Splice: A Standardized Peripheral Logic and Interface Creation Engine

Justin Thiel and Ron K. Cytron

Washington University
Department of Computer Science and Engineering
St. Louis, Missouri 63130 USA
{jthiel, cytron}@cs.wustl.edu

Abstract

Recent advancements in FPGA technology have allowed manufacturers to place general-purpose processors alongside user-configurable logic gates on a single chip. At first glance, these integrated devices would seem to be the ideal deployment platform for hardware-software co-designed systems, but some issues, such as incompatibility across vendors and confusion over which bus interfaces to support, have impeded adoption of these platforms. This paper describes the design and operation of Splice, a software-based code generation tool designed to address these types of issues by providing a bus-independent structure that allows end-users to integrate their customized peripheral logic easily into embedded systems.

1. Introduction and Motivation

In essence, Splice is an interface-generating application that processes a set of C-like function prototypes as input and generates a software driver and HDL logic stub to handle the I/O operations of each defined function. This high-level view is shown below in Figure 1. Through the use of templates, dynamic libraries, and well defined compatibility mechanisms, Splice is able to generate interfaces for an extremely varied range of target bus mechanisms from the same set of input prototypes. By filling in the HDL logic stubs generated by the tool with “business logic,” a wide variety of hardware-based functions can be both modeled and implemented.

The advantages of this approach to hardware-software co-design are threefold:

1) Software driver creation is simplified: Often, hardware designers are confronted with the task of converting pre-existing software routines into hardware logic blocks,

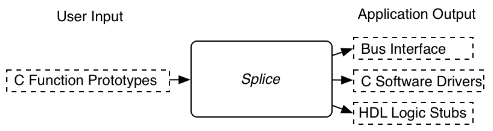

Figure 1. High-Level Functionality of Splice

with the goal of increasing overall system performance. In most cases, designing fast hardware blocks that are functionally equivalent to existing software is a fairly straightforward task, due to the fact that most software source files contain easily identifiable repetitive portions of sequential code that can be converted into logical statements and performed in parallel. However, it is unclear how to maintain some semblance of compatibility with the existing application structure while selectively porting pieces of the code base to hardware.

By using Splice to automate hardware and software interface creation, the task of selectively converting software functions into hardware logic blocks is vastly simplified. This simplification occurs primarily because Splice generates its output files from a set of C-like function definitions. As a result, an end-user can provide the application with a set of extant function prototypes and the tool will generate a set of software drivers with the same calling conventions. These drivers can then be dropped into the existing application as replacements for the software routines, and used to activate the hardware implementation of each function without the need to modify any additional source code. If the end-user is starting from scratch, the drivers generated by Splice form the ideal starting point for application development.

2) Vendor lock-in is eliminated: Currently available SoC FPGAs from vendors such as Xilinx and Altera embed microprocessors and bus interfaces that are fundamentally in-

This work was supported by National Science Foundation grant CNS-0313203, the Air Force Research Laboratory, and the Boeing Corporation.

compatible with one another, while a number of companies such as Gaisler Research provide soft-core (or logic-based) processors that offer their own unique set of interconnects and capabilities [11, 8, 1, 3]. Each of these bus mechanisms has its own timing characteristics, transfer protocol, and feature set, making it next to impossible for a developer to optimize a design for every available platform. Developers might thus omit support for the advanced features that a device is capable of providing, in the interest of simplicity and meeting time-to-market demands.

Splice addresses this issue by defining a bus-independent peripheral logic interface, called the **Splice Interface Standard** (SIS), which is observed by all user-logic files generated by the tool. The SIS transparently handles all interactions between the various bus mechanisms that the tool supports, providing a unified peripheral interface from the point of view of the end-user. As a result, Splice-compliant designs can be moved between FPGA or bus architectures with little or no effort on the part of the developer, assuming that the desired target interface is supported by the tool.

3) Future compatibility is ensured: Due to the rapid pace of advancements in the areas of FPGAs and embedded processors, it is reasonable to assume that the bus interfaces provided by current FPGA-based SoCs will not be available five years from now. Furthermore, even if comparable FPGAs are still in production, faster (incompatible) devices would likely be available, providing a more attractive platform for system deployment. As a result, any legacy hardware designs that were tied to a specific system bus would have to be at least partially recoded to interface with the newly available devices.

