Functional Partitioning Improvements Over Structural Partitioning for Packaging Constraints and Synthesis: Tool Performance

FRANK VAHID
University of California
THUY DM LE
IMA
and
YU-CHIN HSU
University of California

Incorporating functional partitioning into a synthesis methodology leads to several important advantages. In functional partitioning, we first partition a functional specification into smaller subspecifications and then synthesize structure for each, in contrast to the current approach of first synthesizing structure for the entire specification and then partitioning that structure. One advantage is the improvement in I/O performance and package count, when partitioning among hardware blocks with size and I/O constraints, such as FPGAs or blocks within an ASIC. A second advantage is reduction in synthesis runtimes. We describe these important advantages, concluding that further research on functional partitioning can lead to improved results from synthesis environments.

Categories and Subject Descriptors: B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids; *Automatic synthesis*; *Hardware description languages*; *Optimization*; J.6 [**Computer Applications**]: Computer-Aided Engineering—*Computer-aided design* (CAD)

General Terms: Design

Additional Key Words and Phrases: Behavioral systhesis, functional partitioning, system-level design

1. INTRODUCTION

Functional specifications, consisting of a machine-readable program-like description of a system's desired behavior, are now commonly developed

Authors' addresses: F. Vahid, Department of Computer Science, University of California, Riverside, CA 92521; email: vahid@cs.ucr.edu; T. D. Le, IMA, Irvine, CA; Y.-c. Hsu, Department of Computer Science, University of California, Riverside, CA 92521.

when building digital systems. Such specifications can be input to simulators for early behavior verification, as well as to synthesis tools for automatic structural design.

In addition, a functional specification can be functionally partitioned to solve numerous problems, including exploring hardware/software tradeoffs, satisfying hardware-part size and I/O constraints, and reducing synthesis runtimes. Functional partitioning means distributing the specification's functions for implementation among some set of components. The first problem has received much recent research attention, with the focus on functionally partitioning a specification among software processors and custom-hardware processors to achieve good cost and performance tradeoffs [Antoniazzi et al. 1994; Eles et al. 1992; Ernst et al. 1994; Gupta and DeMicheli 1993; Thomas et al. 1993; Vahid and Le 1996; Xiong et al. 1994]. The functional-partitioning solutions for the latter two problems have received less attention but, as we will demonstrate, are very important in synthesis environments.

The problem of satisfying hardware-part size and I/O (input/output) constraints, including application-specific integrated circuit (ASIC) chips, field-programmable gate array (FPGA) chips, or blocks within such chips, has received much research attention for several decades [Johannes 1996]. The focus, though, has been on partitioning already designed structures, such as the gate-level netlist implementation of a system. Such structural partitioning is I/O dominated; in other words, when partitioners fail to partition a design among a given set of parts, it's usually because of an I/O pin shortage rather than a gate shortage [Tessier et al. 1994]. While new techniques multiplex wires to reduce I/O [Tessier et al. 1994], the problem is still hard to solve once one has designed structure. Designers often solve this I/O problem by manually performing functional partitioning, where a system's functions are first partitioned among parts and then each part's functions are implemented as structure. The existence of functional specifications means that such functional partitioning can now be automated. In fact, several research efforts have addressed such automation, hypothesizing that functional partitioning would excel over structural partitioning [Lagnese and Thomas 1991; Gupta and DeMicheli 1990; Kucukcakar and Parker 1991; Chen et al. 1994; Vahid and Gajski 1992]. This paper provides empirical results for the hypothesis that *functionally partitioning a specification results in better satisfaction of hardware-part size and I/O constraints, and often yields better design performance than structural partitioning*.

