# Software-Driven Hardware Development

Myron King, Jamey Hicks, John Ankcorn
Quanta Research Cambridge
{myron.king,jamey.hicks,john.ankcorn}@qrclab.com

## ABSTRACT

The cost and complexity of hardware-centric systems can often be reduced by using software to perform tasks which don't appear on the critical path. Alternately, the performance of software can sometimes be improved by using special purpose hardware to implement tasks which *do* appear on the critical path. Whatever the motivation, most modern systems are composed of both hardware and software components.

Given the importance of the connection between hardware and software in these systems, it is surprising how little automated and machine-checkable support there is for co-design space exploration. This paper presents the Connectal framework, which enables the development of hardware accelerators for software applications by generating hardware/software interface implementations from abstract Interface Design Language (IDL) specifications.

Connectal generates stubs to support asynchronous remote method invocation from software to software, hardware to software, software to hardware, and hardware to hardware. For high-bandwidth communication, the Connectal framework provides comprehensive support for shared memory between hardware and software components, removing the repetitive work of processor bus interfacing from project tasks.

This framework is released as open software under an MIT license, making it available for use in any projects.

## Categories and Subject Descriptors

B.4.3 [**INPUT/OUTPUT AND DATA COMMUNICATIONS**]: Interconnections (subsystems)—*Interfaces; Asynchronous/synchronous operation*

## Keywords

Connectal; Design Exploration; Software


*FPGA'15,* February 22–24, 2015, Monterey, California, USA.

ACM 978-1-4503-3315-3/15/02 ...$15.00.
10.1145/2684746.2689064.

## 1. INTRODUCTION

Because they are so small and inexpensive, processors are now included in all but the smallest hardware designs. This grants flexibility to hardware designers because the non-performance-critical components can be implemented in software and the performance-critical components can be implemented in hardware. Using software for parts of the design can decrease the effort required to implement configuration and orchestration logic (for example). It can also offer hardware developers greater adaptability in meeting new project requirements or supporting additional applications.

As a system evolves through design exploration, the boundary between the software and hardware pieces can change substantially. The old paradigm of "separate hardware and software designs before the project starts" is no longer sustainable, and hardware teams are increasingly responsible for delivering significant software components.

Despite this trend, hardware engineers find themselves with surprisingly poor support for the development of the software that is so integral to their project's success. They are often required to manually develop the necessary software and hardware to connect the two environments. In the software world, this is equivalent to manually re-creating header files from the prose description of an interface implemented by a library. Such ad hoc solutions are tedious, fragile, and difficult to maintain. Without a consistent framework and toolchain for jointly managing the components of the hardware/software boundary, designers are prone to make simple errors which can be expensive to debug.

The goal of our work is to support the flexible and consistent partitioning of designs across hardware and software components. We have identified the following four goals as central to this endeavor:

- Connect software and hardware by compiling interface declarations.
- Enable concurrent access to hardware accelerators from software.
- Enable high-bandwidth sharing of system memory with hardware accelerators.
- Provide portability across platforms (CPU, OS, bus types, FPGAs).

In this paper, we present a software-driven hardware development framework called **Connectal**. Connectal consists of a fully-scripted tool-chain and a collection of libraries which can be used to develop production quality applications comprised of software components running on CPUs

communicating with hardware components implemented in FPGA or ASIC.

When designing Connectal, our primary goal was to create a collection of components which are easy to use for simple implementations and which can be configured or tuned for high performance in more complicated applications. To this end, we adopted a decidedly minimalist approach, attempting to provide the smallest viable programming interface which can guarantee consistent access to shared resources in a wide range of software and hardware execution environments. Because our framework targets the implementation of performance-critical systems rather than their simulation, we have worked hard to remove any performance penalty associated with its use.

We wrote the hardware components of the Connectal libraries in Bluespec SystemVerilog[1, 2, 3] (BSV) because it enables a higher level of abstraction than the alternatives and supports parameterized types. The software components are implemented in C/C++. We chose Bluespec interfaces as the interface definition language (IDL) for Connectal's interface compiler.

This paper describes the Connectal framework, and how it can be used to flexibly move between a variety of software environments and communication models when mapping applications to platforms with connected FPGAs and CPUs.

**Paper Organization:** In Section 2, we present the running example in a number of different execution environments. In Section 3, we give an overview of the Connectal framework and its design goals. In Section 4 we discuss the details of Connectal and how it can be used to implement the example. Section 5 describes the implementation of Connectal, supported platforms, and the tool chain used to coordinate the various parts of the framework. The paper concludes with a discussion of performance metrics and related work.

# 2. ACCELERATING STRING SEARCH

The structure of a hardware/software (HW/SW) system can evolve quite dramatically to reflect changing requirements, or during design exploration. In this section, we consider several different implementations of a simple string search application [4]. Each variation represents a step in the iterative refinement process, intended to enhance performance or enable a different software execution environment.

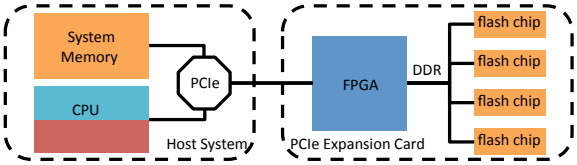


**Figure 1: Target platform for string search application**

Figure 1 shows the target platform for our example. The pertinent components of the host system are the multi-core CPU, system memory, and PCI Express (PCIe) bus. The software components of our application will be run on the CPU in a Linux environment. Connected to the host is a PCIe expansion card containing (among other things) a high-performance FPGA chip and a large array of flash memory. The FPGA board was designed as a platform to accelerate "big data" analytics by moving more processing power closer to the storage device.

