Port Calling: A Transformation for Reducing I/O during Multi-Package Functional Partitioning

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

Abstract

Partitioning a system among multiple input and output pin (I/O) limited packages is a widely researched and hard to solve problem. We previously described a new approach yielding large improvements, which partitioned functions rather than structure, and which used a single bus for all inter-package data transfer. In this paper, we describe an extension permitting arbitrary distribution of I/O among the packages, and highlight experiments demonstrating even further I/O reductions as well as surprisingly improved performance, with nearly no penalty.

1 Introduction

Systems often must be partitioned among multiple packages. For example, an embedded system might use a software micro-controller package for those functions that need not run very fast or that are likely to change, and a hardware part for functions requiring high-speed execution. The hardware part might in turn consist of a number of FPGA packages, since one FPGA might not have enough gates or I/O to implement all the hardware functions. Even with today's increasing package capacities leading to single-chip solutions for entire systems, such a chip itself may still include one or more processor cores, as well as numerous hardware blocks. Logic emulation is another case requiring system partitioning among numerous FPGA's.

The multi-package partitioning problem has been widely researched, but has been hard to solve well. A good survey of the problem is found in [1]. Much of the research focuses on the fact that multi-package partitioning is I/O driven, meaning the number of I/O pins available on a package is the hardest constraint to satisfy, leading to underutilized gates and hence more packages than gate count alone implies. More packages can lead to larger systems, more power consumption, and reduced performance due to more inter-package signal crossings. Recent work [2] significantly reduced I/O for logic emulation, by time-multiplexing inter-package signal transfers over physical I/O, using multiple clocks and distributed controllers. Like most previous approaches, this approach performs structural partitioning, in which one first designs a system's structure (register-transfer components and gates), and then partitions the structure among packages.

We demonstrated significant advantages of a new and different approach, functional partitioning, over structural partitioning [3]. In functional partitioning, we take advantage of the recent trend of designers first specifying a system's functionality using a program-like language, such as VHDL, Verilog, or C. We partition this program among packages, using SpecSyn estimators developed by UC Irvine [4] to guide the partitioning process. After partitioning, structure may be designed separately for each package. The advantages included greatly reduced I/O (often 50%), improved performance, and reduced synthesis runtimes (often by an order of magnitude). Such advantages were predicted by some researchers for many years [5, 6, 7, 8, 9]. In addition, since programs are being partitioned rather than gates, the approach supports hardware-software partitioning, unlike structural partitioning which is for hardware only. Many hardware-software functional partitioning approaches have recently been proposed [10, 11, 12, 13, 14, 15, 16, 17].

In [18], we described an approach to implementing inter-package data transfers, using the FunctionBus. In this paper, we utilize the FunctionBus feature of fixed inter-package I/O size to develop an improved approach. This new approach uses a transformation we refer to as *port calling*, which enables us to almost arbitrarily distribute external port I/O among the packages, permitting better balancing of I/O, and often even improving performance and reducing total I/O, with nearly no penalty. We first provide background on the representation used and on the FunctionBus, introduce the port-calling transformation, describe the port-calling functions that are used, and summarize

1080-1820/97 $10.00 © 1997 IEEE

```
void F()
{
  a = Q / 32;
  G(a, b);
  H(b);
}

void H(int b)
{
  S = b * 100;
  P = d * 32;
}

void G(int a, int& b)
{
  b = 1;
  for (int i=0; i<100; i++)
  {
    b = b + a*R;
    H(b);
  }
}
```

Fig. 1: Example specification.

experiments demonstrating the significant improvements obtained.

2 Background

In this section, we briefly describe the SLIF representation and the FunctionBus, which are used by our technique. In our approach, a program (VHDL or C) is first translated to SLIF (System-Level Intermediate Format [19]). The SLIF, similar to a call-graph used in software profiling, is a directed graph, where each node represents a coarse-grained function or variable, and each edge represents an access by a function to another function or variable. The edge direction indicates the accessor and accessee, not the direction of data flow, which can occur in either or both directions over the same edge. For example, Figure 1 shows a (trivial) functional specification program written in C, consisting of three functions *F, G* and *H*, which call each other, and which access four external 16-bit ports *P, Q, R* and *S*. Figure 5(a) contains the SLIF for this program. Each function becomes a node, and each function call and port access becomes one edge.