For example, consider implementing the simple system whose functional specification is shown in Figure 1 with some number of Xilinx XC4002PC64 FPGAs. Such FPGAs have 2000 gates (we assume this number represents usable gates) and 64 I/O pins. (Note that the specification and the FPGAs are very small and are used for illustrative purposes only). The traditional approach for achieving an implementation consists of three steps. First, we create a register-transfer (RT) level structural design to implement the behavior, as shown in Figure 2. The design has a datapath with registers

```
entity Example is
        port(   clk             : in bit;
                Z               : out integer);
end Example;

architecture bhv of Example is
        signal      a, b, c, d, e:      integer;
        signal      x, y:               integer;
begin
        main: process
                procedure P2 is
                begin
                        x <= a + b;
                end P2;

                procedure P3 is
                begin
                        if ( a > d) then
                                y <= a + c;
                        else
                                y <= d + c;
                        end if;
                end P3;

                procedure P1 is
                begin
                        P2;
                        P3;
                        wait until clk'event and clk = '1';
                        Z <= x + y;
                end P1;

        begin
                wait until clk'event and clk = '1';
                P1;
        end process main;
end bhv;
```

Fig. 1. Example's functional specification.

Table I. Component Gate Counts

Functional units	Type	Area (gate)
REG	32-bits	224
LESS	32-bits	190
ALU	32-bits	384
2-1 MUX	32-bits	97
3-1 MUX	32-bits	193
4-1 MUX	32-bits	288

(e.g., *a, b, c*), multiplexors (MUX), and an arithmetic-logic unit (ALU), and also has a control unit that sequences data through the datapath. Second, we attempt to implement the design on a single FPGA. However, we find that the design requires 2650 gates, exceeding the 2000 gate constraint. Third, we add another FPGA, and try partitioning the design among the FPGAs in a manner satisfying FPGA size and I/O constraints. Figure 3

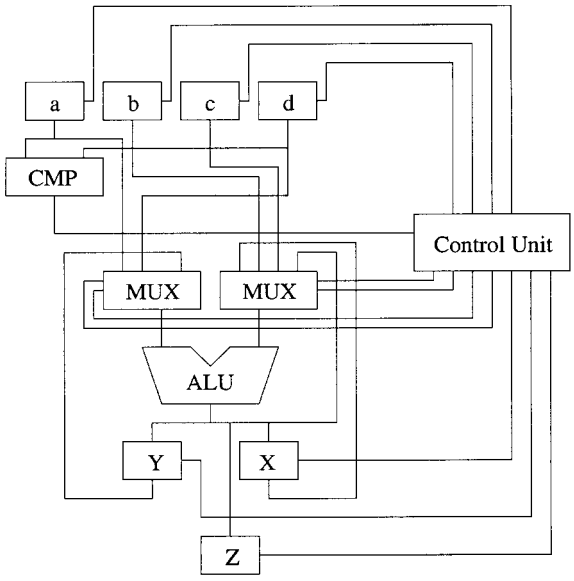

Fig. 2. Example's RT-level structural design.

shows four possible two-way partitions, where each part is annotated by its size and cut. A part's size is the sum of its functional unit sizes, using the component library of Table I. A part's cut is the number of wires crossing between the parts, representing the required I/O. Unfortunately, none of the four partitions satisfies constraints. In Figure 3(a), we partitioned to minimize the cut by putting the control unit into its own part, but this partition has a part violating the 2000 gate FPGA size constraint. The other partitions were created with more balanced part sizes, but these all violate the FPGA I/O constraints.

Failing to successfully partition the design among two FPGAs, we repeat the third step of adding an FPGA and repartitioning, until we finally find a partition that meets the size and I/O constraints. From this example, one can see that structural partitioning often requires many more parts than the design's gate count alone implies. For this example, gate count implies two FPGAs, but we could not find a constraint-satisfying partition that used less than four FPGAs.

Functional partitioning represents an alternative approach. In the third step above, after adding an FPGA, we *partition the functional specification* among two parts, rather than partitioning the original structural design. In this example the specification's three subroutines, *P1, P2,* and *P3*, serve as the functional objects to be partitioned among FPGAs. There are three possible functional partitions, as shown in Figure 4. For each partition, we design an RT-level implementation for each part.

