Short Papers

Techniques for Minimizing and Balancing I/O During Functional Partitioning

Frank Vahid

***Abstract*—Recent work has demonstrated numerous benefits of functionally partitioning a behavioral process into mutually exclusive subprocesses before synthesizing each process into a custom digital-hardware processor. A key problem during partitioning is minimizing the input/output (I/O) pins or wires between processors. The traditional structural partitioning approach is strongly restricted by such I/O. We previously showed that the new approach of functional partitioning eases this restriction. We now demonstrate a further relaxation of the I/O restriction by introducing the FunctionBus interprocessor bus and the port-calling functional transformation. The FunctionBus allows choice of any size for internal I/O by trading off I/O size for performance, while port calling allows distribution of external I/O almost arbitrarily among modules. We describe experiments showing large I/O reductions through these techniques, with only small performance penalties.**

***Index Terms*—Communication synthesis, embedded systems, hardware/software codesign, high-level synthesis, partitioning, power minimization, system on a chip, very-large-scale integration (VLSI).**

I. Introduction

Synthesis automatically converts a behavioral process into a customized digital-hardware processor. However, a large behavioral process may pose several problems for synthesis. First, synthesis runtime may last tens of hours or more. Second, the resulting processor may have high power consumption. Third, the resulting processor's size may exceed package [e.g., field-programmable gate array (FPGA)] constraints or may yield a module too large for a physical-design tool to handle well. Each of these problems has previously been addressed by different techniques. For example, synthesis runtime has been addressed by developing heuristics that trade off runtime with quality [1], package/module size constraints through structural partitioning of the processor [2] and time-multiplexing input/output (I/O) signals over wires [3], and power consumption through isolation of processor subcircuits to avoid unnecessary signal switching [4].

We previously hypothesized that functional partitioning could address many such problems simultaneously. In functional partitioning, we first divide a large process into smaller mutually exclusive subprocesses, then synthesize a subprocessor for each subprocess, resulting in communicating mutually exclusive subprocessors, as illustrated in Fig. 1(b). In contrast, in structural partitioning, one first synthesizes a processor and then partitions that processor's components, as shown in Fig. 1(a). While functional partitioning has been done manually in the past, automated approaches only recently became possible because of the now common practice of describing a system's functionality in a machine-readable language like VHDL or C++. The hypothesis was that such partitioning would make the jobs of synthesis, packaging, physical design, and debugging much easier, i.e., several smaller processes would be easier to work with than one large process.

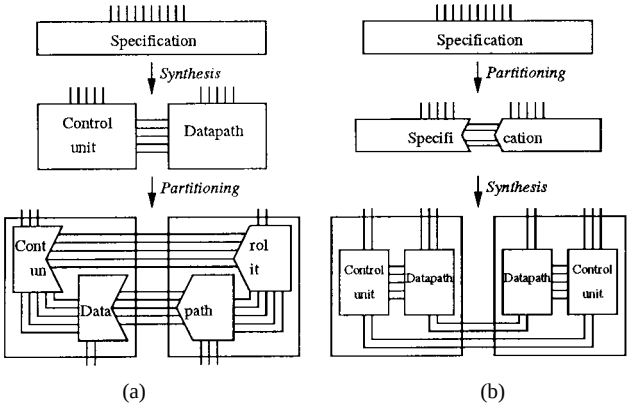

Fig. 1. Partitioning approaches: (a) structural and (b) functional.

Recent experiments have verified this hypothesis. *Synthesis runtime* was reduced by 85% for numerous examples (reducing full-day jobs into less than an hour) [5]. The reductions occurred because synthesis uses nonlinear (e.g., quadratic) heuristics, so the sum of runtimes for synthesizing the parts is less than that for synthesizing the whole. Other experiments showed that *power consumption* was reduced by an average of about 50% [6] because each operation executes on a smaller subprocessor having less switching activity than one large processor, while all other subprocessors are idle and hence naturally isolated from switching activity. Note that no modifications of the synthesis tool are required to obtain such power reduction, since the partitioning is done before synthesis. Recent results have shown that functional partitioning can be followed by existing lower level power reduction techniques (such as latching of functional-unit inputs and clock gating) to achieve even greater reductions. Also, experiments showed that package/module *size constraints* were much more easily met than when using structural partitioning. This is because structural partitioning is typically I/O limited [2], since structural components are highly interconnected and thus any partitioning results in many wires crossing between parts. On the other hand, functional partitioning of a large behavioral process, being a temporal rather than a spatial partitioning, can usually find a partitioning with only a small amount of data shared between parts. Reductions in maximum I/O per part ranged 33–53%, and reductions in total I/O ranged 27–67%, resulting in far fewer required parts to implement a system. Other advantages are also possible; for example, we have found performance improvements due to the smaller subprocessors' having shorter critical paths and hence faster clock periods [5]. Some have also observed that physical design problems, such as clock skew and the high cost of wires in deep submicrometer technologies, could be addressed using functional partitioning [7]. The main drawback of functional partitioning is an increase (typically 20% in our examples) in gates because of less component sharing, but as chip capacities have continued to grow exponentially and hence gates have become cheaper, this increase is not a problem in many if not most applications. Also, if an example does not have a natural temporal partitioning, then communication between subprocessors may be heavy, resulting in a performance

Manuscript received March 30, 1998; revised August 21, 1998. This paper was recommended by Associate Editor G. Borriello.