## 2.1 Initial Implementation

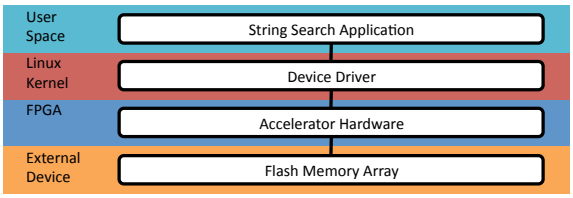


**Figure 2: Logical components of the string search system**

The design process really begins with a pure software implementation of the algorithm, but the first attempt we consider is the initial inclusion of HW acceleration shown in Figure 2. The search functionality is executed by software running in user-space which communicates with the hardware accelerator through a device driver running in the Linux kernel. The hardware accelerator, implemented in the FPGA fabric, executes searches over data stored in the flash array as directed by the software.

The FPGA has direct access to the massive flash memory array, so if we implement the search kernel in hardware, we can avoid bringing data into the CPU cache (an important consideration if we intend to run other programs simultaneously). By exploiting the high parallelism of the execution fabric as well as application aware caching of data, an FPGA implementation can outperform the same search executed on the CPU.

## 2.2 Multithreading the Software

The efficient use of flash memory requires a relatively sophisticated management strategy. Our first refinement is based on the observation that there are four distinct tasks which the application software executes (mostly) independently:

- Send search command to the hardware.
- Receive search results from the hardware.
- Send commands to the hardware to manage the flash arrays
- Receive responses from the flash management hardware

To exploit the task-level parallelism in our application, we can assign one thread to each of the four enumerated tasks. To further improve efficiency, the two threads receiving data from the hardware put themselves to sleep by calling **poll** and are woken up only when a message has been received.

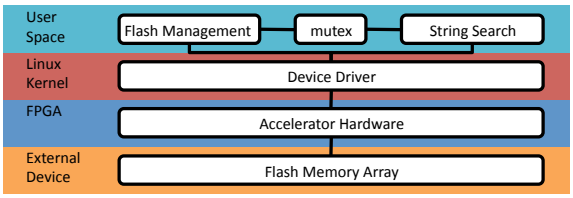


**Figure 3: Using a mutex to coordinate user-level access to hardware accelerator**

With the introduction of multithreading, we will need a synchronization mechanism to enforce coherent access to the hardware resources. Because the tasks which need coordinating are all being executed as user-space threads, the access control must be implemented in software as well. As shown in Figure 3, a mutex is used to coordinate access to the shared hardware resource between user-level processes.

## 2.3 Refining the Interfaces

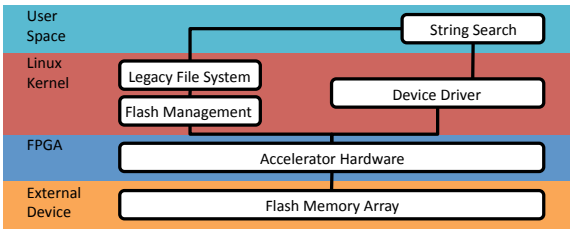


**Figure 4: Movement of functionality from user to kernel space. Software-based coordination between kernel and user processes are prohibitively expensive.**

Figure 4 shows a further refinement to our system in which we have reimplemented the Flash Management functionality as a block-device driver. Instead of directly operating on physical addresses, the string search now takes a file descriptor as input and uses a Linux system-call to retrieve the file block addresses through the file system. This refinement permits other developers to write applications which can take advantage of the accelerator without any knowledge of the internal details of the underlying storage device. It also enables support for different file systems as we now use a POSIX interface to generate physical block lists for the the storage device hardware. The problem with this refinement is that we no longer have an efficient SW mechanism to synchronize the block device driver running in kernel space with the application running in user space.

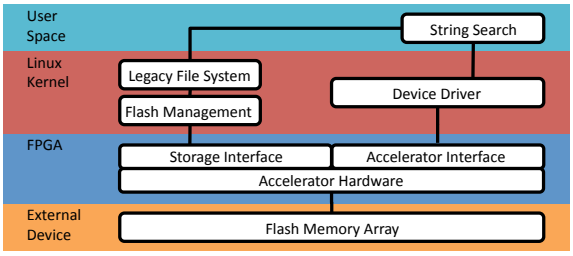


**Figure 5: Correct interface design removes the need for coordination between user and kernel threads.**

To solve to this problem (shown in Figure 5), we can remove the need for explicit SW coordination altogether by giving each thread uncontested access to its own dedicated HW resources mapped into disjoint address regions. (There will of course be implicit synchronization through the file system.)

## 2.4 Shared Access to Host Memory

In the previous implementations, all communication between hardware and software takes place through memory mapped register IO. Suppose that instead of searching for single strings, we want to search for large numbers of (potentially lengthy) strings stored in the flash array. Attempting to transfer these strings to the hardware accelerator using programmed register transfers introduces a performance bottleneck. In our final refinement, the program will allocate memory on the host system, populate it with the search strings, and pass a reference to this memory to the hardware accelerator which can then read the search strings directly from the host memory.

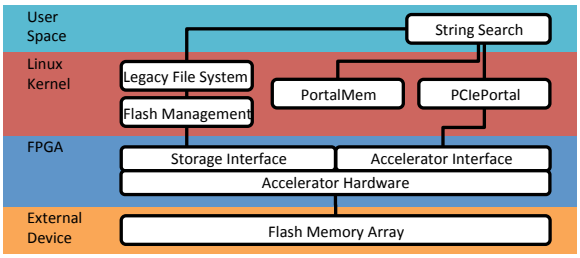


**Figure 6: Connectal support for DMA.**

Efficient high-bandwidth communication in this style requires the ability to share allocated memory regions between hardware and software processes without copying. Normally, a programmer would simply call application space **malloc**, but this does not provide a buffer that can be shared with hardware or other software processes. As shown in Figure 6, a special-purpose memory allocator has been implemented in Linux, using dmabuf[5] to provide reference counted sharing of memory buffers across user processes and hardware.