In an effort to avoid "reinventing the wheel" each time a new type of FPGA device is released, Splice provides support for user-definable bus interfaces. By following a simple API, end-users can create SIS-compliant interface adapters that are loaded by the tool at run-time via dynamic libraries. Through these libraries, hardware and software interface code can be generated for bus interfaces that do not yet exist, enabling existing designs to be ported forward with minimal effort.

The remainder of this paper provides an overview of the implementation of Splice and is organized as follows. The next section provides a discussion of related work, followed by sections that describe the various hardware and software mechanisms provided by the tool. A real-world use-case for Splice is presented in Section 7 along with experimental results, followed by some concluding remarks in Section 8.

2. Related Work

The Splice approach to bus-interface generation and linkage somewhat resembles the methods employed by distributed object specification systems such as CORBA [6]. Like CORBA, Splice generates functional bindings based on relatively abstract interface definitions provided by the developer. CORBA, however, operates solely in the realm of software and thus is incapable of implementing the types of hardware-software interfaces that Splice provides. Furthermore, due to the complexities that a system such as CORBA must handle, it is forced to rely on fairly abstract bindings to handle each and every situation in which it might be deployed. In contrast, the structure of each interface file generated by Splice is governed by well defined protocols and microprocessor ISAs, allowing for better optimization of I/O transfers across the software-hardware boundary regardless of the bus interface that is targeted by the end-user.

Splice can also be compared with hardware-centric design automation languages such as Handel-C [2] or System-C [4]. Much like Splice, these types of systems operate upon C-like source files to create platform-independent hardware-based netlists and/or HDL files. In sharp contrast to Splice, however, both Handel-C and System-C infer not only interface mechanisms, but also business logic from the input source code. The downside to this approach to hardware design is that it is often unclear what type of calculation hardware will be generated from a particular set of end-user input. Splice, on the other hand, does not attempt to infer calculation logic for the user-defined functions it operates upon, and is thus aimed at a somewhat different group of end-users than any other currently available software/hardware generation systems.

3. Defining Hardware-Software Interfaces

Function prototypes for Splice are patterned after the ANSI C programming language. Support for all standard C data types is provided along with a type-defining (`typedef`) mechanism that enables support for any additional types that a developer might need. By providing support for a commonly used language, many already constructed C function prototypes can be passed directly into the tool from a header file without modification. This, in turn, makes it easy for the end-user to maintain software-side interface compatibility within a pre-exiting application.

Due to the fact that ANSI C was not originally designed as an HDL, there are currently some features of the language, such as structs and pass-by-reference, that are either not supported by Splice or require small modifications to be deployed on hardware. In addition, a variety of language extensions were added to support hardware-specific constructs and provide access to advanced interface features. The remainder of this section provides a description of the various types of syntax extensions Splice supports and how they can be activated within user-defined function prototypes. Sections 5 and 6 describe how these features are implemented in the generated user-logic stubs and software drivers that Splice produces as output, while Figure 2 summarizes the various language extensions for which Splice provides support.

Defining Pointer-Based Data Transfers The most commonly used syntax extension defined by Splice is support for data pointers. In typical C applications, data pointers are extremely powerful and able to pass large amounts of data between functions with little effort on the part of the end-user. Hardware devices, however, cannot typically be passed pointers into memory or unbounded arrays due to the physical resource limitations of FPGAs. Therefore, when pointer-type inputs and outputs are required, the user must define how many items need to be transmitted across the bus interface to synchronize with the hardware.

One method used to define pointer transmission is via an explicit numeric declaration. This type of declaration specifies precisely how many items of a particular data type need to be transferred from main memory into hardware and is supported on both input and output data structures. To make use of explicit transfers, standard pointer declarations need to be extended with the "colon" (or ':') operator. As an example, a declaration of `void some_function(int* x:5)` would mean that 5 integers should be passed into the hardware implementation of '`some_function`' as input from a unbounded array (x) each time its corresponding driver is called.