The author is with the Department of Computer Science and Engineering, University of California, Riverside, CA 92521 USA (e-mail: vahid@cs.ucr.edu).

Publisher Item Identifier S 0278-0070(99)00808-8.

penalty.

The large-process partitioning problem we address differs from the more widely studied multiple-process partitioning problem, in which numerous processes are partitioned and scheduled among software and/or custom hardware concurrent processors [8]–[14]. The large-process problem we address focuses on behavior that is inherently sequential rather than parallel, and focuses on implementation issues like synthesis time, power consumption, and I/O; parallelism may be a goal, but it is not the main one. We have found such inherently sequential behavior in large processes to be quite common. For example, a process representing a direct memory access controller may possess ten different modes, each representing a configuration mode or a different type of block transfer mode, and each having tens of states—the controller is only in one of these modes at any given time. Our solution to such a large-process partitioning problem uses subprocessors that may be mutually exclusive, in stark contrast to the concurrently executing processors used in the multiple-process problem. We point out that synthesis-tool vendors typically recommend that such large-process partitioning be done manually before running synthesis (e.g., no process should have more than X states), so our work can be seen as formalizing and automating such manual partitioning.

However, note that the ability to execute operations in parallel within each subprocessor is preserved. This is important since the speed gained through such operation-level parallelism is likely a main reason, along with reduced power consumption and lower high-volume costs, for using custom hardware rather than a software processor despite the lack of process-level parallelism. One should note that a single custom-hardware processor may have hundreds or thousands of mutually exclusive states, so when combined with the fact that gates are becoming cheap, it makes sense to examine the benefits of partitioning a single processor into exclusive subprocessors.

Previous work in single-process partitioning has evolved from fine-grained arithmetic-operation-level approaches to our coarser grained procedure-level approach, in order to handle increasingly larger processes. The early approaches in Yorktown Silicon Compiler (YSC) and Aparty [15], [16] focused on the logic and arithmetic-operation levels, with goals including improved synthesis and physical design. Vulcan [17] also partitioned arithmetic operations, but with the goal of satisfying packaging constraints and extracting some parallelism. Other arithmetic-operation-level approaches with the goal of multichip partitioning combined with the behavioral synthesis tasks of scheduling and component allocation [18]–[20]. Our early work shared the multichip partitioning goal but focused at the higher level granularity of procedures rather than arithmetic operations [21]. Since then, we have proposed numerous heuristics and estimation models for such partitioning [10].[1]

Regardless of whether we perform single-process partitioning to improve I/O, synthesis time, power, or physical design, we want to minimize each subprocessor's I/O. Such I/O exists for one of two reasons: to connect with an input/output external to the system (*external I/O*) or to connect one subprocessor with another (*internal I/O*). In the past, minimizing internal I/O (often called the cut size) dominated research. While functional partitioning can reduce internal I/O as described above, we describe in this paper two additional techniques that further improve I/O. First, the *FunctionBus* enables us to trade off internal I/O size with performance. Second, the *port-calling* transformation, used in conjunction with the FunctionBus, enables us to distribute external I/O almost arbitrarily among subprocessors. With these two techniques, the I/O satisfaction problem is greatly relaxed. We demonstrate through experiments that this relaxation allows partitioning heuristics to better optimize system performance.

[1] See also http://www.cs.ucr.edu/~vahid.

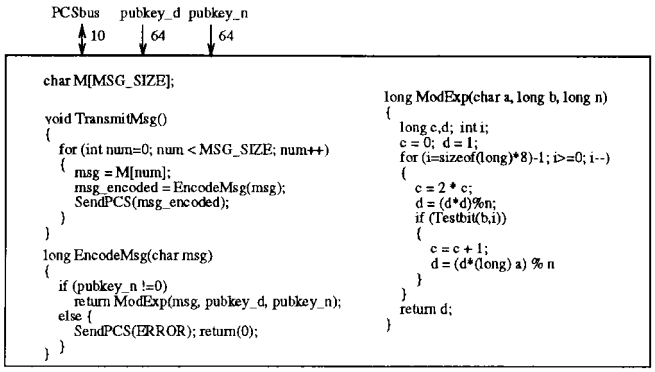

Fig. 2. A single-process RSA encryption example.