To conclude, we consider how the HW/SW interface changed to accommodate each step in the refinement process: The hardware interface required by the design in Figure 2 is relatively simple. Command/response queues in the hardware accelerator are exposed using a register interface with accompanying "empty"/"full" signals. To support the use of **poll** by the refinement in Figure 3, interrupt signals must be added to the hardware interface and connected to the Linux kernel. Partitioning the address space as required by the refinement in Figure 5 necessitates a consistent remapping of registers in both hardware and software.

# 3. THE CONNECTAL FRAMEWORK

In and of themselves, none of the HW/SW interfaces considered in Section 2 are particularly complex. On the other hand, implementing the complete set and maintaining correctness as the application evolves is a considerable amount of care, requiring deep understanding of both the application and the platform. The Connectal framework is a collection of tools and library components which was designed to address these challenges with the following features:

- Easy declaration and invocation of remote methods between application components running on the host or in the FPGA.
- Direct user-mode access to hardware accelerators from software.
- High performance read and write bus master access to system memory from the FPGA

- Infrastructure for sharing full speed memory port access between an arbitrary number of clients in the FPGA fabric
- Portability across platforms using different CPUs, buses, operating systems, and FPGAs
- Fully integrated tool-chain support for dependency builds and device configuration.

In this section, we introduce the Connectal framework through a discussion of its prominent features.

## 3.1 Portals

Connectal implements remote method invocation between application components using asynchronous messaging. The message and channel types are application specific, requiring the user to define the HW/SW interface using BSV interfaces as the interface definition language (IDL). These interfaces declare logical groups of unidirectional "send" methods, each of which is implemented as a FIFO channel by the Connectal interface compiler; all channels corresponding to a single BSV interface are grouped together into a single **portal**.

From the interface specification, the Connectal interface compiler generates code for marshalling the arguments of a method into a message to be sent and unmarshaling values from a received message. It generates a *proxy* to be invoked on the sending side and a *wrapper* that invokes the appropriate method on the receiving side. Platform specific libraries are used to connect the proxies and wrappers to the communication fabric.

In the hardware, each portal is assigned a disjoint address range. On the host, Connectal assigns each portal a unique Linux device (/dev/portal⟨n⟩) which is accessed by the application software using the generated wrappers and proxies. An application can partition methods across several portals, to control access to the interfaces by specific hardware or software modules. To support bi-directional communication, at least two portals are required: one which allows software to "invoke" hardware, and another for hardware to "invoke" software. Each portal may be accessed by different threads, processes, or directly from the kernel.

## 3.2 Direct user-mode access to hardware

We designed Connectal to provide direct access to accelerators from user-mode programs in order to eliminate the need for device-drivers specific to each accelerator. We have implemented a kernel module for both X86 and ARM architectures with a minimal set of functionality: the driver implements **mmap** to map hardware registers into user space and **poll** to enable applications to suspend a thread waiting for interrupts originating from the hardware accelerators. These two pieces of functionality have been defined to be completely generic; no modification is required to kernel drivers as the HW/SW interface evolves. All knowledge of the interface register semantics (and corresponding changes) is encoded by the interface compiler in the generated proxies and wrappers which are compiled as part of the application and executed in user-mode.

This approach is known as user-space device drivers [6, 7] and has a number of distinct advantages over traditional kernel modules. To begin with, it reduces the number of components that need to be modified if the HW/SW interface changes, and eliminates the need for device-driver development expertise in many cases. Secondly, after the hardware registers have been mapped into user address space, the need for software to switch between user and kernel mode is all but eliminated since all "driver" functionality is being executed in user-space.

## 3.3 Shared Access to Host Memory

Connectal generates a hardware FIFO corresponding to each method in the portal interface, and the software reads and writes these FIFOs under certain conditions. To improve throughput, Connectal libraries also support credit-based flow-control. Though credit-based flow-control with interrupts is more efficient than polling status registers from software, there is often the need for much higher bandwidth communication between the hardware and software.

Hardware accelerators often communicate with the application through direct access to shared memory. An important feature of Connectal is a flexible, high performance API for allocating and sharing such memory, and support for reading and writing this memory from hardware and software. The Connectal framework implements this through the combination of a Linux kernel driver, C++ libraries, and BSV modules for the FPGA. We implemented a custom kernel memory allocator for Connectal, **portalmem**, using the kernel dmabuf support. Any solution which allocates and shares memory between hardware and software must meet two high-level requirements:

- Allocated buffers must have reference counts to prevent memory leaks.
- Efficient mechanisms must be provided to share the location of allocated regions.

Using the portalmem driver, programs can allocate regions of system memory (DRAM) and map it into their own virtual address space. Reference-counted access to shared memory regions allocated using portalmem can be granted to other SW processes by transmitting the file descriptor for the allocated region. Reference counting has been implemented in the driver so that once an allocated memory region has been dereferenced by all SW and HW processes, it will be deallocated and returned to the kernel free memory pool.

Simple hardware accelerators often require contiguous physical addresses. Unfortunately, when allocating memory from a shared pool in a running system, obtaining large areas of contiguous memory is often problematic, limiting the size of the region that can be allocated. To support indexed access to non-contiguous memory aggregates, Connectal provides address translation support to hardware accelerators in the FPGA, similar to the MMU functionality on the CPU side.

## 3.4 Distributed Access to Memory Ports

When building accelerators for an algorithm, multiple parameters are often accessed directly from system memory using DMA. As the hardware implementation is parallelized, multiple accesses to each parameter may be required. In these cases, the number of memory clients in the application hardware usually exceeds the number of host memory ports. Sharing these ports requires substantial effort, and scaling up a memory interconnect while maximizing throughput and clock speed is extremely challenging.