As an alternative to explicit pointer declarations, the end-user also has the option of using an implicit index value to define how many items should be passed into or out of a hardware function. Implicit array transfers reference the values of other inputs listed in the prototype to determine how many items should be transferred, allowing end-users to create "dynamic" hardware functions that operate on variable-length parameters. Much like explicit transfers, the "colon" operator extension is used to define an implicit declaration. As an example, a declaration of `void some_function(char x, int* y:x)` would imply that `some_function` takes in a total of 'x' integers transferred from the 'y' integer pointer array.

Regardless of the type of pointers used within a function declaration, the data they require is always passed by-value into the hardware as opposed to by-reference. In effect, this means that a pointer passed into a function as an input cannot be written to by the hardware and then returned as an output. As a result, only a single return value (or array of values) can be passed back from the hardware by any single function call.

Defining "Packed" Data Transfers In addition to providing support for explicit and implicit pointers, Splice also defines an additional language extension to assist in the packing of data values. In simple terms, data packing is a mechanism that allows a designer to transfer multiple data entries to or from a compatible target bus in a single transmission cycle. For instance, if provided with a 32-bit wide interface, a total of four 8-bit characters could be transferred across the bus in a single cycle if data packing is used, resulting in a 75% reduction in transmission time versus transferring one character at a time.

To enable the use of data packing within function declarations, the "plus" (or '+') extension was added to the Splice syntax. Use of this extension must be combined with either an explicit or implicit pointer declaration to be recognized by the tool. As an example, the statement `void some_function(char* x:8+)` would imply that the user wants to transmit 8 characters across the bus in packed mode. This would enable all 8 values of 'x' to be transmitted in 2 cycles (4 values/transmission) as opposed to 8, speeding up operation and freeing the bus sooner for additional transactions.

Defining Direct Memory Access (DMA) Transfers Along with extensions to handle various array operations, Splice also provides syntax enhancements that allow end-users to transfer input and output data parameters via DMA. Enabling this feature allows the designer to transfer information to and from a hardware peripheral without direct CPU-to-memory-to-bus interaction, thus freeing up the processor to perform other tasks while data transmission takes place. This feature is especially helpful in cases where a function requires a large block of input to operate, and thus a large number of CPU cycles could be saved by automating said requests. Support for this feature is limited to use with bus interface types that have built-in physical support for DMA operations. In other words, Splice is not capable of providing DMA support to a bus that does not already have such capabilities.

Assuming that a DMA-supporting target bus is selected, the feature can be activated by including the "caret" (or '^') syntax extension within a user-defined function prototype. As is the case with packing, use of this extension is limited to those I/O parameters that make use of either explicit or implicit transmission. As an example, the statement `void some_function(int* x:8^)` would imply that the user wants to create a hardware block that takes in 8 integers via DMA. This would enable `some_function` to be activated without the need for the CPU to physically pass data into the user-logic block.

Defining Multiple Hardware Instances Another advanced feature that Splice provides syntax extensions for is the ability to automatically generate multiple copies of the same hardware function from single prototype declaration. This feature is useful for a multi-threaded software application in which the developer wishes to have a copy of a specific peripheral hardware function available for each software thread to use. By having multiple copies of the same hardware function, developers can avoid the performance penalty of having to arbitrate access to a single resource and, as a result, drastically improve performance.

Similar to pointer declarations, this feature is activated by the "colon" operator, but the colon is included at the end of a declaration instead of being inserted on a per-parameter basis. As an example, a declaration of `void some_function(int x, int y):4` would generate four independent copies of `some_function` that could all execute calculations in parallel.

Defining Blocking Function Calls By default, any prototypes defined in Splice that return a value are assumed to be synchronous operations, while any that return 'void' are assumed to asynchronous. That is, if a prototype specifies a 'void' return type then the generated driver for such a function will return control to the user application immediately after all input parameters have been transmitted across the bus. As a result, the end-user is not guaranteed that a particular function has finished processing once the driver call has completed and program execution is resumed.

