I/O and Performance Tradeoffs with the FunctionBus during Multi-FPGA Partitioning

Frank Vahid
Department of Computer Science
University of California, Riverside, CA 92521
vahid@cs.ucr.edu, www.cs.ucr.edu

Abstract

We improve upon a new approach for automatically partitioning a system among several FPGA's. The new approach partitions a system's functional specification, now commonly available, rather than its structural implementation. The improvement uses a bus, the FunctionBus, for implementing function calls among FPGA's. The bus can be used with any number of FPGA's, and its protocol uses only a small amount of existing FPGA hardware, requiring no special hardware. While functional rather than structural partitioning can substantially reduce the number of input/output pins (I/O), using the FunctionBus takes such reduction even further. In particular, performance and I/O can be traded-off by varying the bus size, as demonstrated using several examples.

1 Introduction

Partitioning a system among multiple FPGA's has attracted extensive investigation [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. These approaches share one common feature: they each partition a system's *structural implementation* (i.e., a netlist).

In [12], we demonstrate substantial advantages of automatically partitioning a system's *functional specification* rather than its structural implementation. A functional specification is essentially a sequential-program description of a system's desired behavior. Automated functional partitioning was enabled, and began to appear as a viable alternative to structural partitioning, in the early 90's when top-down methodologies using machine-readable functional specification languages (e.g., VHDL and Verilog) gained popularity. Figure 1 illustrates the difference between structural and functional partitioning. In structural partitioning, one first synthesizes a custom-processor hardware structure and then partitions the structure among parts, whereas in functional partitioning, one first partitions the specification into smaller parts, and then synthesizes a custom processor for each part. Functional partitioning's advantages include: (1) tremendously reduced I/O, (2) improved circuit performance, (3) partitioning runtimes in the seconds or minutes rather than hours, (4) order-of-magnitude reductions in behavioral and logic synthesis runtimes, (5) independently readable (and hence simulatable and debuggable) parts, and (6) a consistent technique for both hardware and hardware/software partitioning, recognizing that software and custom hardware are logically equivalent, merely representing different implementation options. These advantages come at the cost of a small increase in gates, and significantly more complex partitioning and estimation tools. Figure 2 highlights two-way partitioning experiments in [12], showing the reductions in I/O (maximum on either part, and total on both parts) obtained when using functional partitioning instead of structural partitioning.

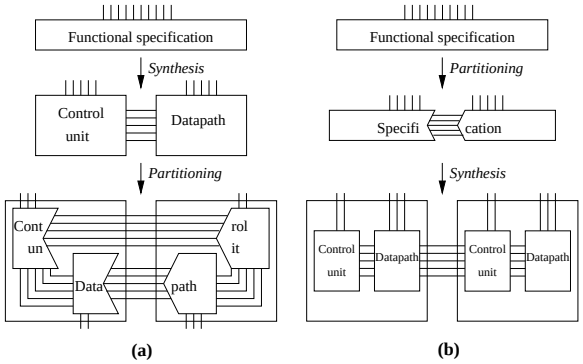

Fig. 1: Partitioning: (a) structurally, (b) functionally.

Several early functional partitioning approaches focused on partitioning arithmetic-level data and control operations among hardware blocks [13, 14, 15, 16, 17], while others have focused on partitioning coarser-grained functions (algorithmic-level subroutines) [18]. More recently, the problem of partitioning among hardware and software blocks has attracted much attention [19, 20, 21, 22, 23, 24, 25, 26]; most of this focus uses coarser-grained functions.

Previous functional partitioning techniques used a cut-edges approach to I/O implementation, similar to structural partitioning techniques. In particular, they first create a graph representation of the system, where each node represents an operation or a function, and each edge represents the flow of data among nodes. They then partition the nodes among parts, and implement a unique set of wires for each data edge that crosses between parts, with one or two control signals for each set. In this paper, we propose a new approach to I/O implementation for functional partitioning, using a single bus for all inter-FPGA communication. In our approach, nodes represent coarse-grained functions from the specification, and each edge represents data

Example	Unpartitioned		Funct. partitioned		Funct. part. w/ bus	
	IO	Size	Max I/O reduction	Total I/O reduction	Max I/O reduction	Total I/O reduction
2p-fact	99	19697	49%	60%	66%	77%
rsa	132	22614	53%	67%	76%	85%
chinese	99	28471	45%	27%	86%	85%
vol	110	17028	33%	29%	39%	39%

Fig. 2: I/O reductions through functional partitioning.

transferred during calls by one function to another. The bus implements these function calls, so we refer to the bus as the *FunctionBus*. In essence, all data edges are multiplexed over a single bus, and all control signals are encoded in an address, also multiplexed over that same bus. Figure 2 illustrates the further reductions in I/O when using a bus rather than cut-edges for I/O implementation after functional partitioning.