To support this common design pattern, the Connectal framework provides provides a portable, scalable, high performance library that applications can use to to facilitate the efficient sharing of host memory ports. This library is

implemented as parameterized Bluespec modules which allow the user to easily configure high-performance memory access trees, supporting both reading and writing.

## 3.5 Platform Portability

We structured Connectal to improve the portability of applications across CPU types, operating systems, FPGAs, and how the CPU and FPGA are connected. The software and hardware libraries are largely platform independent. As a result, applications implemented in the framework can be compiled to run on the range of different platforms.

Supported platforms are shown in Figure 7. Application software can be executed on x86 and ARM CPUs running either Ubuntu or Android operating systems. A range of different Xilinx FPGAs can be connected to the CPU and system memory via PCI Express or AXI. The BSV simulator (Bluesim) can be used in place of actual FPGA hardware for debugging purposes.

When the target application needs to interact with other Linux kernel resources (for example, a block device or a network interface), the application may run in kernel mode with the logic run either in an FPGA or in Bluesim.

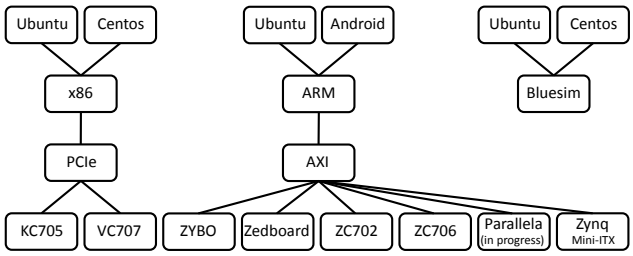


**Figure 7: Platforms supported by Connectal**

# 4. IMPLEMENTING STRING SEARCH

Having covered the features of the Connectal at a high level, we now explain more specifically how the framework can be applied to implement the refinements outlined in Section 2.

## 4.1 Initial Implementation

The FPGA is connected to the host system with a PCIe bus, and to the memory array with wires. In addition to implementing a search kernel, the hardware accelerator must communicate with the software components and with the flash chips. Communication with the software takes place through portals, whose interface declaration is given below:

```
interface StrstrRequest;
  method Action setupNeedle(Bit#(8) needleChars);
  method Action search(Bit#(32) haystackPtr,
                       Bit#(32) haystackLen);
endinterface
interface StrstrIndication;
  method Action searchResult(Int#(32) v);
  method Action setupComplete();
endinterface
```

The hardware implements the StrstrRequest interface, which the software invokes (remotely) to specify the search string and the location in flash memory to search. The software implements the StrstrIndication interface, which the hardware invokes (remotely) to notify the software of configuration completion or search results. The interface compiler generates a separate portal for each of these interfaces. Within each portal, a dedicated unidirectional FIFO is assigned to each logical interface method.

In our initial implementation the accelerator does not access system memory directly, so the search string is transmitted to the accelerator one character at a time via the `setupNeedle` method. We will see in Section 4.3 how to use a pointer to system memory instead.

### 4.1.1 *Invoking Hardware from Software*

Because the StrStrRequest functionality is implemented in hardware, the Connectal interface compiler generates a C++ **proxy** with the following interface to be invoked by the application software:

```
class StrStrRequestProxy : public Portal {
public:
  void setupNeedle(uint32_t needleChars);
  void search(uint32_t haystackPtr,
              uint32_t haystackLen);
};
```

The implementation of StrStrRequestProxy marshals the arguments of each method and en-queues them directly into their dedicated hardware FIFOs. To execute searches in the FPGA fabric over data stored in flash memory, the software developer simply instantiates `StrStrRequestProxy` and invokes its methods:

```
StrStrRequestProxy *proxy =
            new StrStrRequestProxy(...);
proxy->search(haystackPtr, haystackLen);
```

On the FPGA, the user implements the application logic as a BSV module with the StrStrRequest interface. A **wrapper** is generated by the interface compiler to connect this module to the hardware FIFOs. The wrapper unmarshals messages that it receives and then invokes the appropriate method in the StrStrRequest interface. Here is the BSV code that instantiates the generated wrapper and connects it to the user's `mkStrStr` module.

```
StrStrRequest strStr <- mkStrStr(...);
StrStrRequestWrapper wrapper <-
    mkStrStrRequestWrapper(strStr);
```

Figure 8 shows how all the pieces of an application implemented using Connectal work together when hardware functionality is invoked remotely from software. Direct access to the memory mapped hardware FIFOs by the generated proxy running in user-mode is key to the efficiency of our implementation strategy.

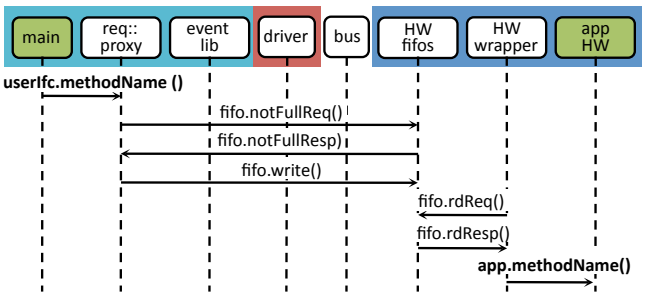


**Figure 8: SW invokes HW: 'main' and 'app HW' are implemented by the user.**

### 4.1.2 Invoking Software from Hardware

Invoking software from hardware takes a slightly different form, due primarily to the fact that "main" is still owned by software. Since the direction of the remote invocation is reversed, the proxy on this path will be instantiated on the FPGA and the wrapper instantiated on host side. The user implements the StrStrResponse interface in software and connects it to the generated wrapper using C++ subclasses:

```
class StrStrResponse:
  public StrStrResponseWrapper {
    ...
  void searchResult(int32_t v) {...}
}
```

The StrStrResponseWrapper constructor registers a pointer to the object with the event library which keeps track of all instantiated software wrappers. The wrapper implementation unmarshals messages sent through the hardware FIFOs and invokes the appropriate subclass interface method. To activate this path, main simply instantiates the response implementation and invokes the library event handler:

```
StrStrResponse *response =
  new StrStrResponse(...);
while(1)
  portalExec_event();
```

On the invocation side, the interface compiler generates a proxy which the application logic instantiates and invokes directly:

```
StrStrResponseProxy proxy <-
     mkStrStrRequestProxy();
StrStrRequest strStr <-
     mkStrStr(... proxy.ifc ...);
```

Figure 9 shows how all the pieces of an application collaborate when software functionality is being invoked from hardware.