This system works well for simple "set-it-and-forget-it" functions, but is incapable of handling 'void'-type hardware operations that rely on ordered execution. In an effort to enable support for these types of functions within Splice, an extension for *blocking* hardware function calls was added to the tool. Support for this feature is enabled by setting the return type for a prototype that returns no value as 'wait' instead of 'void'. As an example, the statement `wait some_function(int x, int y)` would imply that the end-user wants to pass two integers (x and y) into a user-logic block and then wait for the hardware to signal that all internal calculations have completed before enabling further bus transactions to occur.

Feature	Extension	Example
Explicit Pointers	: (colon)	void some_func(int* x**:8**);
Implicit Pointers	: (colon)	void some_func(char x, int* y**:x**);
Data Packing	+ (plus)	int some_func(int* x**:8+**);
DMA	ˆ (caret)	int some_func(int* x**:8**ˆ);
Multiple Instances	: (colon)	void some_func(int x)**:4**;
Blocking Calls	wait	**wait** some_func(int x);

Figure 2. Summary of Language Extensions

4. Splice Interface Standard

Splice was designed to provide end-users with an easily expandable hardware-software system generation platform. In order to provide this functionality, the tool makes extensive use of bus-independent hardware and software interfaces that were created to simplify system development and provide a high degree of design flexibility. One such mechanism employed extensively throughout the tool is known as the **Splice Interface Standard** (SIS).

Essentially, the SIS functions as an intermediate interface between an external system bus and the user-logic files generated by Splice. Any system bus of interest must have its signals and transmission protocols adapted to those of the SIS in order to ensure that communication with user-defined functions can take place. The communication signals utilized by the SIS can be seen in Figure 3, and further information in regards to constructing an interface adapter can be found in Section 5 of this paper.

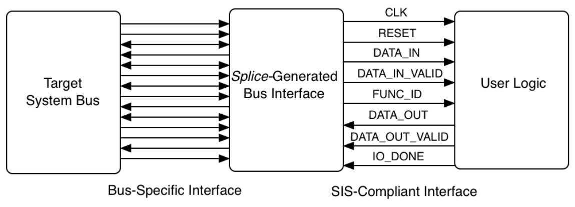

Figure 3. SIS Signaling Conventions

The exact methodology used to adapt the signals of a particular system bus to the SIS will vary depending on the complexity of the interface that is being targeted. There are, however, a number of pre-defined axioms that dictate how the SIS interface should interact with a user-defined function. Through this protocol, which is outlined in Figure 4, almost any type of bus-related transfer operation can be accomplished. Complex operations such as DMA and burst transactions can be handled via the external bus interface and then converted into simple SIS-compliant transactions without the end-user knowing that such a conversion has occurred. As such, every read and write transaction across the SIS appears to be identical from the viewpoint of the user-logic functions, vastly simplifying transmissions compared to traditional bus interface mechanisms.

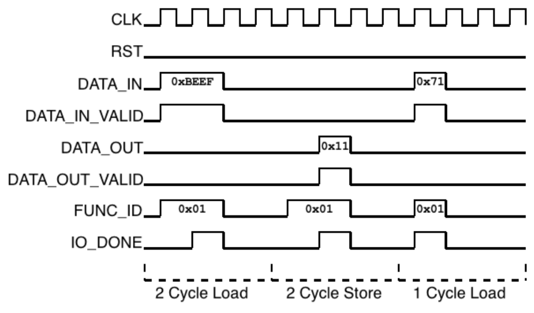

Figure 4. Simplistic SIS Timing Diagram

5. User-Logic and Bus Interface Generation

User logic and interface generation is a multi-step process that consists of three independent stages. In the first stage, one or more top-level interface files are generated to link user hardware blocks to the specified system bus. Next, a bus arbitration file is generated to handle the multiplexing of shared signals between each user-specified function. In the third and final stage, a separate user-logic stub file is generated for each hardware prototype that handles all related input and output operations automatically.

The contents of the logic, arbitration, and interface files that are generated will vary greatly from project to project. This is due to the fact that factors such as the format of the input function prototypes, the bus interface being targeted, and the types of "advanced" features the end-user requires all have an impact on the structure of the resulting stubs. What is constant, however, is the method used to generate the files and the relationships between them.