II. Problem Description

A. Input

The input to our single-process functional partitioning problem consists of a single functional process, such as a C program or a VHDL process. The process describes a complex repeating sequential computation, often consisting of hundreds of modes or states, and typically requiring many hundreds or thousands of lines of sequential program code. The input process can be viewed as consisting of a set of procedures $F = \{f_1, f_2, \cdots, f_n\}$, with one representing a main procedure (in VHDL, the process body is the main procedure). A variable or port is treated as a simple procedure, with reads and writes being procedure calls. Execution of F consists of procedures executing sequentially, starting with the main procedure, which in turn calls other procedures; at any given time, only one procedure is active—in other words, the procedures are mutually exclusive. If straightforward synthesis were applied to this process, then the result would be a single custom-hardware processor, consisting of a controller and a datapath.

Fig. 2 shows a simple single-process example (simplified from a 230-line C example). The example describes a portable Rivest–Shamir–Adleman (RSA) encryption device that holds a message in a character buffer M. The device has a PC serial-port external interface *PCSbus*, which we assume requires ten I/O's. The *TransmitMsg* procedure encodes and then uploads the message to a PC. The actual encoding is done by procedure *EncodeMsg*, which uses two public keys provided as external inputs, each requiring 64 I/O's. This example is quite small and serves only for illustrating concepts in this paper.

B. Partitioning Approach

Partitioning is achieved by first converting the input into a call graph. For example, Fig. 3 shows a call graph for the earlier example. Each node represents a procedure (recall that variables and ports are treated as procedures), and each edge a procedure call (with edge direction indicating the accessor and accessee, not the direction of data flow). We have annotated the port edges with their data widths, since those edges will become wires because the system's external ports are fixed. Further details on the call graph and annotations can be found in [22]. Partitioning of the nodes is achieved using standard and custom partitioning heuristics. Those heuristics are guided by estimates of design metrics. Estimates are computed using two phases. In the first phase, called preestimation, the call-graph nodes and edges are heavily annotated with numerous metric-related values obtained

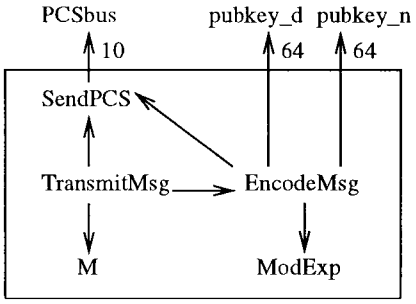

Fig. 3. Call graph for the RSA example.

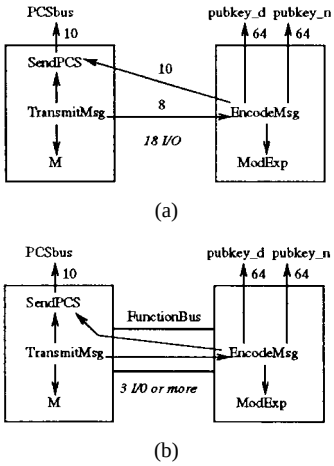

Fig. 4. Partitioned process with alternative internal I/O implementations: (a) cut-edges and (b) FunctionBus.

through rough synthesis and profiling, such as the number of control steps per procedure and the number of bits transferred over an edge. In the second phase, called on-line estimation because it occurs during the actual execution of a partitioning heuristic, annotations are combined using complex equations to yield estimated values for overall design metrics like size, performance, and power. Functional partitioning details have been described elsewhere; we refer the reader to [10].

C. Partitioning Implementation Model

Functional partitioning groups procedures into subsets $\{p_1, p_2, \cdots, p_m\}$, each corresponding to a subprocess, such that every procedure f_i is assigned to exactly one subprocess, i.e., $p_1 \cup p_2 \cup \cdots p_m = F$ and $p_i \cap p_j = \emptyset$ for all $i, j, i \neq j$. Execution of F is the same as above. Since only one procedure in F is active at a time during execution, then only one subprocess will be active at a time, although this mutual exclusion assumption can be relaxed to allow for some parallelism. Fig. 4 shows an example partition of the RSA example among two subprocesses.

Each subprocess p_j will, when generated, consist of a loop that detects a request for one of the subprocess' procedures, receives the necessary input parameters, calls the procedure, and sends back any output parameters. Synthesis will convert each subprocess into a subprocessor; the collection of subprocessors represents an equivalent alternative to a single processor implementation of F. A procedure on a subprocessor may be implemented using any one of various techniques, such as a control subroutine, a datapath component, or even inlining. Synthesis may implement some of a process' procedures in parallel, as long as data dependencies are not violated and no bus contention arises.

We will focus in this paper on subprocesses destined for synthesis, although compilation could convert a subprocess into software running on a standard processor, differing from a synthesized custom processor in its cost, performance, power, design time, etc. Hence the approach described can also be applied to some cases of hardware/software partitioning.

Note that processors are orthogonal to packages. A package, such as an application-specific integrated circuit or an FPGA, is a physical structure that implements processors. A package may implement more than one processor, as is often the case now with system-on-a-chip technology.