A structural-partitioning based technique described in [10] also aims to reduce I/O, particularly for logic emulation systems. The technique analyzes the transfers over inter-FPGA logical wires resulting after structural partitioning, and then multiplexes subsets of those transfers over single physical wires in a pipelined manner, through the use of shift registers, simple distributed controllers, and multiple clocks. Our approach is very different as it is based on functional partitioning, whose advantages were listed earlier.

Functional partitioning techniques are complex and have appeared in many earlier publications [12, 19, 20, 21, 26, 27], so we do not describe such techniques here. Instead, we demonstrate the ability to tradeoff I/O and performance when using the FunctionBus, by varying the bus size and hence making serial versus parallel communication tradeoffs. In the remainder of this paper, we describe the problem, summarize the FunctionBus concept, walk through a small example, summarize tradeoffs obtainable on several real examples, discuss current status and future work, and provide conclusions.

2 Problem description

We focus on partitioning a single functional process among multiple processors (extensions for concurrent processes are discussed in Section 6). This process' execution can be viewed as a series of function calls. Reads and writes of global variables are also treated as function calls, with the input or output data treated as an input or output parameter, and any array indices treated as additional parameters. We assume the functions are non-recursive, which simplifies the FunctionBus implementation, while also being a restriction imposed by most synthesis tools.

Such an execution model has the feature that only one function is active at a time. Thus, when we perform function calls over the bus, there is only one bus user at a time, so there is no bus contention.

We must partition these functions among FPGA's. Although FPGA partitioning is addressed in this paper, the FunctionBus can be applied in an identical manner for most system partitioning problems, such as partitioning among hardware blocks representing ASICs, modules within an ASIC, or even microcontrollers or processors (software blocks). Each FPGA has size and I/O constraints. Functions may have imposed execution-time constraints, applying to that function and all its called functions. A detailed functional partitioning problem formulation can be found in [21].

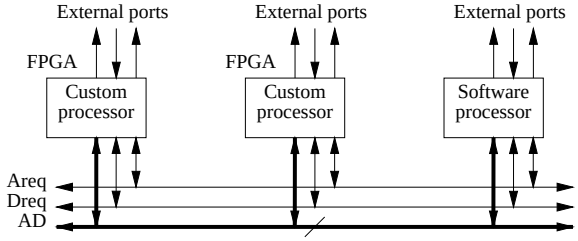

Fig. 3: FunctionBus architecture.

3 The FunctionBus

Figure 3 illustrates the FunctionBus architecture. Several processors are connected by a bus consisting of *AD*, *Areq* and *Dreq* lines. *AD* consists of N lines, used to carry address and data. *Areq* is a single line used to indicate a valid address on *AD*. *Dreq* is a single line used to indicate valid data on *AD*. All lines are bidirectional. Only one processor will control the bus at a time, with the others providing high-impedance values. We refer to N as the *size* of the FunctionBus, even though the number of I/O required will always be $N+2$.

A key feature of the FunctionBus is that N can be made small or large to tradeoff performance with I/O. N can range from 1 to the available number of I/O.

A system executes as a series of function calls. A function A executes on a single FPGA. If this function must call another function B on a different FPGA, A sends B's address over the bus, followed by input parameters to B. B captures these parameters and then executes, possibly calling other functions in like manner. Upon completion, B sends A's address over the bus, followed by any output parameters. A captures these parameters and resumes execution.

Each function must know the return address, i.e., the address of the function's caller. Functions with just one caller have a fixed return address. However, functions with multiple callers will have multiple return addresses. In some cases, the sequence of return addresses from a given function can be determined statically, so we can simply hard-code those addresses. In other cases, the sequence is data dependent and thus can only be known during dynamic execution. In such cases, in order for a function to know the return address, the caller must send its address when calling the function. Note that the non-recursive function requirement means that there will only be one return address per function at a time.