Bus Interface Generation Generation of bus interfaces is accomplished by consulting a set of reference HDL files that describe the basic logical statements and wired connections that are required to implement the target system interface. These files essentially adapt the bus's native input signals into the format required by the SIS (as described in Section 4) to allow bus-independent hardware blocks to be created and deployed.

Arbitration Unit Generation Once a proper interface file for a peripheral has been generated, Splice then creates an arbitration unit to sit between the bus and user-defined hardware functions in order to multiplex access to the shared output signals defined by the SIS. To perform this arbitration, each user-defined function (or instance of a function if multiple instances are required) is assigned a unique function ID based upon a parameter that is specific to the target bus. Software drivers generated by the tool can then use these identifiers to target particular hardware functions and orchestrate calculations. This method of access control is required because all user-logic blocks are attached to a common connection point (the bus), and thus would corrupt the shared signaling lines if some method of arbitration were not employed.

In theory, the multiplexers defined within the arbitration file could create a performance bottleneck if a set of functions has a high ratio of I/O to business-logic time. In reality, however, hardware functions will typically spend an order of magnitude more time (cycles) calculating results than they will handling I/O operations. As a result, having multiple components share the same bus interconnect should not create a performance bottleneck within the system. Furthermore, by sharing the same bus interface between all hardware functions, any additional connection points on the bus will be available for use by other peripherals, such as Ethernet devices or memory controllers.

User-Logic Stub Generation After interface and arbitration files have been generated for a particular peripheral device, Splice then creates a seperate user-logic stub for each prototype that the end-user has defined. Each generated stub implements a minimum of two distinct functional units known as the ICOB (input-calculation-output block) and the SMB (state machine block) that work together to handle all bus interactions for a particular hardware function. In doing so, these blocks allow each function within a system to operate in an autonomous fashion, while also providing support for any calculation logic that may be added by the end-user. In cases where the default functional units do not provide enough flexibility to implement a desired function, the end-user is free to add any additional blocks that are required. The relationship between the ICOB and SMB is portrayed in Figure 5.

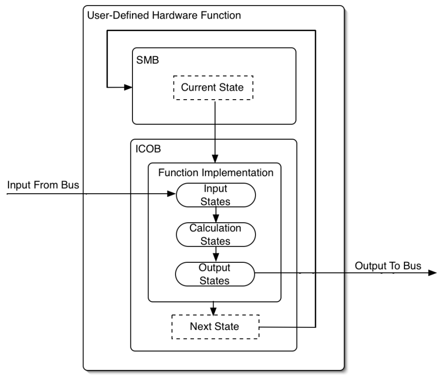

Figure 5. Layout of a Typical User Logic Stub

The ICOB is a clock-sensitive unit that is responsible for implementing all bus interactions for a specified hardware function in the order expressed within its corresponding prototype. By default, all input and output bus-level signaling, such as read and write verification, is handled by the ICOB, but no actual data storage or transfer is done. The reasoning behind this is that end-users may want to allocate their own system-specific structures such a Block RAM or register files for storage that are outside the scope of Splice.

Working in tandem with the ICOB, the SMB is responsible for updating the "current state" signal that is used to move between the various input, calculation and output operations defined within the ICOB. Transitions within the SMB are accomplished via a set of combinational logic statements that are activated each time the ICOB requests a state transition. Since the ICOB is itself a clocked process it will only request a state change a maximum of once per cycle, thus ensuring ordered completion of all function-related operations.

6. Software Driver Generation

The software drivers created by Splice are constructed as simple functions written in the C language. A separate function is created for each prototype that was provided to Splice at run-time, with the resulting calling conventions designed to mimic those provided as input. As such, if Splice is provided with a set of pre-existing software prototypes as input then the drivers generated from them can be used as drop-in replacements for said functions, making it easy to integrate and test the resulting hybrid system.

