issues, running an application which exceeds the size of the configuration cache incurs such a large performance loss that it is impractical. However, we have found that most kernels that have been implemented on PipeRench either fit in the cache, or are easily partitioned on multiple chips. For example, the 8-round IDEA encryption algorithm can be split easily between any of the rounds. We expect future PipeRench based coprocessors to be closer to the host processor, (e.g. at the L2 cache level) where the configuration cache can act more like a true cache, with the full configuration stored in the host's memory.

Other reconfigurable architectures could be adapted to use the PCI-Piperench interface with the addition of compliant I/O controllers to decode and direct packets appropriately. Addressable configuration spaces are supported, but not a required feature, as long the API is aware of what type of device is being used, and does not allow the user to accidentally overwrite the configuration.

## 5 SWORDAPI: Streaming Word-Oriented Reconfigurable Device API

The SWORDAPI was designed with several high-level objectives in mind. In particular, the API had to maximize the efficiency of PCI bus utilization. In addition, we required that the API be generalized and extensible to many different backend simulation tools, and that the API present an interface that would be familiar to programmers who were unaccustomed to reconfigurable computing.

### 5.1 A Generalized & Extensible API

The internals of the SWORDAPI are layered, that is, they have the ability both to interface directly with our hardware prototype and to run applications on many different software-based simulators (including Verilog, Java, and C++). Applications are able to interface with any of these packages using only the API, with no modification to the application source code.

Additionally, it is simple to add interfaces to new types of simulators. There is little need to duplicate code between versions, which simplifies the process of developing new versions of the API for new simulators or prototypes.

We accomplished this by taking advantage of the object-oriented nature of C++. (See Figure 5.) Each API call was written as a virtual public member of a C++ base class, with a different derived class for each type of simulator.[1]

Using a structure of this type, the API for each different simulator can inherit the majority of its functions from the base class. Only the functions which deal directly with the communications mechanisms need to be changed.

As shown in Figure 5, our base class is an abstract class. All the code which does not deal directly with the communications interface was written to be generally applicable to all versions of the API. The interface between the API and Verilog is implemented in the Verilog_PipeRench derived class using named pipes. Verilog PLI calls to access the named

1. A virtual member function of a C++ class is inherited by its derived subclasses, but may be overridden by each of those subclasses.

pipes were created using a similar method to the socket-based PLI calls in [6]. To implement a derived class which would interface directly to the hardware, we will rewrite the communication calls to send and receive data over the PCI bus. No other modifications should be necessary. In addition, we have implemented an instance of the API that uses a second Verilog simulator package; this required less than 20 lines each of new API code and PLI code.

### 5.2 Performance Requirements

It is shown in [8] that for the majority of applications mapped to PCI-PipeRench, overall performance is limited by the PCI bus. Therefore, it is vital that the API maximize the utilization of the PCI bus, and that it not contribute to the degradation of performance.

To this end, the API establishes buffers between the application and the bus which will allow the bus to operate in DMA mode as much as possible. In addition, the API holds input data until a sufficiently large amount has been accumulated to make efficient use of the DMA bursts.

### 5.3 A Familiar Interface

Another feature of the SWORDAPI is that it has an interface that is familiar to application programmers who are not necessarily accustomed to reconfigurable computing. We modeled most of the API calls after the C library calls used for interfacing with files.

The code sample in Figure 6, which performs a vector addition, illustrates the operation of a few SWORDAPI calls.

```
  // Set up input & output tables
  // (defined elsewhere)
piperench->set_in_table(in_table);
piperench->set_out_table(out_table);

  // configure PipeRench
start = piperench->config(“add.bin”);

  // Send initial state and start addr.
piperench->init(start, 5, statestripes,
               states);

  // Send 8 of each input
piperench->pwrite(a, sizeof(int), 8, 0);
piperench->pwrite(b, sizeof(int), 8, 1);

  // tell PipeRench we’re done
piperench->pdone();

  // attempt to read 15 results
i = piperench->pread(c, sizeof(int),
                    15, 0);
```

Fig. 6. SWORDAPI code sample

### 5.4 Implementation

The SWORDAPI allows an application to perform three principal tasks: configuring PipeRench, sending data to PipeRench, and receiving data from PipeRench.