From the above discussion, we see that there are two

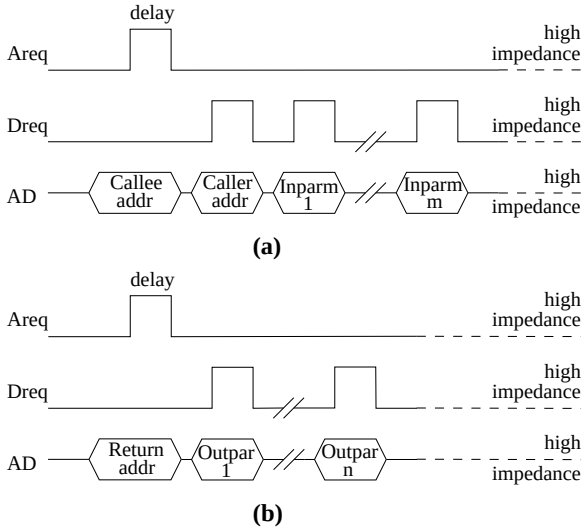

Fig. 4: Timing diagrams: (a) function call, (b) return.

```
FunbusSend(addr, data, num)

// Send the address
AD     = addr
Areq   = 1
wait for fb.delay
Areq   = 0
// Send the data
for i in 1 to num loop
   base = i * fb.size
   AD  = data[base..(base+fb.size)]
   Dreq= 1
   wait for delay
   Dreq= 0
end loop
// Release the bus
Areq = Dreq = AD = high imped.
```

```
FunbusRec(addr, data, num)

// Wait for correct address
wait until Areq=1 and AD=addr

// Receive the data
for i in 1 to num loop
   base = i * fb.size
   wait until Dreq=1
   data[base..(base+fb.size)]← AD
end loop
```

Fig. 5: Send and receive protocols.

types of communications that occur over the FunctionBus. A *function call* sends the address, possibly a return address, and input parameters. A *function return* sends the return address and output parameters.

Both communications use similar protocols, shown in Figure 4. Both begin by placing the address of the receiver function (the callee in (a) and the caller in (b)) on *AD* and pulsing *Areq*. The function call protocol then places the return address on *AD* and pulses *Dreq* (if necessary). Both then send a sequence of data chunks by placing a chunk on *AD* and pulsing *Dreq*.

The pulse delay can be one clock cycle when all parts use the same clock, because only one function, and hence only one FPGA, is active at any time, so the other FPGAs can respond immediately to an address.

Both of the above communications are composed of just two basic protocol actions: *Send* and *Receive*. *Send* places the receiver's address and then the sequence of data (return address plus input parameters, or output parameters), and *Receive* waits for its address and then receives and assembles the data. Figure 5 shows pseudocode suitable for creating C, VHDL or Verilog specifications of the send and receive protocols.

Compared with the previous cut-edges I/O approach

```
char msg[MSG_SIZE];
long pubkey_d;
long pubkey_n;

void XmitMsg ()
{
   pubkey_d = CL_RecSrvLongPCS(c1);
   pubkey_n = CL_RecSrvLongPCS(c1);

   for (num=0; num < MSG_SIZE; num++)
   {
      msg_enc = EncodeMsg(msg[num]);
      CL_SendSrvLongPCS(c1, msg_encoded);
   }
}

long EncodeMsg(char msg_item)
{
   return ModExp(M, pukey_d, pubkey_n);
}

long ModExp(long a, long b, long n)
{
   long c,d;  int i;
   c = 0;  d = 1;
   for (i=sizeof(long)*8)-1; i>=0; i--)
   {
      c = 2 * c;
      d = (d*d)%n;
      if (Testbit(b,i))
      {
         c = c + 1;
         d = (d*a) % n
      }
   }
   return d;
}
```

Fig. 6: Example: unpartitioned specification.

to functional partitioning, we note several differences. First, rather than having distinct control signals for each data item transferred among FPGA's, we encode the signals into addresses, resulting in fewer wires. Second, these addresses are multiplexed with data over the same wires, resulting in even fewer wires but also slightly slower performance. Of course, we could modify the FunctionBus to use distinct address and data lines. Third, data transfers corresponding to distinct function calls are multiplexed over the same bus, which does not degrade performance since such transfers were sequential anyways.

4 Example

In this section, we walk through an example of a functionally-partitioned system using the FunctionBus. We use an example involving RSA encryption [28]. In this example, a portable device permits data entry, storing the data in an internal memory. The device can be connected to a PC's serial port for transfer of the data to the PC. The data is transported using RSA encryption technology. The PC first transmits two public keys to the device, which then encodes each data item before sending it to the PC. The encrypted data on the PC can only be read by a user having the private keys necessary for decryption.

The portion of the portable device responsible for data encryption and PC communication is shown in Figure 6. The device uses *XmitMsg* to transmit the

```
char msg[MSG_SIZE];                                   FPGA1

void XmitMsg ()
{
   FB_Init();
   key = CL_RecSrvLongPCS(c1);
   FB_SendLong(FB_pubkey_d, key);
   key = CL_RecSrvLongPCS(c1);
   FB_SendLong(FB_pubkey_n, key);
   for (num=0; num < MSG_SIZE; num++)
   {
      FB_SendChar(FB_EncodeMsg, msg[num]);
      msg_enc = FB_RecLong(FB_XmitMsg);
      CL_SendSrvLongPCS(c1, msg_enc);
   }
}
```

```
long pubkey_d;                                        FPGA2
long pubkey_n;

void main2()
{
   FB_Init();
   pubkey_d = FB_RecLong(FB_pubkey_d);
   pubkey_n = FB_RecLong(FB_pubkey_n);
   while(1)
   {
      msg_raw = FB_RecChar(FB_EncodeMsg);
      msg_enc = EncodeMsg(msg_raw);
      FB_SendLong(FB_XmitMsg, msg_enc);
   }
}
long EncodeMsg(char msg_item) ...
long ModExp(long a, long b, long n) ...
```