A longer discussion of software-driver generation will appear in the full paper. For now, the details can be inferred from the example shown in Figure 6. It should be noted, however, that when advanced features such as DMA or multi-valued outputs are required to complete a transaction they are handled internal to the driver, thereby freeing software developers from the burden of implementating such transfers on their own.

```
// ID Used to Target sample_function
#define SAMPLE_FUNCTION_ID 2

// Driver Used to Activate sample_function in HW
float sample_function(int* x, int y)
{
  // Allocate Storage for output
  float result;

  // Transfer Two Values of x
  write_double(SAMPLE_FUNCTION_ID, &x[0]);

  // Transfer One Value of 'y'
  write_single(SAMPLE_FUNCTION_ID, &y);

  // Grab Result from Hardware
  read_single(SAMPLE_FUNCTION_ID, &result);
  return result;
}
```

Figure 6. Autogenerated Driver Code

7. Real World Implementation Example

While Splice is able to provide a wide-array of interface generation capabilities, the tool would be useless if the hardware it generated was substantially slower than that which could be implemented “by hand”. Therefore, in an effort to quantify the performance impact that the bus interfaces and arbiters generated by Splice have upon overall hardware performance, a pre-existing hardware device was chosen and re-implemented to make use of a number of Splice-compliant interfaces. Specific details about the device that was used in this testing are presented below along with a selection of performance results.

The device selected for testing was a linear interpolator that is used within the Sonic Eagle unmanned aerial vehicle (UAV) to approximate continuous flight control data for the aircraft from a set of time-valued samples. The interpolated data is then used to steer the aircraft properly during the time periods in which sampled data is not being received. This device was chosen primarily for the following three reasons:

1. We have access to two pre-existing bus interconnects for the device that were coded by hand for use in previous research.

2. The calculation logic for the device runs in a predictable manner and requires the same numbers of clock cycles to produce results each time it is run, making it simpler to obtain reproducible performance results.

3. The interpolator can be run in four modes (scenarios), each of which require differing amounts of input to execute, thus providing the chance to test the performance of Splice-generated interfaces under a wide range of usage patterns.

For the purposes of testing, a Xilinx ML-403 development board was used as the hardware deployment platform. The ML-403 contains a Virtex4-FX12 FPGA with an embedded PowerPC 405 (PPC405) microprocessor, and a number of supporting peripherals such as 64 MB of DDR-SDRAM, a UART interface, and two separate Ethernet controllers. The onboard CPU is able to communicate with user-logic deployed on the FPGA through three distinct bus interfaces: the Fabric Co-Processor Bus (FCB), the Processor Local Bus (PLB), and the On-chip Peripheral Bus (OPB) [5, 9, 7]. In our testing, only the FCB and PLB were utilized as interfaces for the linear interpolator. The OPB, by virtue of the fact that is designed for use with low-speed peripherals, was deemed too slow to handle the transfer rates required by the device.

In terms of functionality, the FCB is a 32-bit bus that is intended to be used as a co-processor interconnect for a single device. The bus is not directly addressable through memory mappings, but can be accessed via a number of FCB-specific opcodes that are included in the PPC 405 ISA. In addition to simple single-word load and store operations, the FCB also has native support for double- and quad-word burst transmissions [5]. Due to the fact that the interface is not memory-accessible, however, support for DMA transfers is not provided.

Unlike the FCB, the PLB is a general-purpose interconnect designed to interface to a number of different 32/64-bit devices at the same time. Access to the bus is controlled through an arbiter that operates in a similar fashion to those generated by Splice. PLB devices are memory-accessible and can be configured by the end-user to support any number of the advanced features (such as DMA) that the bus provides [9]. One peculiarity of the bus is that although burst transactions are supported, the PPC 405 has no instructions that are capable of activating such transmissions. The impact this has on performance is further explored later in this section.

Each of the four usage scenarios of the linear interpolator operate upon three sets of input values to generate a single output. The exact meanings of these values are not important for the purposes of this analysis since the amount of calculation done in each implementation is constant. As such, a full explanation of the inner workings of the interpolator is omitted for the sake of brevity. The precise number of inputs that each scenario requires is shown in Figure 7. It should be noted that since each set of values transferred to the hardware is contained in a separate data array, it would be impossible to transfer all items across the bus via a single burst or DMA transaction.