III. FunctionBus

Because we focus on a particular functional partitioning subproblem, namely, that of partitioning one large process rather than numerous processes, we can develop a specialized shared bus structure and protocol specifically optimized for that subproblem. In particular, we require that the multiple processes are synchronized such that only one will ever attempt to write to the bus at a given time. This assumption is satisfied in the above problem definition, since the output processes are mutually exclusive.

A. Comparison with Earlier Cut-Edges I/O Approach

Most previous approaches to functional partitioning among hardware parts used a cut-edges I/O approach. Specifically, each edge crossing between parts would require a unique I/O, as illustrated in Fig. 4(a). Each edge would also require one control line to carry out a two-phase handshake data-transfer protocol. A similar cut-edges approach is also used during structural partitioning. However, because procedures modularize a process into pieces that each operate on a data subset, the cut-edges approach used during functional partitioning can yield much less internal I/O than when used during structural partitioning [5].

In the FunctionBus approach, all internal I/O is time-multiplexed over a single bus, as illustrated in Fig. 4(b). The bus protocol must include an address used to demultiplex the bus data. The FunctionBus allows us to choose the internal I/O size, trading off I/O with performance. At one extreme, we can choose a size equal to the maximum size of data to be transferred, and we can choose a one-hot address encoding scheme; note that this extreme is identical to a cut-edges approach in which time-exclusive edges share the same I/O. At the other extreme is a serial bus. We will most often choose something in between these two extremes.

Note that the FunctionBus (as well as the buses created in a cut-edges approach) is intended to serve as an internal bus for a "processor" (in quotes because that processor consists of several subprocessors), as illustrated in Fig. 5. In particular, the FunctionBus is *not* intended as a replacement to existing interprocessor system buses such as peripheral component interconnect (PCI) or controller area network (CAN).

B. Architecture

Fig. 5 illustrates the FunctionBus architecture. Multiple subprocessors are connected by a single bus. 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 subprocessor will control the bus at a time, with the others providing high impedance. Although conceptually N could be as small as one, an N smaller than the size required to encode all addresses could result in significant performance and size penalties due to the extra cycles and hardware needed to decode each address.

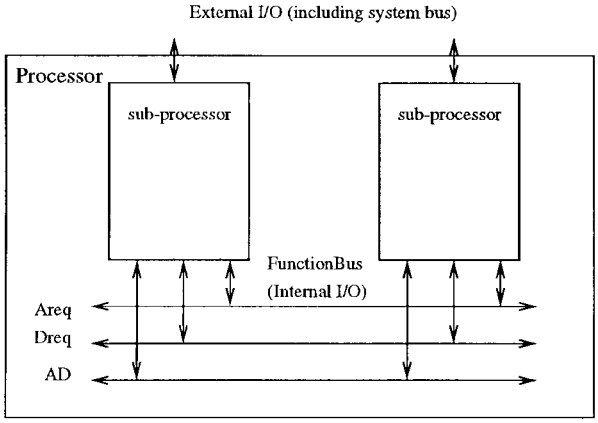

Fig. 5. FunctionBus architecture.

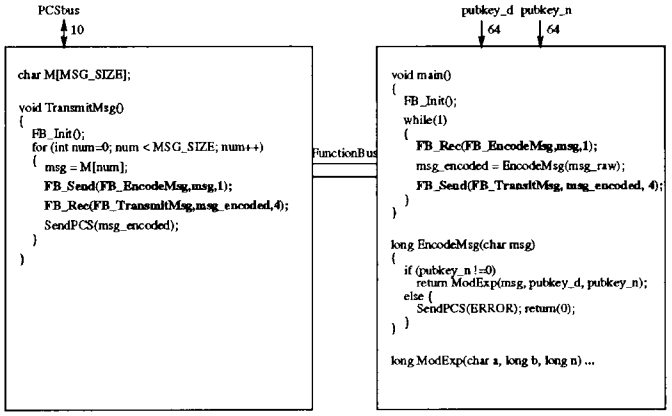

Fig. 6. Refined RSA specification for a partition, using FunctionBus send and receive procedures.

C. General Behavior

We illustrate the general behavior of the FunctionBus using the earlier example of Fig. 2, partitioned into two processes as shown in Fig. 6. Previously, *TransmitMsg* stepped through its message, passing each character to *EncodeMsg*, which in turn passed that character along with two keys (from external ports) to *ModExp*. After partitioning, *TransmitMsg* can no longer call *EncodeMsg* directly, since the latter is on a different subprocessor. Instead, *TransmitMsg* sends the character over the FunctionBus, with a destination address of *FB_EncodeMsg*, and then waits to receive the encoded result. We assume a FunctionBus of size eight, requiring one data transfer by the *FB_Send* routine. Meanwhile, the other process receives the character, calls *EncodeMsg*, and then returns the encoded character by sending it over the FunctionBus to *FB_TransmitMsg*. Note that each procedure must know the return address, i.e., the address of the procedure's caller. A procedure with just one caller has just one return address, so the address can be hard coded. Functions with multiple callers will have multiple return addresses. In some cases, the sequence of return addresses from a given procedure 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, the caller must send its address when calling the procedure. Note that since we require nonrecursive specifications, then there will only be one return address per procedure at a time, and so no stack is necessary. In the case of the simple example above, the FunctionBus transfers are statically determinable, and therefore actually we could eliminate the addresses altogether, but this situation is not the norm.