Fig. 7: Example after partitioning using FunctionBus.

data to the PC. This function first receives the two keys over the PC serial port using a communication library function (having a *CL_ prefix*) describing the protocol; we omit this function's details. The keys are stored in two global variables. Next, *XmitMsg* steps through each data item, encodes the data by calling another function *EncodeMsg*, and then sends the encoded data to the PC. In turn, *EncodeMsg* calls *ModExp*, which uses the two keys to encrypt the data.

An automated functional partitioner (as described in [21]) might find that *ModExp* uses too much hardware to coexist on the same FPGA as the other functions, since it uses a multiplier and a divider. Thus, *ModExp* may be partitioned to a second FPGA. Because functional partitioning is usually driven not only by capacity constraints but also by execution-time constraints, such partitioning will try to minimize time-consuming inter-FPGA communication. To avoid transmitting the two public keys to the second FPGA for every data item, the two keys' global variables may also be moved to the

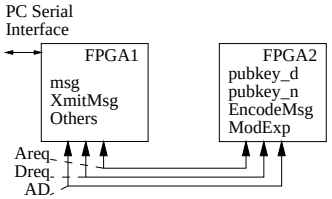

Fig. 8: Example's FunctionBus architecture.

```
void FB_SendChar(
     byte fb_addr;
     char fb_data )
{
   // Send the address
   FB_AD      = fb_addr;
   FB_Areq    = 1;
   FB_Delay();
   FB_Areq    = 0;

   // Send the data
   FB_AD      = fb_data;
   FB_Dreq    = 1;
   FB_Delay();
   FB_Dreq    = 0;

   // Release the bus
   FB_AD = Z;
   FB_Areq = FB_Dreq = Z;
}
```

```
char FB_RecChar(
   byte fb_addr )
{
   char fb_data;
   // Wait for the address
   while ( ! (  FB_Areq &&
                FB_AD==fb_addr
         ) );

   // Receive the data
   while (! FB_Dreq);
   fb_data = FB_AD;
   while (FB_Dreq);

   // Return the data
   return(fb_data);
}
```

Fig. 9: FunctionBus character-data transfer routines.

```
void FB_SendLong(
     byte fb_addr;
     long fb_data )
{
   // Send the address
   FB_AD      = fb_addr;
   FB_Areq    = 1;
   FB_Delay();
   FB_Areq    = 0;

   // Send the data
   for (i=0; i<4; i++)
   {
      FB_AD =fb_data>>(8i)
      FB_Dreq    = 1;
      FB_Delay();
      FB_Dreq    = 0;
   }

   // Release the bus
   FB_AD = Z;
   FB_Areq = FB_Dreq = Z;
}
```

```
char FB_RecLong(
   byte fb_addr )
{
   long fb_data;
   // Wait for the address
   while ( ! (  FB_Areq &&
                FB_AD==fb_addr
         ) );

   // Receive the data
   for (i=0; i<4; i++)
   {
      while (! FB_Dreq);
      fb_data
        = (fb_data<<8)|FB_AD;
      while (FB_Dreq);
   }

   // Return the data
   return(fb_data);
}
```

Fig. 10: FunctionBus long-data transfer routines.

second FPGA, along with the *EncodeMsg* function, as shown in Figure 7.

The above functional partition, which satisfies capacity constraints while minimizing communication and hence maximizing performance, has created four inter-FPGA communications: writing the two keys, and calling and returning from *EncodeMsg*. We use the FunctionBus to implement those communications. Each key is assigned an address, shown in Figure 7 as *FB_pubkey_d* and *FB_pubkey_n*, which represent two-bit values. *EncodeMsg* is also assigned an address *FB_EncodeMsg*. Because *XmitMsg* must receive output parameters from *EncodeMsg*, it too requires an address, *FB_XmitMsg*. After receiving a key, *XmitMsg* calls a FunctionBus routine *FB_SendLong* to send the key to a global variable on the second FPGA. *XmitMsg* also uses another FunctionBus routine *FB_SendChar* to send each data item to *EncodeMsg*. Finally, it uses *FB_RecLong* to receive the encoded data. Complementary routines are found on the second FPGA. The two-FPGA architecture is shown in Figure 8.

The FunctionBus routines for a size of 8 are shown in Figure 9 and Figure 10, written in C for conciseness. Each send routine requires two parameters: the address

of the function to be called, and the data to be sent to that function. Each receive routine requires the address of the receiver function, *not* the address of the function sending the data. These routines have been optimized for the data types being transferred, unlike the general routine templates shown earlier in Figure 5. Character data is sent in one chunk, while long data (four bytes) is sent in four chunks. (For conciseness, we use shifts to indicate which byte is being transferred or captured, though for synthesis we would access the bits directly).