Note that the partition in Figure 4(b) satisfies constraints, so functional partitioning led to a two FPGA solution. Alternatively, if the I/O constraints are violated, we simply create a bus between the two parts, and

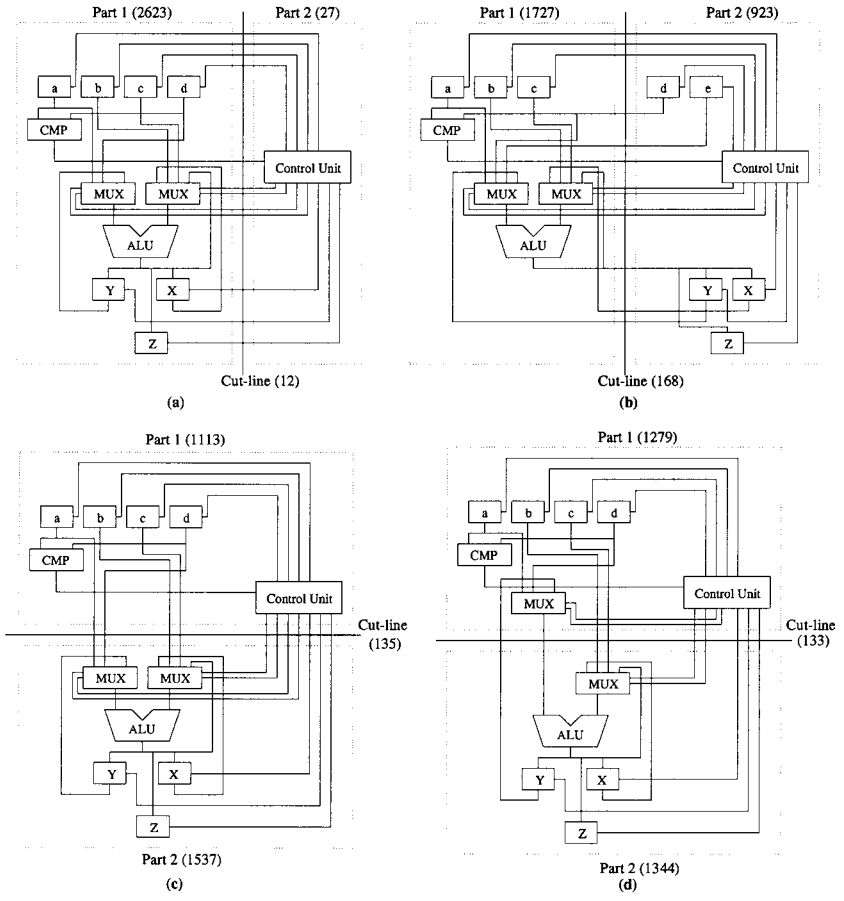

Fig. 3. Possible structural partitions: no two-FPGA partition found.

modify the specification to time-multiplex the data transfers between the two parts over a single 32-bit bus, as illustrated by Figure 5. The part sizes increase slightly due to additional multiplexors and control. In contrast, introducing such a bus after structural partitioning is extremely difficult.

Functional partitioning yields better performance than structural partitioning in this example. For the structurally-partitioned design, the clock period is determined by the MEBS [Hsu et al. 1994] behavioral synthesis tool to be 43 nanoseconds, which includes an interchip delay of 7 nanoseconds, and 10 cycles are necessary for the behavior's execution. For the functionally-partitioned design, the clock period is determined to be only 32 nanoseconds, with 3 extra cycles needed for data transfers, for a total of 13 cycles. Thus, the total execution times t_{sp} and t_{fp} achieved by structural and functional partitioning, respectively, are:

$$t_{sp} = 43 \times 10 = 430ns$$

$$t_{fp} = 32 \times 13 = 416ns$$

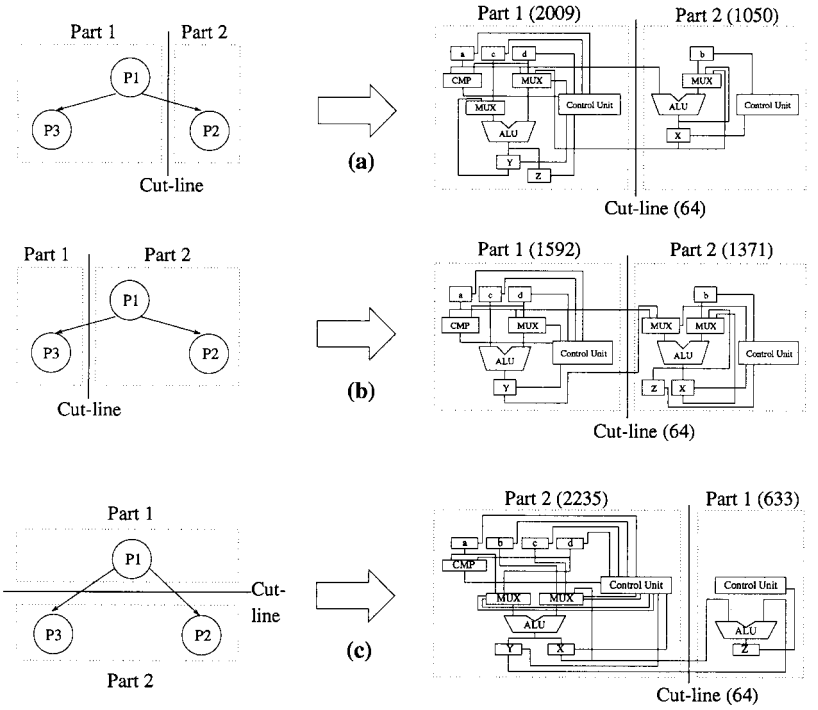

Fig. 4. Possible functional partitions: two-FPGA partition found.

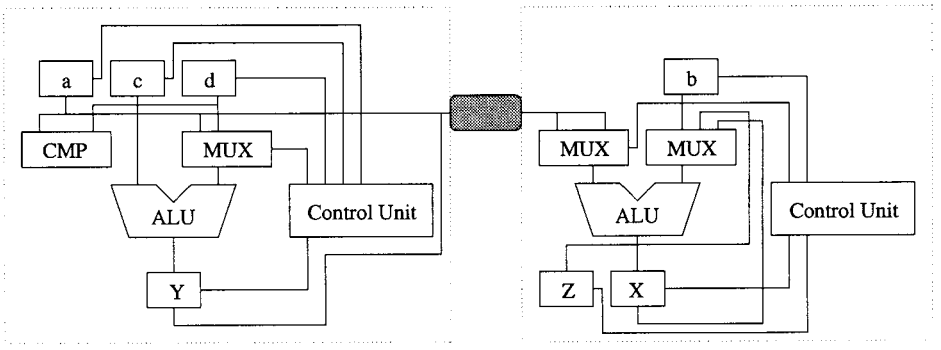

Fig. 5. Functional partitioning with bus: Permits even further I/O reduction.

In other words, functional partitioning yields improved performance because, although additional clock cycles are needed to transfer data from one part's processor to another, the clock period is reduced, since each processor is fully contained within a part, and hence the critical path does not cross between parts, and this reduction offsets the additional cycles. For larger examples, the number of clock cycles is larger, so the reduced clock period can have an even greater effect.

Intuitively, functional partitioning excels over structural partitioning by assigning each function to one part, rather than spreading a function over several parts. Such isolation (1) reduces I/O; (2) prevents the critical path from crossing parts, thus reducing the clock period; and (3) often yields simpler hardware, thus further reducing the clock period. I/O is reduced because I/O is only needed to transmit parameters between functions, and the number of such parameters is typically small. The critical path doesn't cross between parts because a critical path is the longest register-to-register transfer of a processor, but each processor is fully contained within a part. Simpler hardware results because two small processors are each simpler than one large processor. Perhaps more importantly, we have *complete control over I/O* at the functional level, and can easily tradeoff performance and I/O. In particular, data transfers between parts can be easily time-multiplexed over one bus by inserting an addressed-protocol behavior (or even an arbiter behavior), and transfers can be partially or fully serialized. Another advantage is that each part's behavior is readable and late changes are often isolated to one part, leading some designers to call this approach "partitioning for debug."