During a pre-estimation phase, each SLIF edge is annotated with estimation and profiling information, such as the number of data bits for each transfer (function parameters or variable data) and the access frequency. Each SLIF node is also annotated with information, such as the execution time and size on various package types. Only external-port bit-size annotations are shown in the example.

During partitioning, online-estimation combines these annotations, using non-trivial equations, to obtain fast yet adequately accurate estimates of design metrics, such as size (gates or bytes), I/O, and execution time. For example, the execution-time equation considers the time for a procedure to execute on its current component, plus the time to transfer data to and from any accessed procedures and variables (which depends heavily on the locality of those objects in the current partition), plus the execution time for accessed procedures. The cost function guiding partitioning is a weighted-sum of normalized constraint violations. The partitioning itself consists of moving SLIF nodes among packages, where each package implements a custom or standard processor, and is thus performed at the granularity of procedures and large variables (though procedures can be inlined or exlined to adjust the granularity). It is achieved by using any of several heuristics (such as simulated annealing, a modified Kernighan/Lin heuristic, clustering or greedy improvement) or by manually moving objects. For further details on the partitioning approach, the reader is referred to [19, 4, 12, 20].

In addition to using the SLIF representation, our approach uses the FunctionBus for inter-part communication [18]. In contrast, many earlier multi-package functional partitioning approaches used a cut-edges approach to I/O implementation. In such an approach, a graph's nodes, representing functions of varying granularities, are partitioned among parts. When an edge, which represents data, is cut (i.e., crosses between two parts), a unique set of I/O pins is used to transfer the data. However, observing that a large process often must be partitioned among parts, and that a process' functions can or do execute sequentially, we used a data transfer approach using what was called a FunctionBus. The bus consists of one address valid line, one data valid line, and a set of multiplexed address/data lines of a designer-chosen width, as shown in Figure 2. The bus works as follows. Each function is assigned a unique address. Whenever a function *A* calls a function *B* on another part, *A* first places *B*'s address on the bus and asserts the address line, as shown in the timing diagram of Figure 3(a). If *B* is called by more than one function, then *A* must also send its own address as a return address. *A* then places any input parameters that must be passed to *B* on the bus, in chunks if the parameter size exceeds the bus size, asserting the data valid line for each chunk, and then *A* suspends. *B* detects its address, receives the parameter data, executes (perhaps calling other functions), and then returns to *A* by sending *A*'s address over the bus followed by any output parameters. *A* detects its address, receives any parameter data, and resumes execution. Figure 4 provides example send and receive C routines for transferring long integer data over a FunctionBus with 8 address/data lines.

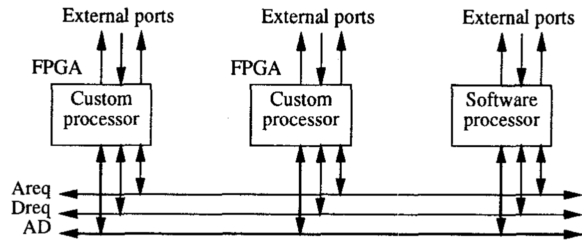
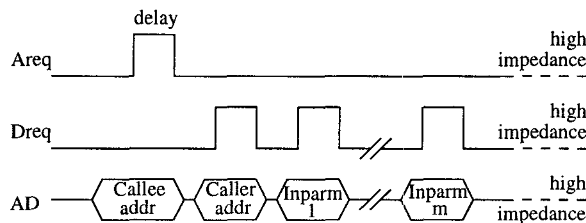


Fig. 2: FunctionBus architecture.



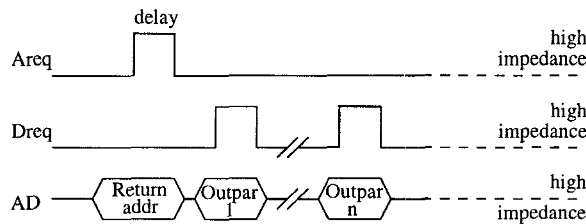


Fig. 3: Timing diagrams: (a) function call, (b) function return.