The activity over a FunctionBus of size 8 is shown in Figure 11. Each long is sent as four chunks of 8 bits each. Note that the address for each long is only sent once. Suppose we change the FunctionBus size to 16. The character-data transfer routines would remain mostly the same. However, the long-data transfer routines would now only loop twice instead of four times, transferring two chunks of 16 bits each, as illustrated in Figure 12, resulting in a performance improvement at the expense of 8 I/O pins.

Let us now consider the range of possible tradeoffs of I/O and performance. First, we derive an execution-time equation for *XmitMsg* in Figure 7 as follows:

$$75+2*FB_{long}+(512\times(FB_{char}+128+FB_{long}+35)) \quad (1)$$

$$or: \quad 83531+514*FB_{long}+512*FB_{char} \quad (2)$$

In other words, *XmitMsg* first spends 75 cycles initializing the FunctionBus (assumed to require 5 cycles) and receiving the two keys from the PC (assumed to require 35 cycles each). It then spends $2*FB_{long}$ cycles sending those keys to *FPGA2*. Then, *XmitMsg* loops 512 times (assuming *MSG_SIZE* is 512) each time spending FB_{char} cycles sending a character to *FPGA2*, 128 cycles waiting while the encoding takes place on *FPGA2* (assumed), FB_{long} cycles receiving the encoded long data, and another 35 cycles sending the encoded data to the PC. FB_{char} and FB_{long} are times spent transferring character and long data, plus the address, over the FunctionBus. These times will vary depending on the FunctionBus size, as shown in Figure 13 under columns *Char* and *Long*. When the FunctionBus size is 1, FB_{char} is 10 because we transmit 2 address bits and 8 data bits serially; likewise, FB_{long} is $2+32=34$ cycles, yielding an $XmitMsg$ with an execution time of $83531+514*34+512*10=106127$ cycles. Instead, when the FunctionBus size is 4, FB_{char} is 3 because we transmit 2 address bits in one chunk, followed by a chunk of 4 data bits and then another chunk of 4 data bits; likewise, FB_{long} is $1+32/4=9$, so execution time is then $83531+514*9+512*3=89693$ cycles. In this example, the largest practical FunctionBus size is 32, since the largest data item transferred over the FunctionBus in the example is 32-bit long data.

Looking at Equation 2, we see that we can compute a behavior B's execution time by separating that time into non-FunctionBus time and FunctionBus time, as follows:

Size	Char	Long	Time
1	10	34	106127
2	5	17	94829
3	4	12	91747
4	3	9	89693
5	3	8	89179
6	3	7	88665
7	3	6	88151
8	2	5	87125
9	2	5	87125
10	2	5	87125
11	2	4	86611
12	2	4	86611
13	2	4	86611
14	2	4	86611
15	2	4	86611
16	2	3	86097

Size	Char	Long	Time
17	10	3	86097
18	2	3	86097
19	2	3	86097
20	2	3	86097
21	2	3	86097
22	2	3	86097
23	2	3	86097
24	2	3	86097
25	2	3	86097
26	2	3	86097
27	2	3	86097
28	2	3	86097
29	2	3	86097
30	2	3	86097
31	2	3	86097
32	2	2	85583

Fig. 13: Example's I/O versus performance tradeoffs obtained by varying the FunctionBus size.

$$\begin{aligned} B.et &= nonFB.time \quad (3)\\ &+ \sum_{fbtrans\in B} (FB.delay \times fbtrans.freq \times \\ &\quad \lceil \frac{FB.addrbits}{FB.size} \rceil + \quad \lceil \frac{fbtrans.bits}{FB.size} \rceil) \end{aligned}$$

where $B.et$ is B's execution time, *nonFB.time* is the execution time of B (including its called functions) excluding FunctionBus transfers, *fbtrans* is a FunctionBus transfer involving B or one of its called functions, *FB.delay* is the time for which data must be held valid on the FunctionBus (usually just one clock cycle, but possibly longer when dealing with slower devices like microcontrollers), *fbtrans.freq* is the number of times the transfer occurs, *FB.addrbits* is the number of address bits (which is usually log of the number of possible inter-FPGA transfers), *FB.size* is the FunctionBus size, and *fbtrans.bits* is the number of data bits of the current transfer.

5 Experiments

We performed several experiments to demonstrate the ability to easily tradeoff performance and I/O when using the FunctionBus. We used four examples: *ether*, an Ethernet coprocessor; *fuzzy*, a fuzzy-logic controller; *itv*, an interactive TV processor; and *mwt*, a microwave transmitter controller. We first applied an automatic functional partitioner [21] to partition among two Xilinx XC4000 FPGA's, using simulated annealing and a cost function seeking to minimize communication while meeting FPGA size and I/O constraints.