#### 5.4.1 Configuration

- `int config(char *filename)`
  Configuration is acheived through a function call which reads configuration data out of a binary file and sends it (along with the appropriate headers) to PipeRench in one large configuration packet.
- `int set_in_table(char *in_table)`
  `int set_out_table(char *out_table)`
  These tables allow the user to assign names to inputs and outputs.

-
  ```
  int init(int config_num,
           int num_stripes,
           int *vstripes,
           int *statevals)
  ```
  The `init()` function instructs PipeRench to begin executing with configurations beginning at `config_num`. It also permits the application to send initial state values into the stripes on PipeRench.

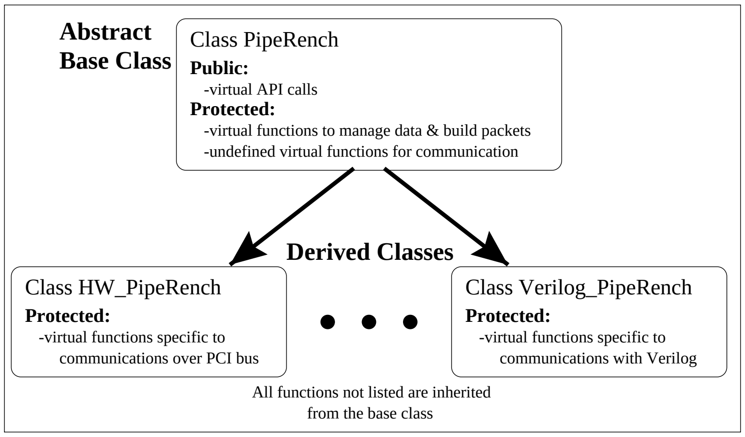
Abstract Base Class
Class PipeRench
Public:
-virtual API calls
Protected:
-virtual functions to manage data & build packets
-undefined virtual functions for communication
Derived Classes
Class HW_PipeRench
Protected:
-virtual functions specific to communications over PCI bus
Class Verilog_PipeRench
Protected:
-virtual functions specific to communications with Verilog
All functions not listed are inherited from the base class

Fig. 5. The Object-Oriented Structure of the API

5.4.2 Sending Data

- `int pwrite(void *input_ptr, int size,`
  `             int num_elements,`
  `             unsigned char input_num)`

  To send data to PipeRench, the application specifies to the SWORDAPI a pointer to a number of values for a particular logical input specified by `input_num`. The API copies the values out of the application and places them directly into a buffer of data words to be sent in packets.(See Section 5.5 for details on this copy operation.)
- `void psetbufsize(int size)`
  `void pflush()`

  Each data word contains exactly one value for each input to PipeRench. As the SWORDAPI accumulates values for the various inputs to PipeRench, it keeps track of how many data words are complete, containing a value for every input. Once the number of complete data words meets the number set using `psetbufsize()`, the API sends PipeRench a packet containing all the data that is ready to be sent. `pflush()` can be used to force that buffer to flush.

5.4.3 Output

- `void pdone()`

  When the last input word has been queued for output using `pwrite()`, tell PipeRench to flush the virtual pipeline and prepare all outputs for reading.
- `int pread(void *output_ptr, int size,`
  `          int num_elements,`
  `          char output_num)`

  Whenever the application requests output data, the API retrieves all the data words that are waiting in PipeRench's internal queues. It then copies the data into the application.

### 5.5 Performance Issues

As described in Section 5.4.2, when the SWORDAPI builds packets to be sent to PipeRench, the data is stored in unpadded slots within a contiguous block of memory. There are two reasons why this copy is deemed worthwhile: it enables PipeRench to autonomously retrieve large numbers of packets over the PCI bus using DMA, and it protects the data from modification by the application during the DMA transfer.

Nonetheless, this copy presents a particular hazard to the performance of the API. It is critical that the number of times that input data is copied from one region in memory to another be held to a minimum, as the API takes a performance hit every time this data is copied.

There is another concern involved in assembling packets of input data. If an application calls for multiple vector inputs, the inputs will most likely be stored in separate arrays, which the SWORDAPI interleaves into single packet, because PipeRench must read in one of each input at a time. For example, in the simple case of the vector add shown in Figure 6: $\overline{A}+\overline{B}=\overline{C}$, $\overline{A}$ and $\overline{B}$ will be stored in host memory in separate contiguous blocks. But PipeRench will want $A_n$ and $B_n$ at the same time, and so on. Interleaving the input vectors can cause another loss of performance for the API; however, it is necessary in order to support the performance gains associated with DMA.

We have not yet determined exactly how much effort should go into aligning data for DMA transfers before it begins to offset the performance gained from burst mode. The best solution to this problem is to move the task of interleaving onto the chip itself, so that simple contiguous blocks of single inputs are sent by DMA across the bus; however, silicon area constraints prevented us from implementing this feature on this version of PCI-PipeRench. We are also investigating high-performance API calls which will offer less protection, but which will eliminate this copy procedure.