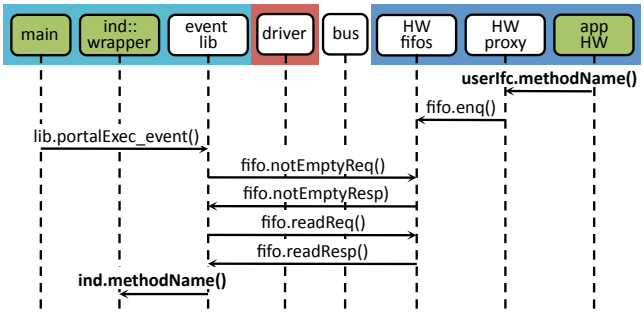


**Figure 9: HW invokes SW: 'main', 'ind::wrapper', and 'app HW' are implemented by the user.**

The simplest software execution environment for the string search accelerator is to have a single thread making requests and waiting for responses as follows:

```
void search(char *str){
  StrStrRequestProxy *req =
    new StrStrRequestProxy(...);
  StrStrResponse *resp =
    new StrStrResponse(...);
  while (char c = *str++)
    req->setupNeedle(c);
  // start search
  req->search(...);
  // handle responses from the HW
  while(1)
    portalExec_event();
}
```

The call to portalExec_event() checks for a response from HW. If there is a pending response, it invokes the method corresponding to that FIFO in the wrapper class. This generated method reads out a complete message from the FIFO and unmarshals it before invoking the user-defined call-back function, which in this case would be `StrStrResponse::searchResult`.

### 4.1.3 Connecting To Flash

On our target platform, the flash memory array is connected directly to the FPGA chip, and DDR signals are used to read/write/erase flash memory cells. The RTL required to communicate with the memory requires some commonly used functionality, such as *SerDes* and DDR controllers, both of which are included in the BSV libraries distributed as part of the Connectal framework.

## 4.2 Multithreading The Software

In many cases, we would like to avoid a hardware-to-software path which requires the software to poll a hardware register on the other side of a bus for relatively infrequent events. To accommodate this, the Connectal framework generates interrupts which are raised when hardware invokes software interface methods. The generic Connectal driver connects these signals to the Linux kernel and the software wrappers can exploit then by calling poll. Connectal applications often use a separate thread to execute hardware-to-software asynchronous invocations, since dedicated thread can put itself to sleep until the hardware raises an interrupt. The "main" thread is free to do other work and can communicate with the "indication" thread using a semaphore as shown below:

```
class StrStrResponse:
    public StrStrResponseWrapper {
  sem_t sem;
  int v;
  void searchResult(int32_t v) {
    this->response = v;
    sem_post(&sem);
  }
  void waitResponse(){sem_wait(&sem);}
};
StrStrResponse *resp;
StrStrRequestProxy *req;
int search(char *str){
  while (char c = *str++)
    req->setupNeedle(c);
  // start search
  req->search(...);
  // wait for response
  resp->waitResponse();
  // return result
  return resp->v;
}
```

The polling thread is started by a call to `portalExec_start()`, which ultimately invokes the `portalExec_poll()` function implemented in the Connectal event library. `portalExec_poll()` invokes the system call **poll** on the FDs corresponding to all the indication or response portals, putting itself to sleep. When an interface method is invoked in the hardware proxy, an interrupt is raised, waking the indication thread. A register is read which indicates which method is being called

and the corresponding wrapper method is invoked to read-/marshal the arguments and invoke the actual user-defined methods. Figure 10 shows this process.

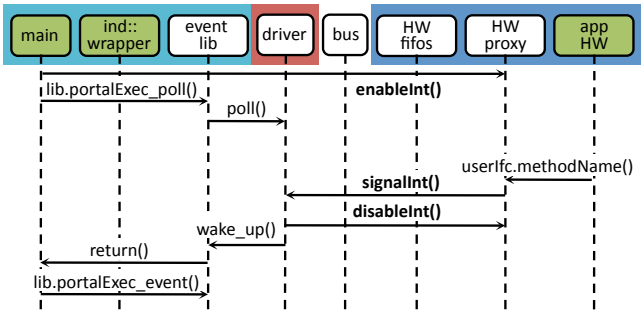


**Figure 10: HW invokes SW using interrupts**

Multithreading often leads to simultaneous access to shared hardware resources. If a software solution to protect these resources (such as mutex) is not available, the hardware interface can be refactored into separate portals, one for each control thread.

Each interface will generate a separate Portal which is assigned its own address space and Linux device. Using Linux devices in this way enables access control restrictions to be specified individually for each portal. This feature can be used to grant different users or processes exclusive access and prevent unauthorized access to specific pieces of hardware functionality.