Given this partition for each example, we then varied the FunctionBus size from 1 to the maximum bits transferred during any function call. For each size, we recomputed the execution time. Results are shown in Figure 14. Each point on the X-axis has two numbers. The bottom number represents the FunctionBus size.

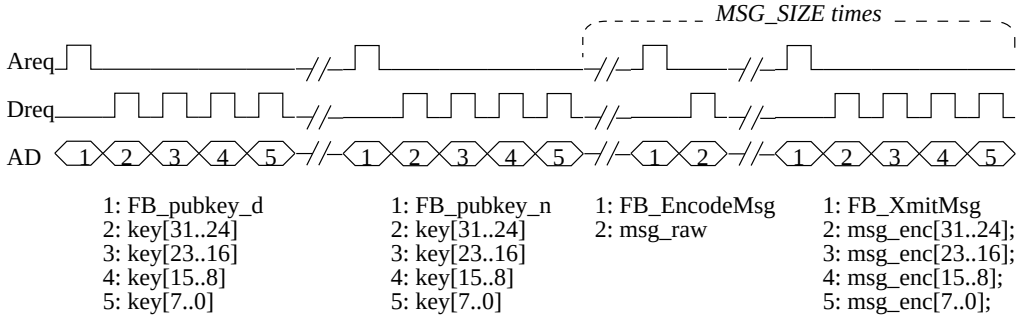

Fig. 11: 8-bit FunctionBus activity.

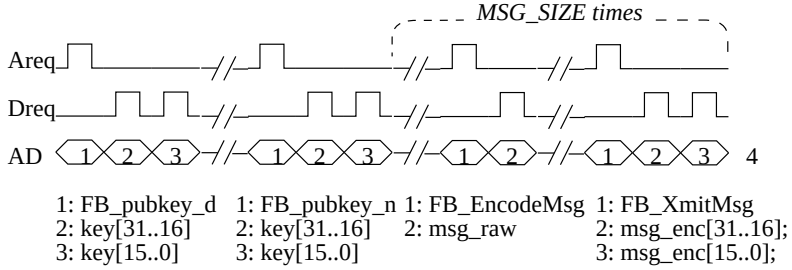

Fig. 12: 16-bit FunctionBus activity.

The top number represents the maximum I/O required on either FPGA. All examples were larger than 15,000 gates when implemented – note the extremely low maximum I/O sizes for these examples. The Y-axis represents total execution-time of all constrained behaviors in the system, measured in clock cycles. This time includes all computation and communication.

The curves demonstrate a wide range of performance obtainable by varying the FunctionBus size. Performance varied by a factor of 4 for the *fuzzy* example, indicating that the partitioned system involves much communication; in other words, the second term in Equation 3 was large. On the other hand, performance varied by only 1.5 for the *itv* example, indicating that the partition involved much less communication relative to computation within each FPGA; in other words, the first term in Equation 3 was dominant. In addition, the curves demonstrate that numerous sizes can give the same performance. For example, performance is identical for sizes ranging from 12 to 22 in the *fuzzy* example, so sizes 13 to 22 would never be beneficial. Finally, the curves demonstrate the rapid, non-linear deterioration of performance as we go below a size of 8 towards serial transfer. Such deterioration is due in part to the fact that a significant percentage of transfers are of 8-bit or larger items.

Such data can serve the purpose of allowing a designer to explore various design points, especially when trying to choose a package size or trying to allocate I/O to various types of communication (perhaps over more than one FunctionBus). It can also help us to pick a FunctionBus size for a standard board or a logic emulator, such that we use the fewest I/O that still gives us reasonable performance for a large set of applications.

6 Status and future work

We have been developing an automated functional partitioner since 1991. VHDL algorithmic-level code is parsed into the Specification-Level Intermediate Format (SLIF) [29]. SLIF is then heavily annotated with values indicating hardware size or contributors to hardware size (such as required control steps, functional units, data paths, control lines, etc.), data sizes, frequencies of communication, computation times, and software sizes, with distinct annotation values for each possible implementation technology. Such annotation is obtained by using estimators and profilers developed at UC Irvine [21]. The SLIF is then partitioned among allocated packages, using any of a wide variety of heuristics, including clustering, Kernighan/Lin, and simulated annealing. Work is underway at UC Irvine to automatically generate a refined VHDL description reflecting the chosen functional partition. The previous I/O model was based on a cut-edges model. The FunctionBus model is now being incorporated. The overall system consists of over 100,000 lines of C code, and has been released to numerous companies and universities.