Fortunately, few real PipeRench-friendly applications have multiple input streams. Encryption, transforms, filters, and the like operate on a single stream, and aren't affected by this problem. But a future expansion to the SWORDAPI and PCI-PipeRench may include separate FIFOs for each logical input, perhaps sent through separate DMA channels.

## 6 Analogy to the Layered Network Model

Since the SWORDAPI uses a packet-based communications system, it is natural to draw comparisons between it and computer networking protocols. When framed in this context, some useful and interesting aspects of the API's design come to light.

Network protocols are often specified as a set of layers, each having a well-defined interface. The most famous layered protocol stack is the seven-layered Open Systems Interconnection (OSI) stack specified by the ISO. We can define the SWORDAPI as a stack of "protocol layers" as well. A layer diagram is shown in Figure 7.

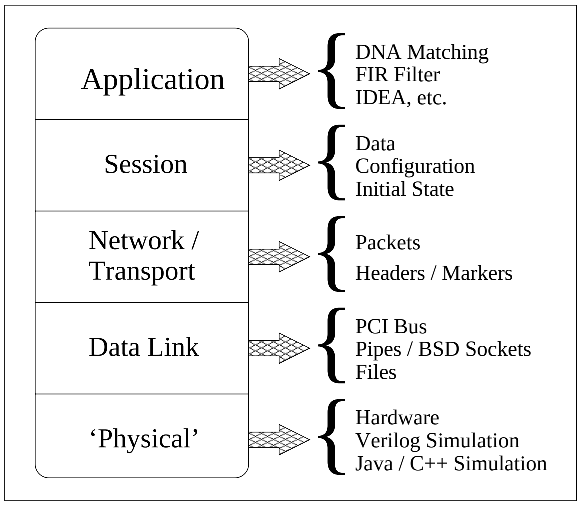

Fig. 7. API Network Layers

Viewing the SWORDAPI in this way, we can see clearly how its object-oriented nature provides levels of abstraction which can be very useful both to the application programmer and to the developer of the API.

- The Session layer represents the programmer's interface. It allows any application to utilize the hardware without any awareness of the API internals, or of the hardware implementation details.
- The Network / Transport layer contains all the code necessary to do the buffering described in Section 5.4.2. This allows the developer of the API to leave this code unchanged when interfacing to the hardware and to various simulators. It also allows the API to route packets to specific PCI-PipeRench chips in a multichip system, and to direct packets to the appropriate controller on-chip.
- The Data Link layer allows the API developer to interface all of the above layers with different hardware prototypes and with different software simulations. At this layer, the only difference between models of the reconfigurable device is the way in which they communicate with the application.

This modularity allows a programmer to adapt the API with a minimum of coding. It has been simple to benchmark different applications with different hardware simulators using different API internals, all without rewriting any more code than necessary.

Coupled with the forward-compatible nature of PipeRench, this standardized API structure permits the exchange of PipeRench chips for more advanced ones (containing more physical stripes) with absolutely no modification or recompilation of code necessary - even the configuration bits would remain the same.

APIs for systems other than PipeRench, if designed with this model in mind, would benefit in the same ways. If an agreed-upon interface were used, it would even be possible to use application code written for one pipelined reconfigurable coprocessor with the API and hardware for another. This could all be accomplished without the need to write additional code to compensate for differences between the fabrics; these compensations would be handled automatically by the layered nature of the API.

## 7 Related Work

The Wormhole Runtime Reconfiguration system from Virginia Polytechnic Institute and State University shares many similar features to the SWORDAPI. In both systems, configuration and data streams are both sent through the same wide port. Header information directs the flow of both configuration and data streams through the fabric. However, in Wormhole RTR, the reconfigurable fabric itself must be able to sort out configuration at each routing point. This does allow multiple streams to be processed simultaneously, but increases the complexity required within the fabric. The SWORDAPI only requires compliant data and configuration controllers and is therefore adaptable to many fabric architectures. In addition, the SWORDAPI allows for cached configurations, a major feature of PCI-PipeRench which enables context switching between applications without re-sending the configuration data.[3]

The Cheops/Magic8 system is another example of a network of stream-based processors.[4] While PCI-PipeRench is attached to a PC or similar host via the PCI bus, the Cheops system uses a specialized backplane and busses, which gives it flexibility and large bandwidth. Also, the Cheops system is specialized for image and video processing. But, the treatment of data as a stream and the modular processor capabilities show the emergence of similar solutions for similar problems.