We demonstrated that the FunctionBus yielded even further reductions in I/O over the already large reductions of functional partitioning over structural partitioning. The multiple functions of a process on a given part require only one process, which looks for an address of any of its functions, and then branches to the appropriate one. In addition, the FunctionBus could be used for either hardware partitioning, or for hardware/software partitioning where the software component had parallel I/O (e.g., a micro-controller having several N-bit ports to which it can read or write, independent of memory accesses).

3 Port-calling transformation

In a FunctionBus approach, the number of I/O pins required for communication among packages is fixed (and typically small, like 18 or 34 pins). Hence, the only variation in a package's I/O comes from the I/O connected to external ports, needed by the functions on that package. The port-calling transformation will allow us to redistribute such ex-

```
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);
}
```

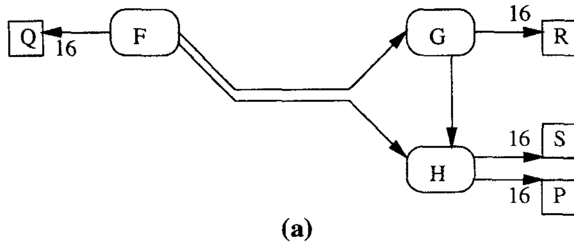
Fig. 4: FunctionBus long-data transfer routines.

ternal port I/O to packages other than the accessing function's package.

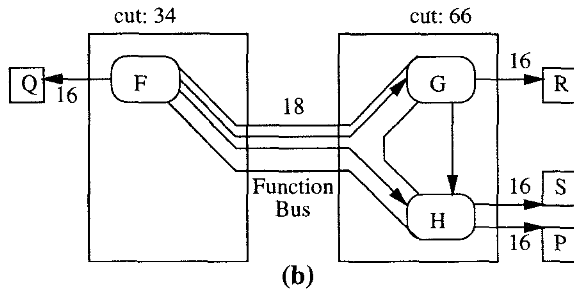
The transformation consists of introducing a new function, called a port-call function, in between the original accessor function and the port itself. This function may, upon being called by the accessor function, read the port or write the port (as will be discussed further in the next section) on behalf of that function. Thus, from the accessor's perspective, accessing the port has been replaced by a function call.

In SLIF, a port-call function is represented as any other function, i.e., a node. This node can be partitioned among packages just like any other function. If this node is separated from its accessor function, any data transfer will take place over the existing FunctionBus; since the I/O for the FunctionBus already exists and is fixed, such data transfer does not require any additional inter-package I/O. Since a port-call node has extremely simple contents and hence when implemented will not contribute noticeably to a package's size, it can be partitioned to nearly any package. Hence, we see that introducing port-call nodes, in conjunction with the FunctionBus, yields the ability to freely distribute I/O among packages, at the possible expense of a few extra clocks cycles required to pass the data along from the port-call function to the accessor function.

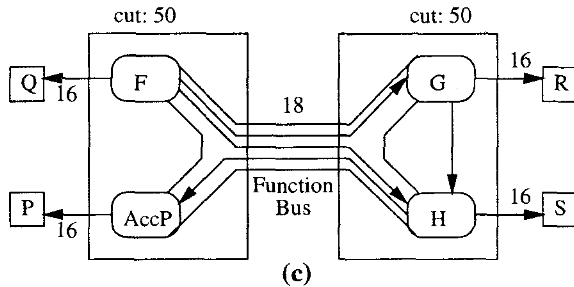
For example, Figure 5(b) provides a simple example of a partitioned SLIF with several port accesses. Assuming a FunctionBus size of 18, the total I/O for the part on the right would be 66, since 48 I/O pins are required by the part's functions that



(a)



(b)



(c)

Fig. 5: Port calling example: (a) Original SLIF, (b) Partitioned SLIF using FunctionBus, (c) After port-calling transformation.

access three 16-bit external ports. However, notice that the left part only requires 34 I/O. There is a significant imbalance in I/O. Though the total I/O is 100, we would not be able to use two 50-pin packages, but instead would have to use a more expensive and larger package on the right with at least 66 pins.