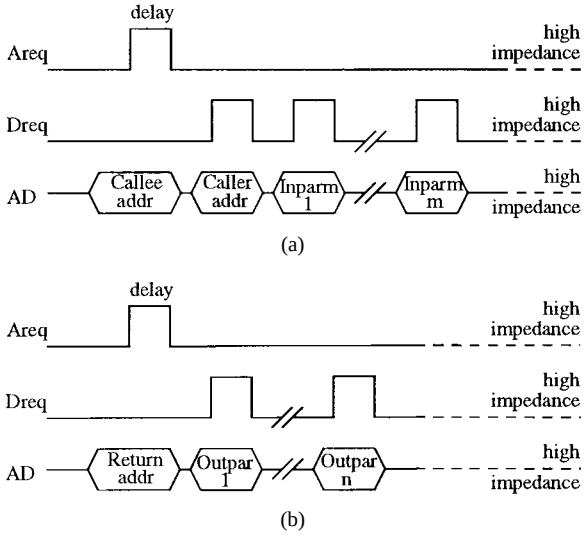

Fig. 7. Timing diagrams: (a) procedure call and (b) procedure return.

D. Communication

From the above discussion, we see that there are two types of communications that occur over the FunctionBus. A *procedure call* consists of sending the address, possibly a return address, and input parameters. A *procedure return* consists of sending the return address and output parameters.

Both communications use similar protocols, shown in Fig. 7. Both begin by placing the address of the receiver procedure [the callee in (a) and the caller in (b)] on AD and pulsing $Areq$. The procedure call protocol then places the return address on AD and pulses $Dreq$. Both then send a sequence of data chunks by placing a chunk on AD and pulsing $Dreq$.

The parameter data must therefore be broken into chunks of size $AD.width$. The number of chunks will thus equal $Ceiling(parms.bits/AD.width)$. As stated earlier, we require $AD.width$ to be at least equal to the number of address bits, eliminating the need to assemble the address bits and thus simplifying the FunctionBus control design.

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. Fig. 8 provides pseudocode FunctionBus procedures suitable for creating C, VHDL, or Verilog implementations of the send and receive protocols. For efficiency, we would likely create unique procedures for each data size being transmitted (byte, word, etc.), eliminating the need for the third parameter. Generating the FunctionBus procedures automatically is a straightforward task.

Since each subprocessor is dedicated to executing one process, each can respond immediately to its address on the bus. Thus, the length of a pulse can be just one clock, when all components use the same clock. We see that a while a procedure call within a subprocessor has no communication overhead, a procedure call to another subprocessor has a minimum overhead of one clock cycle for communication, and possibly more if the parameter data width exceeds the FunctionBus width.