The USC Information Sciences Institute's SLAAC architecture also uses a layered software interface to connect heterogeneous adaptive devices to a host processor. SLAAC's software layers are more like a traditional network interface, with the goal of incorporating multiple disparate devices in a single system. The lower layers of the SLAAC protocol are device dependent, whereas the SWORDAPI puts the device dependent operations in PCI-PipeRench's on-chip data controllers.[7]

Like the SWORDAPI, JHDL, from Brigham Young University, stresses the need for reconfigurable coprocessors to be treated in a familiar way by applications programmers. It also exploits object oriented programming to allow simulators and real hardware to be used interchangeably in cosimulation.[2]

Commercial reconfigurable logic boards, such as Annapolis Micro Systems' WILD-ONE$^{TM}$ PCI Board, have proprietary APIs. These APIs are intended to be used with traditional FPGAs and are not well-suited for runtime reconfiguration and unconventional reconfigurable fabrics like PipeRench. The WILD-ONE$^{TM}$ API also includes hardware specific calls such as setting clock frequencies and FIFO thresholds.[1] These are issues that SWORDAPI hides from the application programmer.

## 8 Conclusions

In this paper we have introduced both hardware and software interfaces for reconfigurable devices which free programmers from the need to understand low-level details of the hardware. The hardware interface embodied in PCI-PipeRench is particularly suited for the high-bandwidth streaming applications (which are increasingly important tasks). The SWORDAPI uses a layered approach to implement this interface for multiple back-ends, while minimizing the amount work for the developer. It supports PipeRench's inherent forward compatibility by maintaining the capacity to execute configurations on future chips without recompilation

PCI-Piperench consists of a PipeRench reconfigurable fabric with added control units, and on-chip FIFOs to compensate for differences between PCI and PipeRench data flows. Making efficient use of the PCI bus was the foremost concern in designing the chip.

The SWORDAPI's layered nature maintains abstractions between the application, the API itself, and the hardware. This makes it easily extensible, and enables it to integrate unmodified application code with present and future PipeRench implementations. In addition, it supports cosimulation with many software models of PipeRench.

At this time, two working models of PCI-PipeRench have been written in Verilog, and SWORDAPI instances for two separate Verilog simulation packages have been completed. Cosimulations of the PipeRench models and applications using the SWORDAPI are running successfully and have been invaluable in verification of the chip. VLSI design of the two PCI-PipeRench chips is in progress; the chips will be fabricated in 0.25 and 0.35 micron processes.

## 9 References

[1] Annapolis Micro Systems, Inc. *WILD-ONE$^{TM}$ Reference Manual*. Revision 3.1. 1998.

[2] P. Bellows and B. Hutchings. JHDL - An HDL for Reconfigurable Systems. In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines, pages* 175-184, Napa Valley, CA, April 1998.

[3] R. Bittner and P. Athanas. Wormhole Run-Time Reconfiguration. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 79-85, Monterey, CA, February 1997.

[4] V. M. Bove, Jr. and J. A. Watlington. Cheops: A Reconfigurable Data-Flow System for Video Processing, *IEEE Transactions on Circuits and Systems for Video Technology*, pages 140-149, April 1995.

[5] S. Cadambi, J. Weener, S.C. Goldstein, H. Schmit, and D. Thomas. Managing Pipeline-Reconfigurable FPGAs. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 55-64, Monterey, CA, February 1998.

[6] S. L. Coumeri and D. E. Thomas. A Simulation Environment for Hardware-Software Codesign. *International Conference on Computer Design*, October 1995.

[7] S. Crago, B. Schott, R. Parker. SLAAC: A Distributed Architecture for Adaptive Computing. In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 286-287, Napa Valley, CA, April 1998.

[8] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, R. Taylor, R. Laufer. PipeRench: A Coprocessor for the Future. In *International Symposium on Computer Architecture*, Atlanta, GA June1999. To be published.

[9] M. Moe, H. Schmit, S.C. Goldstein. Characterization and Parameterization of a Pipeline Reconfigurable FGPA.In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 294-295, Napa Valley, CA, April 1998.

[10] J. Rose and D. Hill. Architectural and Physical Design Challenges for One-Million Gate FPGAs and Beyond. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 129-132, Monterey, CA, February 1997.

[11] H. Schmit. Incremental reconfiguration for pipelined applications. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 47-55, Napa Valley, CA, April 1997.