These advantages come with some drawbacks. First, functional partitioning must be guided by estimates of size, I/O, and performance for possibly thousands of examined partitions, but obtaining fast yet accurate size and performance estimates can be hard. In contrast, in structural partitioning, we can easily estimate size and and delay quickly and accurately by summing object sizes and counting critical path cuts. However, sophisticated estimation techniques for use with functional partitioning can help alleviate this drawback [Vahid and Gajski 1995]. Second, a functionally-partitioned system often (though not always, as we shall see) uses more gates, since hardware units are not shared by functions on different parts. However, structural partitioning is I/O dominated, and hence does not use all the gates on a part in any case, so this increase is often not very significant.

The problem of reducing synthesis runtime using functional partitioning has also received only limited attention. Synthesis runtime, as well as memory usage, may become excessive for large specifications. By functionally partitioning the specification into several smaller subspecifications and applying synthesis to each independently, runtimes can be reduced by an order of magnitude. An analogous problem and solution are presented in Camposano and Brayton [1987] for partitioning logic equations before logic synthesis; in this paper we describe results of partitioning a functional specification before behavioral synthesis.

This paper is organized as follows. In Section 2 we discuss related work in functional partitioning. In Section 3 we describe experiments that demonstrate the superiority of functional over structural partitioning with several examples. In Section 4 we demonstrate the improvement in synthesis-tool performance achievable with functional partitioning. Section 6 provides conclusions.

2. RELATED WORK

In this section we summarize related efforts in functional partitioning, and describe their relationship to our work; we refer the reader to Kirkpatrick and Cheng [1991]; Fiduccia and Mattheyses [1982]; Kernighan and Lin [1970]; Krishnamurthy [1984]; Sanchis [1989];Sechen and Chen [1988] for information on structural partitioning. We first describe five features that can be used to compare functional partitioning systems: computation model, granularity, heuristics, estimation techniques, and design flow.

The *computation model* is the underlying model of the system's input specification (regardless of the input language). Possible models include synchronous dataflow [Lee and Messerschmitt 1987]; hierarchical/concurrent finite-state machines [Harel 1987]; communicating sequential processes [Hoare 1978; IEEE Inc., 1988]; program-state machines [Vahid et al. 1995]; and control/dataflow graphs (CDFGs) [Gajski et al. 1991].

Granularity is a measure of the complexity of each partitioning object into which we decompose an input specification, where such objects might be processes, procedures or functions, statement blocks, statements, or even fine-grained arithmetic operations. These atomic objects are assigned to parts during partitioning. Fine granularities yield more partitioning choices, but require longer heuristic runtimes and result in less accurate estimations [Vahid and Gajski 1995]. Many systems partition at the granularity of arithmetic operations [Lagnese and Thomas 1991; Gupta and DeMicheli 1990; Chen et al. 1992; Kucukcakar and Parker 1991].

Partitioning heuristics can be classified as either iterative-improvement or constructive heuristics. Constructive heuristics start with no initial partition, and build a partition. A common constructive heuristic is clustering, in which objects deemed by a closeness function to be "close"to one another are merged until only a few objects remain. On the other hand, iterative-improvement heuristics start with a complete partition, and modify the partition to try to find a better one as defined by some cost function. Examples include group-migration [Kernighan and Lin 1970] and simulated annealing [Kirkpatrick et al. 1983].

Estimation techniques are the methods used to quickly yet accurately evaluate design metrics, like size, I/O, and performance, without actually synthesizing a design. Since iterative-improvement heuristics usually examine far more partitions than constructive ones, they require faster estimation techniques.

The *design flow* is the sequence of steps comprising the partitioning process and the type of designer interaction. A designer might be allowed much interaction, such as selecting closeness or cost functions, choosing granularity, manually moving objects between parts, and iterating the partitioning process.