## 4.3 Shared Access to Host Memory

In the first three refinements presented in Section 2, all communication between hardware and software takes place through register-mapped IO. The final refinement in Section 2.4 is to grant hardware and software shared access to host memory. The interface to the search accelerator shown below has been updated to use direct access to system memory for the search strings:

```
interface StrstrRequest;
  method Action setup(Bit#(32) needlePtr,
                      Bit#(32) mpNextPtr,
                      Bit#(32) needleLen);
  method Action search(Bit#(32) haystackPtr,
                       Bit#(32) haystackLen,
                       Bit#(32) iterCount);
endinterface
interface StrstrIndication;
   method Action searchResult(Int#(32) v);
   method Action setupComplete();
endinterface
```

In order to share memory with hardware accelerators, it needs to be allocated using `portalAlloc`. Here is the search function updated accordingly:

```
int search(char *str){
  int size = strlen(str)+1;
  int fd = portalAlloc(size);
  char *sharedStr = portalMmap(fd, size);
  strcpy(sharedStr, str);
  // send a DMA reference to the search pattern
  req->needle(dma->reference(fd), size);
  // start search
  req->search(...);
  resp->waitResponse();
  ... unmap and free the string
  return resp->v;
}
```

The application allocates shared memory via `portalAlloc`, which returns a file descriptor, and then passes that file descriptor to `mmap`, which maps the physical pages into the application's address space. The file descriptor corresponds to a dmabuf[5], which is a standard Linux kernel mechanism.

To share that memory with the accelerator, the application calls `reference`, which sends a logical to physical address mapping to the hardware's address translator. The call to `reference` returns a handle, which the application sends to the accelerator. Connectal's BSV libraries for DMA enable the accelerator to read or write from offsets to these handles, taking care of address translation transparently.

To fully exploit the data parallelism, `mkStrStr` partitions the search space into $p$ partitions. It instantiates two memory read trees from the Connectal library (`MemreadEngineV`, discussed in Section 3.4), each with $p$ read servers. One set is used by the search kernels to read the configuration data from the host memory, while the other is used to read the "haystack" from flash.

On supported platforms such as Zynq which provide multiple physical master connections to system memory, Connectal interleaves DMA requests over the parallel links. It does this on a per-read-client basis, rather than a per-request basis.

## 4.4 Alternate Portal Implementations

Connectal separates the generation of code for marshalling and unmarshaling method arguments from the transport mechanism used to transmit the messages. This separation enables "swappable" application-specific transport libraries. In light of this, a large number of transport mechanism can be considered. Switching between mechanism requires a simple directive in the project Makefile (more details are given in Section 5).

By default, each portal is mapped to a region of address space and a memory-mapped FIFO channel is generated for each method. Though software access to all FIFO channels in a design may occur through single bus slave interface, Connectal libraries implement their multiplexing to ensure that each FIFO is independent, allowing concurrent access to different methods from multiple threads or processes.

The default portal library implements the method FIFOs in the hardware accelerator. This provides the lowest latency path between hardware and software, taking about 1 microsecond to send a message. If higher bandwidth or transaction rates are needed, FIFOs implemented as a ring buffer in DRAM can be used instead. This requires more instructions per message send and receive, but may achieve higher throughput between the CPU and hardware.

During the design exploration process, a component originally implemented on the FPGA may migrate to software running on the host processor. Remote invocations which were originally from software to hardware must be recast as software to software. Without changing the IDL specification, the transport mechanism assigned to a portal can be re-specified to implement communication between software components running either on the same host or across a network.

Connectal uses UNIX sockets or shared memory to transport messages between the application software components or the hardware simulator. In other situations, TCP or UDP can be used to transport the messages to hardware running on another machine. Viable connections to the FPGA

board range from low-speed interconnects such as JTAG, SPI, to higher-speed interconnects such as USB or Aurora over multi-gigabit per second transceivers.

# 5. WORKFLOW USING CONNECTAL

In this section, we give an overview of the Connectal workflow and toolchain. The complete toolchain, libraries, and many running examples may be obtained at *www.connectal.org* or by emailing *connectal@googlegroups.com*.

## 5.1 Top level structure of Connectal applications

The simplest Connectal application consists of 4 files:

**Makefile** The top-level Makefile defines parameters for the entire application build process. In its simplest form, it specifies which Bluespec interfaces to use as portals, the hardware and software source files, and the libraries to use for the hardware and software compilation.

**Application Hardware** Connectal applications typically have at least one BSV file containing declarations of the interfaces being exposed as portals, along with the implementation of the application hardware itself.

**Top.bsv** In this file, the developer instantiates the application hardware modules, connecting them to the generated wrappers and proxies for the portals exported to software. To connect to the host processor bus, a parameterized standard interface is used, making it easy to synthesize the application for different CPUs or for simulation. If CPU specific interface signals are needed by the design (for example, extra clocks that are generated by the PCIe core), then an optional CPU-specific interface can also be used.

If the design uses multiple clock domains or additional pins on the FPGA, those connections are also made here by exporting a 'Pins' interface. The Bluespec compiler generates a Verilog module from the top level BSV module, in which the methods of exposed interfaces are implemented as Verilog ports. Those ports are associated to physical pins on the FPGA using a physical constraints file.

**Application CPP** The software portion of a Connectal application generally consists of at least one C++ file, which instantiates the generated software portal wrapper and proxies. The application software is also responsible for implementing main.

## 5.2 Development cycle

After creating or editing the source code for the application, the development cycle consists of four steps: generating makefiles, compiling the interface, building the application, and running the application.

**Generating Makefiles** Given the parameters specified in the application Makefile and a platform target specified at the command line, Connectal generates a target-specific Makefile to control the build process. This Makefile contains the complete dependency information for the generation of wrappers/proxies, the use of these wrappers/proxies in compiling both the software and hardware, and the collection of build artifacts into a package that can be either run locally or over a network to a remote 'device under test' machine.

**Compiling the Interface** The Connectal interface compiler generates the C++ and BSV files to implement wrappers and proxies for all interfaces specified in the application Makefile. Human readable **JSON** is used as an intermediate representation of portal interfaces, exposing a useful debugging window as well as a path for future support of additional languages and IDLs.

**Building the Application** A target in the generated Makefile invokes GCC to compiler the software components of the application. The Bluespec compiler (bsc) is then invoked to compiler the hardware components to Verilog. A parameterized Tcl scripts is used to drive Vivado to build the Xilinx FPGA configuration bitstream for the design.