E. Relaxing the Mutual-Exclusion Restriction

We initially assumed that each procedure in F executed in mutual exclusion. This assumption assured us that after partitioning, each

```
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. 8. Send and receive protocols.

process was mutually exclusive with respect to the need to send data over a shared bus, and therefore we could design the FunctionBus without arbitration. We can relax the mutual-exclusion restriction further. After partitioning, we can execute same-process procedures as well as different-process procedures in parallel, as long as we can analyze those procedures such that we can guarantee that two procedures will not simultaneously try to access the FunctionBus. More generally, one can treat the FunctionBus as a shared resource and use static scheduling (if possible) to prevent bus contention. Ideally, after such scheduling, the various processes would operate such that no additional global scheduler would need to be added. Otherwise, a global scheduler process would need to be added, likely requiring lines in addition to the FunctionBus to coordinate the processes.

Further relaxation would require bus arbitration. Such arbitration increases bus cycles and bus logic complexity, but is useful if the input process possesses a large amount of potential parallelism among its procedures. Such a process differs from those we are focusing on having a large number of (sequential) states.

IV. Port-Calling Transformation

A. Overview

In a FunctionBus approach, the internal I/O size is fixed (and typically small, like 16 or 32). Hence the only variation in a subprocessor's I/O comes from the external I/O accessed by that subprocessor's procedures. The port-calling transformation will allow us to redistribute such external port I/O to subprocessors other than the accessing procedure's subprocessor.

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

In a call graph, a port-call procedure is represented as any other procedure, i.e., a node. This node can be partitioned among subprocessors just like any other procedure. If this node is separated from its accessor procedure, 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 interprocessor I/O. Since a port-call node has extremely simple contents, and hence when implemented will not contribute noticeably to a subprocessor's size, it can be partitioned to nearly any subprocessor. We therefore see that introducing port-call nodes, in conjunction with the FunctionBus, yields the ability to freely distribute I/O among subprocessors, at the possible expense of a few extra clock cycles required to pass the data between the port-call procedure and the accessor procedure.

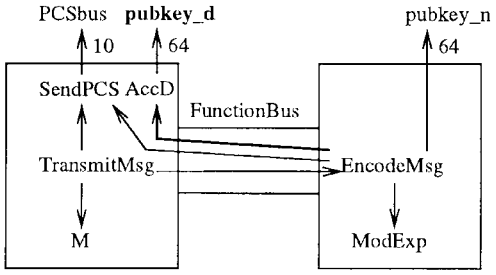

Fig. 9. Port-call transformation enabling redistribution of external I/O in the RSA example.

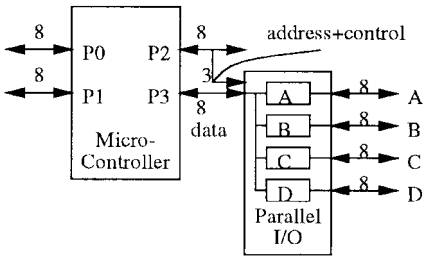

Fig. 10. Extended parallel I/O.

For example, consider the RSA example from Fig. 8. The second subprocessor required 128 external I/O's, while the first only required ten. We can alleviate this imbalance using port calling. We introduce a port-call procedure *AccD* for accessing the *pubkey_d* port, and then we repartition by moving that procedure to the first subprocessor, as in Fig. 9. The external I/O for the subprocessors becomes 64 and 74; the internal I/O stays the same.

We must decide which port accesses should have port-call nodes introduced to achieve improvements. Predicting those accesses that, when replaced by port-call nodes, would lead to such improvements is a difficult task. We note, though, that the number of external ports typically grows only sublinearly with respect to the specification size. Therefore, we can introduce a port-call node for every port access without excessive growth in computational complexity. Thus, our approach is to:

1) *transform* the call graph by introducing port-call nodes for every port access;
2) *partition* the call graph using existing heuristics;
3) *inverse transform* the call graph by eliminating each port-call node that appears on the same subprocessor as its accessor.

The *inverse transform* step is necessary to eliminate unnecessary port-call nodes, so that only those nodes needed to distribute I/O to another subprocessor remain. Note that the partitioning heuristics and associated estimation models and cost functions need not be changed to account for the port-call transformation, since the introduced nodes look just like any other procedures.

Note that port calling is a generalization of the commonly used design technique of extended parallel I/O. For example, consider Fig. 10. A microcontroller 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 microcontroller and PIO chip is akin

```
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. 11. Port-calling procedures.

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 during hardware/software partitioning, or even another microcontroller.

B. Port-Calling Procedures

After the call graph is partitioned, a new subprocess must be generated for each group of procedures representing a subprocessor. Each subprocess will monitor the FunctionBus for the address of one of its procedures, capture any input parameters from the bus, call the procedure, return by placing a return address and any output parameters on the bus, and resume detecting an address. A call to a procedure of another subprocess is replaced by FunctionBus call and return routines. Port-calling procedures require no special treatment, appearing as any other procedure. We thus only describe the contents of such procedures here; partitioning and subsequent FunctionBus routine insertion will take care of the communication between the port-call procedure and the accessor procedure.

The port-call procedures for accessors that read, write, or both read and write a port are shown in Fig. 11. Note that each is trivial to implement, and so could be moved freely among parts. Inserting these procedures automatically is a straightforward task.

V. Experiments

We have implemented a functional partitioning tool as part of the SpecSyn system exploration environment [10]. We have extended the tool to support the FunctionBus and port calling. Earlier work demonstrated that functional partitioning can significantly reduce I/O requirements compared with structural partitioning [5].

A. FunctionBus Versus Cut-Edges I/O During Functional Partitioning

Functional partitioning using a FunctionBus approach can yield even further I/O improvements. To demonstrate this idea, we experimented with five examples: an Ethernet coprocessor (*ether*), a fuzzy logic controller (*fuzzy*), an interactive TV settop box (*itv*), a microwave transmitter controller (*mwt*), and an answering machine (*ans*). Each consisted of a few hundred lines of VHDL algorithmic code. We performed two-way partitioning on each example, using the simulated annealing heuristic built into the partitioning tool, and using a cost function seeking to minimize execution time and I/O while maintaining balanced subprocessor sizes. In particular, a simplified version of the cost function looks like

$$\text{Cost} = N_{\text{et}}(et) + N_{\text{io}}(io) + N_{\text{size}}(|\text{size1} - \text{size2}|)$$

where et is execution time, io is total I/O, size1 and size2 are the sizes of the two parts, and the N's are functions that normalize each term to a number between one and zero. Table I shows the I/O improvements obtained for each example for a FunctionBus data-bus size of 16, along with the performance penalties. Note that these I/O improvements are in addition to those I/O improvements already gained over structural partitioning; examples of such complexity typically have several hundred I/O's for a two-way structural partition.