Now that we have summarized the key features of partitioning systems, we proceed to summarize several approaches. These approaches can be broadly categorized into that of hardware/hardware partitioning and the

more recent hardware/software approach. We focus on hardware/hardware approaches, since they are most closely related to the problems we address.

2.1 BUD

BUD [McFarland and Kowalski 1990] was designed to provide accurate estimations of area and delay to guide high-level synthesis decisions. BUD takes a single process as input and partitions the arithmetic operations into groups. Each group represents a datapath component, so partitioning accomplishes the synthesis allocation step. In addition, operations in a group must execute sequentially because they use one component, so partitioning essentially determines a scheduling also. After partitioning, BUD quickly places the components and derives area and delay information. Thus output is used to create the best design during synthesis.

BUD's partitioning design flow consists of several steps. First, the input process is converted to a CDFG. Second, the CDFG is decomposed into a set of arithmetic and logical operations. Third, a cluster tree is built. A cluster tree represents successive mergings of the closest operations until only one object remains. Closeness of operations is measured with a closeness function incorporating three closeness metrics: the minimal number of functional units needed to perform the operations; the number of wires shared by operations; and the parallelism of operations. Fourth, different tree cutlines (i.e., different numbers of clusters) are evaluated using an objective function that is a weighted sum of area and execution time, and the best cut-line is chosen.

2.2 Aparty

Aparty [Lagnese and Thomas 1991] is an architectural partitioner in the System Architect's Workbench, intended to improve register-transfer designs, particularly by reducing global routing wires. Aparty tries to solve two of BUD's limitation. One, the basic clustering technique in BUD produces error in estimating the closeness values between two clusters as clustering proceeds, because the closeness between clusters is estimated rather than recomputed. Two, the closeness function in BUD combines different metrics such as wires, sharable hardware, and parallel execution into one number, but it may be very hard to balance the relative weights of these metrics to achieve a good design.

Aparty solves these problems by performing clustering in multiple stages, allowing each stage to use any combination of closeness metrics. Aparty also defines several new metrics, including those based on control, data, procedures/control, procedures/data, and operators. The partitioning design flow consists of converting the input process into a CDFG and decomposing the CDFG into arithmetic-level operations. Then the user chooses a closeness function, applies clustering, and chooses the criteria for selecting the best cutline, such as smallest area, minimum connections, or shortest schedule length. Rough synthesis is used for estimation. The best cut is made, resulting in a set of groups. The user can then repeat the

process by selecting a new closeness function and repeating clustering on the groups, until a satisfactory design is achieved.

2.3 Vulcan I

Vulcan I [Gupta and DeMicheli 1990] is designed to partition a system for chip-area, I/O, and latency constraints. First, Vulcan converts the input process into a hierarchical CDFG, annotating each node with sizes and latencies and edges with widths. Second, Vulcan partitions the hierarchical CDFG at the coarsest granularity among chips using the group migration and simulated annealing partitioning heuristics, trying to satisfy constraints. Estimates are computed by summing sizes and latencies and computing cut sizes. Third, if constraints are violated, Vulcan decomposes certain CDFG nodes, usually choosing the larger nodes, and repeats the partitioning process.

2.4 Multipar

Multipar [Chen et al. 1992] is designed to maximize system performance and minimize communication between hardware components by simultaneously considering scheduling and partitioning. Multipar converts the input process into a CDFG, along with communication properties and processor constraints. It then translates the CDFG into an integer linear programming (ILP) formulation. To minimize the communication between system components, Multipar performs as-soon-as possible and as-late-as possible scheduling to determine the range and mobility of operations for the ILP formulation. To partition a large CDFG, Multipar uses a simplified ILP formulation and a recursive method. The scheduled CDFG is recursively partitioned into two groups such that the two groups have balanced area and minimal communication. The process stops when the groups are small enough to accurately estimate the partitioned cost to satisfy the constraints.