A Connectal utility called *fpgamake* supports specification of which Bluespec and Verilog modules should be compiled to separate netlists and to enable separate place and route of those netlists given a floor plan. Separate synthesis and floor planning in this manner can reduce build times, and to make it easier to meet timing constraints.

Another Connectal utility called *buildcache* speeds recompilation by caching previous compilation results and detecting cases where input files have not changed. Although similar to the better-known utility *ccache*, this program has no specific knowledge of the tools being executed, allowing it to be integrated into any workflow and any tool set. This utility uses the system call **strace** to track which files are read and written by each build step, computing an 'input signature' of the MD5 checksum for each of these files. When the input signature matches, the output files are just refreshed from the cache, avoiding the long synthesis times for the unchanged portions of the project.

**Running the Application** As part of our goal to have a fully scripted design flow, the generated Makefile includes a `run` target that will program the FPGA and launch the specified application or test bench. In order to support shared target hardware resources, the developer can direct the run to a particular machines, which can be accessed over the network. For Ubuntu target machines, ssh is used to copy/run the application. For Android target machines, 'adb' is used.

## 5.3 Continuous Integration and Debug Support

Connectal provides a fully scripted flow in order to make it easy to automate the building and running of applications for continuous integration. Our development team builds and runs large collections of tests whenever the source code repository is updated.

Connectal also provides trace ring buffers in hardware and analysis software to trace and display the last transactions on the PCIe or AXI memory bus. This trace is useful when debugging performance or correctness problems, answering questions of the form:

- What were the last memory requests and responses?
- What was the timing of the last memory request and responses?

| | KC705 | VC707 | ZYBO | Zedboard | ZC702 | ZC706 | Parallel | Mini-ITX |
|---|---|---|---|---|---|---|---|---|
| HW → SW | 3 | 3 | X | 0.80 | 0.80 | 0.65 | X | 0.65 |
| SW → HW | 5 | 5 | X | 1.50 | 1.50 | 1.10 | X | 1.10 |

**Figure 11: Latency ($\mu$s) of communication through portals on supported platforms**

| | KC705 | VC707 | ZYBO | Zedboard | ZC702 | ZC706 | Parallel | Mini-ITX |
|---|---|---|---|---|---|---|---|---|
| Read | 1.4 | 1.4 | X | 0.8 | 0.8 | 1.6 | X | 1.6 |
| Write | 1.4 | 1.4 | X | 0.8 | 0.8 | 1.6 | X | 1.6 |

**Figure 12: Maximum bandwidth (GB/s) between FPGA and host memory using Connectal RTL libraries on supported platforms**

- What were the last hardware method invocations or indications?

# 6. PERFORMANCE OF GENERATED SYSTEMS

A framework is only useful if it reduces the effort required by developers to achieve the desired performance objective. Trying to gauge the relative effort is difficult since the authors implemented both the framework and the running example. On PCIE-based platforms we were able to reduce the time required to search for a fixed set of strings in a large corpus by an order of magnitude after integrating hardware acceleration using Connectal. Performance improvements on the Zynq-based platforms was even greater due to the relative processing power of the ARM CPU and scaled with the number of bus master interfaced used for DMA. In the Connectal framework, developing these applications took very little time.

## 6.1 Performance of Portals

The current implementation of HW/SW **portal** transfers 32 bits per FPGA clock cycle. Our example designs run at 100MHz to 250MHz, depending on the complexity of the design and the speed grade of the FPGA used. Due to their intended use, the important performance metric of Portals is latency. These values are given in Figure 11.

The Xilinx KC705 and VC707 boards connect to x86 CPUs and system memory via PCIe gen1. The default FPGA clock for those boards is 125MHz. The other platforms use AXI to connect the programmable logic to the quad-core ARM Cortex A9 and system memory. The ZYBO, Zedboard and ZC702 use a slower speed grade part on which our designs run at 100MHz. The ZC706 and Mini-ITX use a faster part on which many of our designs run at 200MHz. The lower latency measured on the ZC706 reflects the higher clock speed of the latency performance test.

## 6.2 Performance of Reads/Writes of System Memory

For high bandwidth transfers, we assume the developer will have the application hardware read or write system memory directly. Direct access to memory enables transfers with longer bursts, reducing memory bus protocol overhead. The framework supports transfer widths of 32 to 128 bits per cycle, depending on the interconnect used.

Our goal in the design of the library components used to read and write system memory is to ensure that a developer's application can use all bandwidth available to the FPGA when accessing system memory. DMA Bandwidth on supported platforms is listed in Figure12.

On PCIe systems, Connectal currently supports 8 lane PCIe gen1. We've measured 1.4 gigabytes per second for both reads and writes. Maximum throughput of 8 lane PCIe gen1 is 1.8GB/s, taking into account 1 header transaction per 8 data transactions, where 8 is the maximum number of data transactions per request supported by our server's chipset. The current version of the test needs some more tuning in order to reach the full bandwidth available. In addition, we are in the process of updating to 8 lane PCIe gen2 using newer Xilinx cores.

Zynq systems have four "high performance" ports for accessing system memory. Connectal enables an accelerator to use all four. In our experiments, we have been able to achieve 3.6x higher bandwidth using 4 ports than using 1 port.

# 7. RELATED WORK

A number of research projects, such as Lime [8], BCL [9], HThreads [10], and CatapaultC [11] (to name just a few) bridge the software/hardware development gap by providing a single language for developing both the software and hardware components of the design. In addition, Altera and Xilinx have both implemented OpenCL [12] on FPGAs [13, 14] in an attempt to attract GPU programmers.

The computation model of software differs significantly from that of hardware, and so far none of the unified language approaches deliver the same performance as languages designed specifically for hardware or software. Connectal is intended to be used for the design of performance-critical systems. In this context we think that designers prefer a mix of languages specifically designed for their respective implementation contexts.