There are many directions in which a FunctionBus approach to functional partitioning can be improved. First, we might select a FunctionBus size simultaneously with partitioning, since these two tasks are interdependent. Second, a FunctionBus approach naturally leads to the idea of developing a power-reduction technique that shuts down the FPGA's that are not actively executing a function, except for the address detection logic on each FPGA, in order to reduce power; we are investigating such techniques. Third, we can develop techniques to interface standard components, such as memories and processors, having fixed buses and protocols.

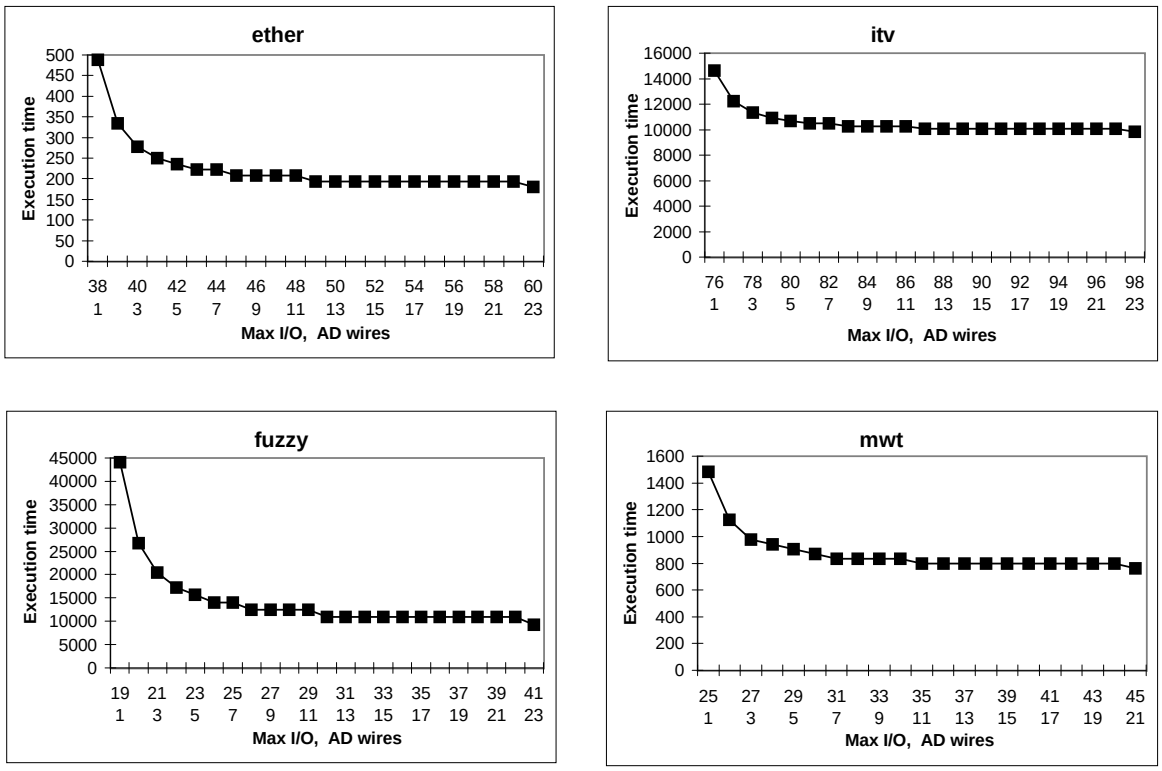

Fig. 14: I/O and performance tradeoff curves.

Fourth, there are likely many optimizations that can be applied to improve performance and hardware size. For example, we can replicate threads so that each has a single return address, eliminating the need to send a return address to a multiply-called function. As another example, we might experiment with different address encoding techniques to see their effects on hardware size. Further examples include cloning multiply-called functions to reduce FunctionBus traffic at the expense of replicated hardware [30], and enclosing external port accesses into function calls to enable us to distribute ports among FPGA's to obtain a balance of I/O. Fifth, we need to extend the approach for multiple processes. One simple method is to use a distinct bus and a distinct processor for each process. More sophisticated methods might share a FunctionBus among multiple processes, possibly requiring arbitration.

7 Conclusions

We have presented an improvement to a new approach to multi-FPGA partitioning. Earlier work demonstrated greatly reduced I/O and other improvements obtained through functional partitioning, while the work in this paper demonstrates techniques for further reducing I/O using the FunctionBus, and for trading off performance for even further I/O reductions. Functional partitioning combined with the FunctionBus therefore comprise a promising solution to multi-FPGA partitioning, particularly in overcoming the I/O limitations of traditional structural partitioning. The FunctionBus approach is applicable not only for partitioning among FPGA's, but also for partitioning among any custom hardware blocks (even within the same package) as well as among hardware and software processors. Therefore, the FunctionBus approach provides a simple yet powerful basis for general hardware and software system partitioning.