TABLE I
I/O Improvements Using the FunctionBus Versus a Cut-Edges Approach During Functional Partitioning

Example	Size0 (gates)	Size1 (gates)	IO0 (wires)	IO1 (wires)	Perf. (cycles)	% improvement Max IO	Total IO	Perf.
ether	12142	12527	119	154	196			
ether_fb	11795	12874	53	34	194	66%	68%	1%
fuzzy	52414	59287	43	67	7980			
fuzzy_fb	51899	59802	26	34	10888	49%	45%	-36%
itv	54746	98386	140	40	9617			
itv_fb	51649	101483	91	61	10049	35%	16%	-4%
mwt	4734	5224	65	79	759			
mwt_fb	4511	5447	40	32	799	49%	50%	-5%
ans	6387	6244	119	66	44			
ans_fb	6140	6491	85	98	44	18%	1%	0%

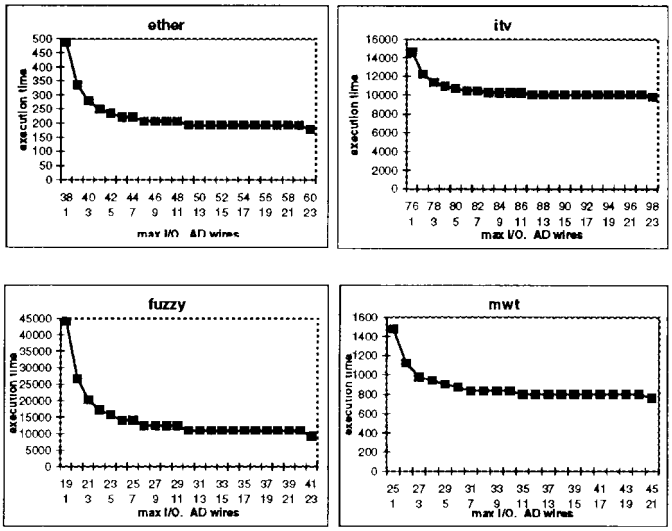

Fig. 12. I/O and performance tradeoffs by varying FunctionBus size.

B. I/O and Performance Tradeoffs Using the FunctionBus

The above results assumed a fixed data bus of 16 bits. However, a key feature of the FunctionBus is the ability to trade off I/O with performance by varying the data-bus size. Fig. 12 illustrates the I/O and performance tradeoffs obtained by varying the FunctionBus size for several of the examples (given a fixed partition for each example). The numbers on the x-axis at the top are the maximum I/O for either subprocessor, while the numbers at the bottom are the bus size. The y-axis shows the performance in cycles. We can see that the FunctionBus provides quite a bit of flexibility to trade off I/O with performance.

Particular attention should be paid to the values on the y-axis. Note that the *fuzzy* example is extremely sensitive to the bus size, meaning that the partition had significant interprocessor communication. Thus, a large FunctionBus might be in order for that example if performance is important. On the other hand, the *itv* example did not demonstrate this sensitivity, so a smaller FunctionBus might be used. Of course, the actual selection of the bus size depends on the relative importance of performance, I/O, and other metrics.

TABLE II
I/O AND PERFORMANCE IMPROVEMENTS WHEN USING THE PORT-CALLING TRANSFORMATION IN CONJUNCTION WITH THE FUNCTIONBUS DURING FUNCTIONAL PARTITIONING

Example	Size0 (gates)	Size1 (gates)	IO0 (wires)	IO1 (wires)	Perf. (cycles)	% improvement Max IO	Total IO	Perf.
ether_fb	13326	11343	39	48	194			
ether_fb_pc	13478	11202	38	37	194	21%	14%	0%
fuzzy_fb	51899	59802	26	34	10888			
fuzzy_fb_pc	58511	53193	34	26	8356	0%	0%	23%
itv_fb	51649	101483	91	61	10049			
itv_fb_pc	53288	99872	70	57	9653	23%	16%	4%
mwt_fb	4511	5447	40	32	799			
mwt_fb_pc	4877	5095	35	23	791	13%	19%	1%
ans_fb	6140	6491	85	98	44			
ans_fb_pc	6164	6494	32	35	44	64%	63%	0%

C. Port Calling

The FunctionBus reduces the number of internal I/O's, thus reducing maximum I/O per subprocessor as well as total I/O. The maximum I/O per subprocessor can be reduced further using port calling. We applied the same partitioning heuristic on the examples, but this time applying the port-call transform before partitioning and the port-call inverse transform after partitioning. Table II summarizes results comparing maximum I/O, total I/O, and performance when partitioning using a FunctionBus without and with port calling. They show improvements in maximum I/O averaging 24%, with the improvement in one case reaching 64%. Again, these improvements are in addition to those obtained using the FunctionBus over the cut-edges.