Infrastructures such as LEAP [15], Rainbow [16], and OmpSs [17] (to name just a few) use resource abstraction to enable FPGA development. We found that in their intended context, these tools were easy to use but that performance tuning in applications not foreseen by the infrastructure developers was problematic.

Some projects such as TMD-MPI [18], VFORCE/ VSIPL++ [19], and GASNet/GAScore [20] target only the hardware software interface. These tools provide message passing capabilities, but rely on purely operational semantics to describe the HW/SW interface. Apart from the implementation details, Connectal distinguishes itself by using an IDL to enforce denotational interface semantics.

UIO [7] is a user-space device driver framework for Linux. It is very similar to the Connectal's portal device driver, but it does not provide a solution to multiple device nodes per hardware device. The portal driver provides this so that different interfaces of a design may be accessed independently, providing process boundary protection, thread safety, and the ability for user processes and the kernel both to access the hardware device.

# 8. CONCLUSION

Connectal bridges the gap between software and hardware development, enabling developers to create integrated solutions rapidly. With Connectal, we take a pragmatic approach to software and hardware development in which we try to avoid any dependence on proposed solutions to open research problems.

Use of Connectal's interface compiler ensures that software and hardware remain consistent and make it easy to update the hardware/software boundary as needed in a variety of execution contexts. The generated portals permit concurrent and low-latency access to the accelerator and enable different processes or the kernel to have safe isolated access through dedicated interfaces. Support for sharing memory between software and hardware makes it easy to achieve high transfer speeds between the two environments.

Connectal supports Linux and Android operating systems running on x86 and ARM CPUs. It currently supports Xilinx FPGAs and runs on the full range of Xilinx Series 7 devices. Our fully-scripted development process enables the use of continuous integration of software and hardware development. Integrating software development early makes it easier to ensure that the complete solution actually meets design targets and customer requirements.

# 9. REFERENCES


[1] Bluespec Inc., `http://www.bluespec.com`.

[2] J. C. Hoe, "Operation-Centric Hardware Description and Synthesis," Ph.D. dissertation, MIT, Cambridge, MA, 2000.

[3] J. C. Hoe and Arvind, "Operation-Centric Hardware Description and Synthesis," *IEEE TRANSACTIONS on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 9, September 2004.

[4] Z. G. Alberto Apostolico, *Pattern Matching Algorithms*, 1997, ch. 1, pp. 7–11, mp algorithm.

[5] S. Semwal, "DMA Buffer Sharing API Guide," https://www.kernel.org/doc/Documentation/dma-buf-sharing.txt.

[6] Y. A. Khalidi and M. N. Thadani, "An Efficient Zero-Copy I/O Framework for UNIX," Mountain View, CA, USA, Tech. Rep., 1995.

[7] "The Userspace I/O HOWTO," https://www.kernel.org/doc/htmldocs/uio-howto/index.html.

[8] S. S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah, "Liquid metal: Object-oriented programming across the hardware/software boundary," in *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, Berlin, Heidelberg, 2008.

[9] M. King, N. Dave, and Arvind, "Automatic generation of hardware/software interfaces," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 325–336. [Online]. Available: http://doi.acm.org/10.1145/2150976.2151011

[10] W. Peck, E. K. Anderson, J. Agron, J. Stevens, F. Baijot, and D. L. Andrews, "Hthreads: A computational model for reconfigurable devices," in *Proceedings of the 2006 International Conference on Field Programmable Logic and Applications (FPL), Madrid, Spain, August 28-30, 2006*, 2006, pp. 1–4. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/FPL.2006.311336

[11] Mentor Graphics, "Catapult-C," http://www.mentor.com/products/esl/.

[12] The Kronos Group, https://www.khronos.org/registry/cl/.

[13] Altera Inc., http://www.altera.com/products/software/opencl/opencl-index.html.

[14] Xilinx Inc., http://www.xilinx.com.

[15] K. Fleming, H. Yang, M. Adler, and J. S. Emer, "The LEAP FPGA operating system," in *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014*, 2014, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/FPL.2014.6927488

[16] K. Jozwik, S. Honda, M. Edahiro, H. Tomiyama, and H. Takada, "Rainbow: An operating system for software-hardware multitasking on dynamically partially reconfigurable fpgas," *Int. J. Reconfig. Comp.*, vol. 2013, 2013. [Online]. Available: http://dx.doi.org/10.1155/2013/789134

[17] A. Filgueras, E. Gil, D. Jiménez-González, C. Alvarez, X. Martorell, J. Langer, J. Noguera, and K. A. Vissers, "Ompss@zynq all-programmable soc ecosystem," in *The 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '14, Monterey, CA, USA - February 26 - 28, 2014*, 2014, pp. 137–146. [Online]. Available: http://doi.acm.org/10.1145/2554688.2554777

[18] M. Saldaña, A. Patel, C. A. Madill, D. Nunes, D. Wang, P. Chow, R. Wittig, H. Styles, and A. Putnam, "MPI as a programming model for high-performance reconfigurable computers," *TRETS*, vol. 3, no. 4, p. 22, 2010. [Online]. Available: http://doi.acm.org/10.1145/1862648.1862652

[19] N. Moore, M. Leeser, and L. A. S. King, "Vforce: An environment for portable applications on high performance systems with accelerators," *J. Parallel Distrib. Comput.*, vol. 72, no. 9, pp. 1144–1156, 2012. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2011.07.014

[20] R. Willenberg and P. Chow, "A remote memory access infrastructure for global address space programming models in fpgas," in *The 2013 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '13, Monterey, CA, USA, February 11-13, 2013*, 2013, pp. 211–220. [Online]. Available: http://doi.acm.org/10.1145/2435264.2435301