2.5 Chop

Chop [Kucukcakar and Parker 1991] explores system partition alternatives. It uses probability methods to determine the implementation feasibility of partitions. The designer creates and modifies a partition, and Chop evaluates the feasibility of the partition's implementation. There are several steps for evaluating a partition. First, Chop takes the input DFG with a component library and initial partition. It uses BAD (behavioral area-delay predictor) to generate to several possible implementations; at this point Chop is not concerned with the feasibility of the generated implementations. Second, Chop searches for all possible combinations of generated implementations to match the design characteristics of different modules, such as area, performance, delay, etc. Third, Chop integrates the implementations into the system and evaluates different global feasible implementations with input constraints and chip sets. If the complete predicted implementations don't meet the constraints, then the designer can modify

the partition and repeat the process. Otherwise, no possible implementation is predicted.

2.6 Yorktown Silicon Compiler (YSC)

YSC [Camposano and van Eijndhoven 1987, Camposano and Brayton 1987] is intended to improve logic-synthesis tool performance. YSC's partitioning scheme is very similar to BUD's. There are three steps for partitioning a process in YSC. First, the behavior is converted to a CDFG. Second, closeness values are computed among CDFG operations, using the following closeness metrics: the similarity of two operations, meaning they can share some logic gates, the number of inputs and outputs shared by clusters, the number of inputs and outputs connecting clusters, and the cluster size. Third, YSC uses a hierarchical clustering algorithm to form clusters, with a closeness threshold used to terminate clustering. Results showed greatly reduced logic synthesis times when synthesizing each of the clusters separately, as compared with synthesizing the entire behavior.

2.7 SpecSyn

SpecSyn [Gajski et al. 1994; Vahid and Gajski 1992] is a partitioner at the procedure level. SpecSyn considers three design metrics: area, I/O, and performance. It converts the input processes or program-state machines into SLIF [Vahid and Gajski 1995] format, resembling a call graph where the nodes are procedures and variables and edges represent accesses. The SLIF is partitioned manually or automatically by clustering, group migration, or simulated annealing algorithms. Two types of estimators are used, one to initially annotate the the SLIF and another to rapidly combine annotations during iterative-improvement partitioning into metric values. Partitioning can be repeated with different allocations of parts until the constraints are satisfied. SpecSyn then generates a refined specification.

2.8 Hardware/Software Functional Partitioning

There are many efforts in hardware/software functional partitioning. Vulcan II [Gupta and DeMicheli 1992] starts with an all-hardware implementation and uses a greedy heuristic to move threads of operations from hardware to software to reduce hardware costs. Its cost function considers the metrics of hardware size, software size, synchronization cost, and communication frequencies. Cosyma [Ernst et al. 1993] starts with an all-software implementation and migrates statement blocks to hardware by using simulated annealing. Partif [Ismail et al. 1994] provides the designer with a suite of specification transformations to manually partition a specification. Ptolemy [Kalavade and Lee 1994] provides a custom partitioning heuristic for simultaneously partitioning and scheduling tasks among hardware and software processors. SpecSyn [Gajski and Gong 1994] provides an environment for interactive exploration of different partitions of functions (data, behavior, and communication) among various system components (processors, memories, buses). Various heuristics are provided,

including clustering, group migration, and simulated annealing, along with sophisticated estimators. Tosca [Antoniazzi 1994] uses clustering to partition a hierarchical-concurrent finite-state machine. Tabu-search is used to partition procedures in Eles et al. [1996].

In much of the previous work described here, functional partitioning researchers [Gupta and DeMicheli 1990; Kucukcakar and Parker 1991; Vahid and Gajski 1992] predicted that functional partitioning is better than structural partitioning. This work provides empirical results demonstrating this hypothesis.

3. FUNCTIONAL VS. STRUCTURAL PARTITIONING FOR PACKAGE CONSTRAINTS

3.1 Method

Figure 7 illustrates the difference between structural and functional partitioning. Current automated tools satisfy package constraints by performing structural partitioning. In structural partitioning, one first implements the system with structure, consisting of a controller and datapath pair containing connected RT and gate-level objects. One then partitions the structure among parts. In functional partitioning, the specification's functions are partitioned among parts. Inter-part communication is achieved using a high-level transfer protocol such as a handshake. Each part's functions are then implemented as structure, optionally repeating the functional partitioning within each part. (In hardware/software partitioning, a part's implementation is achieved by synthesizing either software or hardware for each part).