References

[1] C. Alpert and A. Kahng, "Recent directions in netlist partitioning," *Integration, The VLSI Journal*, vol. 19, pp. 1–81, 1995.

[2] D. Brasen and G. Saucier, "FPGA partitioning for critical paths," in *Proceedings of the European Design and Test Conference (EDTC)*, pp. 99–103, 1994.

[3] P. Chan, M. Schlag, and J. Zien, "Spectral based multiway FPGA partitioning," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 133–139, 1995.

[4] N. Chou, L. Liu, C. Cheng, W. Dai, and R. Lindelof, "Local ratio cut and set covering partitioning for huge logic emulation systems," in *IEEE Transactions on Computer-Aided Design*, pp. 1085–1092, 1995.

[5] S. Hauck and G. Borriello, "Logic partition orderings for mult-fpga systems," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 32–38, 1995.

[6] D. Huang and A. Kahng, "Multi-way system partitioning into single and multiple type FPGAs," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 140–145, 1995.

[7] R. Kuznar and F. Brglez, "PROP: A recursive paradigm for area-efficient and performance oriented partitioning for large FPGA netlists," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 644–649, 1995.

[8] K. Roy-Neogi and C. Sechen, "Partitioning with performance optimization," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 146–152, 1995.

[9] P. Sawkar and D. Thomas, "Multi-way partitioning for minimum delay for look-up table based FPGAs," in *Proceedings of the Design Automation Conference*, pp. 201–205, 1995.

[10] R. Tessier, J. Babb, M. Dahl, S. Hanono, and A. Agarwal, "The virtual wires emulation system: A gate-efficient asic prototyping environment," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, 1994.

[11] F. Johannes, "Partitioning of VLSI circuits and systems," in *Proceedings of the Design Automation Conference*, 1996.

[12] F. Vahid, T. Le, and Y. Hsu, "A comparison of functional and structural partitioning," in *International Symposium on System Synthesis*, 1996.

[13] E. Lagnese and D. Thomas, "Architectural partitioning for system level synthesis of integrated circuits," *IEEE Transactions on Computer-Aided Design*, vol. 10, pp. 847–860, July 1991.

[14] R. Gupta and G. DeMicheli, "Partitioning of functional models of synchronous digital systems," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 216–219, 1990.

[15] K. Kucukcakar and A. Parker, "CHOP: A constraint-driven system-level partitioner," in *Proceedings of the Design Automation Conference*, pp. 514–519, 1991.

[16] Y. Chen, Y. Hsu, and C. King, "MULTIPAR: Behavioral partition for synthesizing multiprocessor architectures," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 2, pp. 21–32, March 1994.

[17] C. Gebotys, "An optimization approach to the synthesis of multichip architectures," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 2, no. 1, pp. 11–20, 1994.

[18] F. Vahid and D.Gajski, "Specification partitioning for system design," in *Proceedings of the Design Automation Conference*, pp. 219–224, 1992.

[19] R. Gupta and G. DeMicheli, "Hardware-software cosynthesis for digital systems," in *IEEE Design & Test of Computers*, pp. 29–41, October 1993.

[20] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," in *IEEE Design & Test of Computers*, pp. 64–75, December 1994.

[21] D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and design of embedded systems*. New Jersey: Prentice Hall, 1994.

[22] X. Xiong, E. Barros, and W. Rosentiel, "A method for partitioning UNITY language in hardware and software," in *Proceedings of the European Design Automation Conference (EuroDAC)*, 1994.

[23] A. Kalavade and E. Lee, "A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem," in *International Workshop on Hardware-Software Co-Design*, pp. 42–48, 1994.

[24] P. Eles, Z. Peng, and A. Doboli, "VHDL system-level specification and partitioning in a hardware/software co-synthesis environment," in *International Workshop on Hardware-Software Co-Design*, pp. 49–55, 1992.

[25] P. Knudsen and J. Madsen, "PACE: A dynamic programming algorithm for hardware/software partitioning," in *International Workshop on Hardware-Software Co-Design*, pp. 85–92, 1996.

[26] A. Balboni, W. Fornaciari, and D. Sciuto, "Partitioning and exploration strategies in the tosca co-design flow," in *International Workshop on Hardware-Software Co-Design*, pp. 62–69, 1993.

[27] F. Vahid and D. Gajski, "Incremental hardware estimation during hardware/software functional partitioning," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 3, no. 3, pp. 459–464, 1995.

[28] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1989.

[29] F. Vahid and D. Gajski, "SLIF: A specification-level intermediate format for system design," in *Proceedings of the European Design and Test Conference (EDTC)*, pp. 185–189, 1995.

[30] F. Vahid, "Procedure cloning: A transformation for improved system-level functional partitioning," in *Proceedings of the European Design and Test Conference (EDTC)*, p. to appear, 1997.