Scenario	Set 1	Set 2	Set 3	Total
1	2	1	2	5
2	4	2	4	10
3	8	3	6	16
4	16	4	8	28

Figure 7. Input Parameters for Each Scenario

A total of five interface implementations for the linear interpolator were used in our testing: two hand-coded variants and three generated via Splice. Of the hand-coded implementations, the PLB variant (referred to as "Initial PLB" in the figures provided below) was the product of the first attempt at generating an interface for the linear interpolation device. At the time the interface was coded, the designer was not aware of all of the intricacies of the PLB and thus the interface was not nearly as optimized as it could have been. Thus, the performance results obtained from this interface should be considered representative of what an end-user who is unfamiliar with the protocols of a particular bus would likely create. Conversely, the hand-coded FCB interface ("Optimized FCB") is a highly optimized implementation that was created to replace the slower PLB interconnect.

All three Splice-generated interfaces are attached to an identical user-logic function that makes use of implicit pointer declarations to transfer the required number of values from each of the three datasets depending on the scenario that is run. One of the generated interfaces ("Splice PLB (Simple)"), is a minimally sized PLB interconnect that is capable of orchestrating single-word (32-bit) transmissions across the bus. The generated FCB interface ("Splice FCB"), on the other hand, is able to facilitate double and quad-word transfers for sets of data values that can benefit from such transmissions.

The third and final interface generated for testing via Splice for testing is an additional PLB interconnect ("Splice PLB (DMA)") that contains the supplementary control logic required to support DMA transactions. The use of DMA across the PLB bus is interesting in that it does not benefit transactions of four or fewer data values. This is due to the fact that the DMA circuitry requires a minimum of four bus transactions to setup and take down, thus negating any benefits for lesser transmissions [10].

To perform the testing, a memory-accessible clock-cycle accurate timer was loaded onto the FPGA along with a small Xilinx Embedded Development Kit (EDK) project consisting of the bare minimum hardware to activate each peripheral interface and obtain results. The embedded PPC-405 was clocked at 300 MHz, while the on-chip PLB and FCB interconnects were clocked at 100 MHz. For the hand-coded PLB and FCB interfaces pre-existing drivers were used to transmit data to and from the interpolator, while Splice-created drivers were used for the three generated interfaces. The results of these tests for each interpolation scenario can be seen in Figure 8.

The performance results show that Splice-generated interfaces compare favorably to those generated by hand. Overall, the Splice-generated simple PLB Interface is approximately 25% faster than the naïve hand-coded implementation. Furthermore, the Splice-generated FCB interface is approximately 43% faster than the naïve PLB implementation and only 13% slower than an optimized hand-coded FCB interconnect. For this particular hardware device, DMA transactions enacted via the PLB bus are not very beneficial, representing only a 1-4% performance increase versus a non-DMA implementation.

Besides pure performance results, there is also the issue of how many FPGA resources each implementation consumes. Although reconfigurable logic devices continue to grow in size at a rapid rate, there are still cases where it is difficult to fit all of the logic required for a particular design onto the chip. Resource usage statistics for each of the tested bus interconnects are provided in Figure 9.

The resource usage numbers obtained for each bus implementation are comparable to the performance results. On average, the Splice-generated simple PLB interface consumes about 23% less FPGA resources than the naïve hand-coded implementation. Similarly, the Splice-generated FCB interface requires (on average) 28% less resources than the naïve PLB implementation, and only around 2% more resources than an optimized hand-coded FCB interconnect.

The most surprising usage statistic perhaps, is the astronomical resource usage that results from enabling DMA transactions for a PLB device. In this particular case, the DMA-supporting interface requires anywhere from 57-69%