Before performing the port-calling experiments, we expected port calling to decrease maximum I/O, but we expected total I/O to stay the same (recall that we fixed the FunctionBus size at 16 for these partitioning experiments). However, we were surprised to see that total I/O actually *decreased* significantly. Upon investigation, we discovered the reason for this decrease. In many cases, multiple procedures access the same external port. When these procedures exist on different subprocessors, then each subprocessor requires external I/O to connect to the port. However, because port-call procedures are small and our cost function sought to minimize I/O, the introduced port-call procedures of one port were almost always partitioned to a single subprocessor, so only that subprocessor required external I/O for the port.

We also expected that performance would suffer slightly, due to the increased time to transfer external port data through a port-call procedure over the FunctionBus to/from the accessing procedure. However, again in all the examples, there was actually an *improvement* in performance or no change at all. We believe this can be explained as follows. Before introducing port calling, the partitioning heuristic had to simultaneously satisfy three difficult metrics: minimize I/O, minimize execution time, and keep sizes balanced. After introducing port calling, the I/O metric became much easier to satisfy, therefore freeing the heuristic to investigate many other partitions previously prohibitive due to I/O, and thus finding partitions with lower execution times.

VI. CONCLUSIONS

We have introduced two related techniques that relax I/O problems during functional partitioning. Using the FunctionBus, we can reduce the internal I/O size arbitrarily, trading off I/O size with performance. Using port calling, I/O can be easily redistributed from one subprocessor to another. We showed through experiments that these two techniques yield extremely small I/O sizes for several examples, with only small overall performance penalty, if any. Thus, port calling and the FunctionBus are important features in a functional partitioning tool, and such tools are becoming more important for improving I/O, synthesis runtime, power, cost/flexibility tradeoffs, and physical design problems.

REFERENCES

[1] G. DeMicheli, *Synthesis and Optimization of Digital Circuits.* New York: McGraw-Hill, 1994.

[2] F. Johannes, "Partitioning of VLSI circuits and systems," in *Proc. Design Automation Conf.,* 1996.

[3] R. Tessier, J. Babb, M. Dahl, S. Hanono, and A. Agarwal, "The virtual wires emulation system: A gate-efficient ASIC prototyping environment," in *Proc. Int. Symp. Field-Programmable Gate Arrays (FPGA),* 1994.

[4] E. Macii, M. Pedram, and F. Somenzi, "High-level power modeling, estimation and optimization," in *Proc. Design Automation Conf.,* 1997.

[5] E. Hwang, F. Vahid, and Y. C. Hsu, "Functional partitioning for reduced power," in *Proc. Design Automation and Test in Europe (DATE),* to be published.

[6] E. Hwang, F. Vahid, and Y. C. Hsu, "Experiments on functional partitioning for reduced power consumption," Computer Science Dept., University of California, Riverside, Tech. Rep. 98-2, 1998.

[7] L. Hammond, B. Nayfeh, and K. Olukotun, "A single-chip multiprocessor," *IEEE Comput. Mag.,* vol. 30, no. 9, pp. 79–85, Sept. 1997.

[8] R. Gupta and G. DeMicheli, "Hardware-software cosynthesis for digital systems," *IEEE Design Test Comput. Mag.,* pp. 29–41, Oct. 1993.

[9] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," *IEEE Design Test Comput. Mag.,* pp. 64–75, Dec. 1994.

[10] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems.* Englewood Cliffs, NJ: Prentice-Hall, 1994.

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

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

[13] A. Balboni, W. Fornaciari, and D. Sciuto, "Partitioning and exploration strategies in the Tosca co-design flow," in *Proc. Int. Workshop Hardware-Software Co-Design,* 1996, pp. 62–69.

[14] J. Hou and W. Wolf, "Process partitioning for distributed systems," in *Proc. Int. Workshop Hardware-Software Co-Design,* 1996, pp. 70–75.

[15] R. Camposano and J. T. van Eijndhoven, "Partitioning a design in structural synthesis," in *Proc. Int. Conf. Computer Design,* 1987.

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

[17] R. Gupta and G. DeMicheli, "Partitioning of functional models of synchronous digital systems," in *Proc. Int. Conf. Computer-Aided Design,* 1990, pp. 216–219.

[18] C. H. Gebotys, "An optimization approach to the synthesis of multichip architectures," *IEEE Trans. VLSI Syst.,* vol. 2, no. 1, pp. 11–20, 1994.

[19] Y. Y. Chen, Y. C. Hsu, and C. T. King, "MULTIPAR: Behavioral partition for synthesizing multiprocessor architectures," *IEEE Trans. VLSI Syst.,* vol. 2, pp. 21–32, Mar. 1994.

[20] K. Kucukcakar and A. Parker, "CHOP: A constraint-driven system-level partitioner," in *Proc. Design Automation Conf.,* 1991, pp. 514–519.

[21] F. Vahid and D. Gajski, "Specification partitioning for system design," in *Proc. Design Automation Conf.,* 1992, pp. 219–224.

[22] ____, "SLIF: A specification-level intermediate format for system design," in *Proc. Eur. Design and Test Conf. (EDTC),* 1995, pp. 185–189.