If, however, we insert a port-call function *AccP* in between function *H* and port *P*, as shown in Figure 5(c), we can repartition the SLIF to achieve a perfect balance of 50 I/O for each part. *H* now must call *AccP*, which in turn accesses *P* directly. The write data is transferred from *H* to *AccP* over the FunctionBus.

We must decide which port accesses should have port-call nodes introduced to achieve improvements like that illustrated in Figure 5. Predicting those accesses that, when replaced by port-call nodes, would

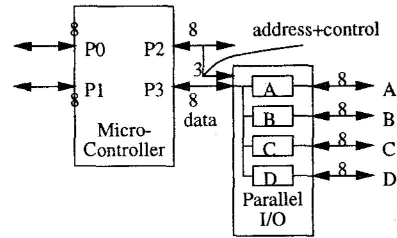


Fig. 6: Extended parallel I/O.

lead to such improvements, is a difficult task. We note, though, that the number of external ports is such that we can introduce a port-call node for every port access without much problem. Thus, our approach is to:

1. *Transform* the SLIF by introducing port-call nodes for every port access,
2. *Partition* the SLIF using existing heuristics, and
3. *Untransform* the SLIF by eliminating each port-call node that appears on the same part as its accessor.

The *untransform* step is necessary to eliminate unnecessary port-call nodes, so that only those nodes needed to distribute I/O to another part remain.

Note that port-calling is a generalization of the commonly used technique of extended parallel I/O. For example, consider Figure 6. A micro-controller with limited ports must interface to four 8-bit external ports A, B, C and D, using just one 8-bit port P3 and a few bits of P2. A common solution to this problem is to introduce a parallel I/O (PIO) chip, which multiplexes the four external ports over the single 8-bit data port, or demultiplexes the 8-bit data port to the four external ports, depending on its input address and control lines. The bus between the micro-controller and PIO chip is akin to the FunctionBus, and the control internal to the PIO is essentially equivalent to port-call functionality. Port-calling is more general since we can move the functionality to chips other than just PIO chips, such as to an FPGA or even another micro-controller.

4 Port-call functions

After the SLIF is partitioned, new program-like descriptions must be generated for each package. For a process *A* from the original specification, each

```
datatype PortCallRead()
{
   return(P);
}
void PortCallWrite(datatype d)
{
   P = d;
}
datatype PortCallReadOrWrite(datatype d, bit read)
{
  if (read)
    return(P)
  else
    {P = d; return(0);}
}
```

Fig. 7: Port-call functions.

package will have its own process for A. This process will consist of detecting one of its function's addresses on the FunctionBus, capturing any input parameters from the bus, calling the function, and then returning by placing a return address and any output parameters on the bus. A call to a function on another part is replaced by FunctionBus call and return routines, as shown in Figure 4. Port-call functions are treated as any other function, requiring no special treatment. We thus need only describe the contents of such functions here; partitioning and subsequent FunctionBus routine insertion will take care of the communication between the port-call function and the accessor function.

The port-call functions for accessors that read, write, or both read and write a port are shown in Figure 7. Note that each is trivial to implement, so could be moved freely among parts.

5 Experiments

We implemented SLIF transformations to insert port-call functions for every port and to delete port-call functions accessed only by functions on their own parts. We incorporated these transformations with existing tools that convert a VHDL specification to SLIF (using SpecSyn [12] estimators to annotate the SLIF with size, I/O, and execution information), and that partition the SLIF using standard heuristics (we used simulated annealing here) making use of the FunctionBus. We then applied two-way partitioning with and without port-calling on five examples: an answering machine controller *ans*, and Ethernet coprocessor *ether*, an fuzzy-logic controller *fuzzy*, an interactive TV processor *itv*, and a microwave transmitter controller *mwt*, each consisting of a few hundred lines of VHDL algorithmic code. Partitioning was achieved using simulated annealing, with a cost function that sought to minimize I/O and execution time while balancing part sizes. Details of the partitioning and estimation system are extensive and have been described elsewhere [12, 4].