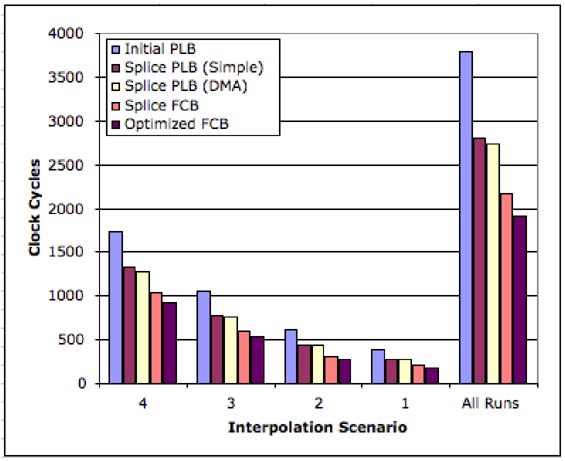

Figure 8. Runtime for Each Scenario

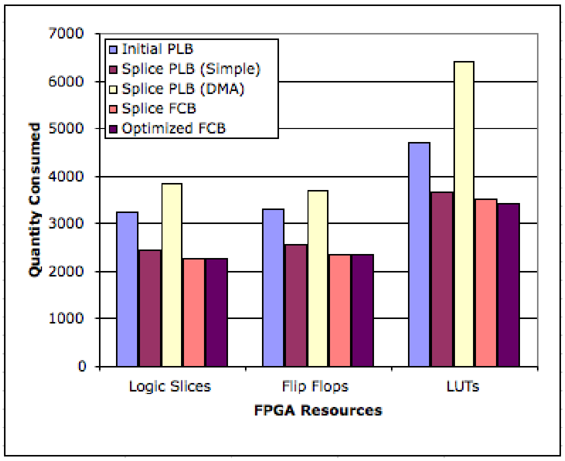

Figure 9. FPGA Resource Consumption

more FPGA resources to implement than the otherwise identical simple PLB interconnect. As such, when an end-user chooses to enable DMA support for a particular device they need to ensure that the possible performance benefit is worth the additional resource cost.

8. Conclusions and Future Work

Manufacturers such as Xilinx and Altera have recently introduced a number of FPGA-based SoC devices in an attempt to provide developers with a flexible embedded system development platform. Issues such as a lack of inter-vendor compatibility between platforms, however, have prevented these devices from being adopted *en masse*. Splice attempts to address such problems by providing flexible and easy-to-use facilities by which end-users can automatically generate interfaces, user-logic stubs, and C-based software drivers for a wide variety of target bus mechanisms from a set of simple C-like function prototypes.

In terms of both performance are resource usage, Splice-generated interfaces compare favorably with manually created optimized interconnects and can outperform “naïve” bus implementations by a significant margin. Furthermore, by virtue of the fact that the tool can generate interconnects almost instantly, Splice allows an end-user to experiment with the various bus mechanisms supported by their SoC without the need to manually code a single interface. This, in turn, saves valuable development time and allows the end-user to focus exclusively on creating optimized “business logic” for their peripheral.

Currently, Splice suffers from a number of limitations, including a lack of support for C-language features such as structs, and a somewhat inflexible implementation of implicit pointers. Furthermore, built-in support for a wider variety of bus interfaces will likely be required in order for the tool to gain wide acceptance. Despite these small issues, however, Splice is still able to provide a set of powerful hardware/software generation tools that enable end-users to generate devices that are easily expandable and that are not instantly outdated each time a new family of FPGAs is introduced.

References

[1] Altera. *Nios II Processor Reference Handbook*, May 2006.

[2] S. Chappell and C. Sullivan. Handel-c for co-processor and co-design of field programmable system on chip. In *Workshop on Reconfigurable Computing and Applications*, September 2002.

[3] J. Gaisler, S. Habnic, and E. Catovic. *GRLIB/LEON3 Users Manual*, August 2006.

[4] IEEE. *IEEE Standard System C Language Reference Manual*, 2005.

[5] H.H. Ng and L. Pillai. Accelerated system performance with the apu controller and extreme dsp slices. *Xilinx Application Notes*, September 2005.

[6] Object Management Group. *CORBA Specification*, January 2006.

[7] Xilinx. *Processor Local Bus (PLB) Specification*, July 2003.

[8] Xilinx. *Microblaze Processor Reference Guide*, October 2005.

[9] Xilinx. *On-Chip Peripheral Bus (OPB) Specification*, December 2005.

[10] Xilinx. *PLB IP Interface Guide*, April 2005.

[11] Xilinx. *PowerPC 405 Processor Block Reference Guide*, 2.1 edition, July 2005.