Example	Without Port Calling					With Port Calling				
	Size1	Size2	IO1	IO2	Time	Size1	Size2	IO1	IO2	Time
ans	6140	6491	85	98	44	6164	6494	32	35	44
ether	13326	11343	39	48	194	13478	11202	38	37	194
fuzzy	51899	59802	26	34	10888	58511	53193	34	26	8356
itv	51649	101483	91	61	10049	53288	99872	70	57	9653
mwt	4511	5447	40	32	799	4877	5095	35	23	791

Fig. 8: Partitioning without and with port calling.

Results are summarized in Figure 8. The table shows the size in gates for each part, the I/O for each part, and the execution time of the example, including communication time over the FunctionBus. Port calling yielded significant improvements. Note that not only is the maximum required I/O usually reduced, but that in most cases, total I/O is also reduced and performance is actually improved; these improvements come at the cost of negligible total size increase in some cases.

The improvements can be better seen in Figure 9. The first chart shows the maximum I/O required on either part without and with port calling. Note the reductions, especially in the case of *ans*. The second chart shows the reductions in total I/O, which is the sum of the I/O of both parts. The third chart shows the reductions in execution time. Note that we developed port-calling in order to achieve improvements in the maximum I/O metric; we would have been satisfied with the same total I/O and a slight penalty in performance (due to extra cycles for transferring port data over the FunctionBus). However, those other metrics were actually *improved*. Such improvement can be explained by noting that the maximum I/O is a difficult metric to satisfy, and it can dominate the cost function and hence steer the partitioning heuristics. By introducing port-calling, which allows for arbitrary I/O distribution, this pressure is greatly relieved, and hence the cost function is driven to a greater extent by the other metrics, so the partitioning heuristics can seek to improve those metrics; hence, we see substantial improvements in those metrics.

Future work includes investigating the reduction in power that can be obtained using a FunctionBus-based approach to functional partitioning, developing heuristics for partitioning, and developing libraries for communication among the various hardware and software parts after partitioning.

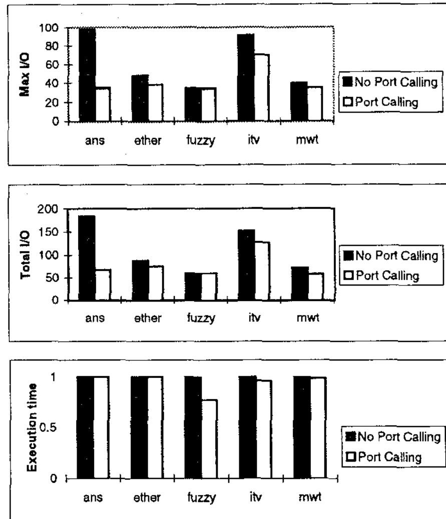


Fig. 9: Partitioning without and with port calling.

6 Conclusions

We have introduced an approach allowing for nearly arbitrary distribution of I/O among packages during partitioning. The approach extends functional partitioning with a port-calling transformation, using the FunctionBus for inter-package communication. The ability to arbitrarily distribute I/O among packages leads to smaller maximum I/O requirements and hence can reduce package cost or quantity. In addition, by easing this previously burdensome metric, partitioning heuristics can focus on other metrics, like performance and total I/O, and hence we see even further improvements in those metrics. Such improvements are important for multi-package partitioning, as well as for partitioning among blocks within a single large ASIC, which may become necessary to perform hardware-software partitioning or to reduce design complexity, routing complexity, power, and synthesis runtimes. Thus, port-calling in conjunction with the FunctionBus can significantly improve increasingly-important functional partitioning environments.

References

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

[2] 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.

[3] F. Vahid, T. Le, and Y. Hsu, "A comparison of functional and structural partitioning," in *International Symposium on System Synthesis*, pp. 121–126, 1996.

[4] D. Gajski, F. Vahid, S. Narayan, and J. Gong, "Specsyn: An environment supporting the specify-explore-refine paradigm for hardware/software system design," *IEEE Transactions on Very Large Scale Integration Systems*, p. to appear, 1997.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

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

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] F. Vahid, "I/O and performance tradeoffs with the functionbus during multi-FPGA partitioning," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 27–34, 1997.

[19] 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.

[20] 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.