Functional partitioning requires estimations of performance, I/O, and hardware size for any given set of functions, making such partitioning more challenging to automate because obtaining accurate estimates is difficult. In structural partitioning, on the other hand, one merely associates a size and delay with each hardware object, and then sums sizes to obtain size estimates and introduces delays for intercomponent communications to obtain performance estimates. However, structural partitioning requires that we assign functions to hardware objects *before* knowing about the eventual partition. Since each function is implemented with many objects and each object is shared by many functions, we have a situation where finding a partition requiring a small amount of communication delays and wires between components is very difficult. Intuitively, functional partitioning would seem to solve this problem, since we can partition the functions to minimize communication and then create intercomponent buses to reduce I/O and maximally share hardware objects by the functions assigned to a given component. In this section we describe experiments that demonstrate this concept.

Four VHDL design descriptions are used in the experiments.

(1) *2p-fact* — given a number n, computes the prime numbers p and q such that $n = pq$; if there are no such p and q, outputs zero. It consists of 161 lines of VHDL.

(2) *chinese* — computes the "Chinese Remainder Theorem" [Cormen et al. 1989], which finds the value of x such that it satisfies three congruent equations

$$x = a_1 \text{mod} m_1$$

$$x = a_2 \text{mod} m_2$$

$$x = a_3 \text{mod} m_3$$

such that the moduli, m_1, m_2, and m_3 are pairwise relatively prime. It consists of 168 lines of VHDL.

(3) *8-bits rsa* — a simple version of the RSA encryptography system [Cormen et al. 1989], performing encryption or decryption with public and private keys. It consists of 169 lines of VHDL.

(4) *vol* — a volume-measuring medical instrument controller, which repeatedly receives sonar data from which it computes the volume of an object. It consists of 229 lines of VHDL.

All examples are written at the algorithmic level, as opposed to a state-machine or RT level.

3.1.1 *Functional Partitioning*. We first decompose the specification into a set of functional objects to be assigned to system components. The objects' granularity is that of procedures and variables. Arguments for this granularity, as opposed to finer-granularities such as statements or arithmetic-operations, can be found in Eles et al. [1992]; Thomas et al. [1993]; Gupta et al. [1994]; and Vahid [1995]. Techniques in Vahid [1995] can be used to group a procedure's statements into subprocedures when a procedure is too large.

We then partition the functional objects among two groups, using both automated and manual techniques. We applied the prototype automated partitioner in SpecSyn [Gajski et al. 1994]. SpecSyn creates an internal model, extensively annotates that model with results from estimators, builds complex equations for rapid metric estimation during partitioning, and then inputs the model to a partitioning engine (GPP, General Purpose Partitioner). We used the simulated annealing heuristic in this case. We also independently partition the examples manually, using rough hand-calculated estimates of size, performance and I/O to guide decisions. The automated and manual partitions are usually very close, and on two occasions the automated partitions were slightly better.

After choosing a partition, we manually rewrote the specification as two processes, each process containing a subset of the original procedures and

Table II. Size Library Partial List

Functional units	type	Area(gates)
REG	32-bits	224
REG	1-bits	7
ALU	32-bits	384
LESS	32-bits	190
MUL	32-bits	3230
2-1 MUX	32-bits	97
3-1 MUX	32-bits	193
4-1 MUX	32-bits	288
5-1 MUX	32-bits	384
6-1 MUX	32-bits	576
...	...	...

variables. These processes communicated via global signals, where some signals were used for data and others for control handshaking. The partitioning, specification rewriting, and subsequent simulation of the new specification required about one hour per example, and we typically performed two iterations. (SpecSyn can automatically partition and rewrite the specification in just a few minutes, so when the tool matures, iteration time may be greatly reduced).

3.1.2 *Structural Partitioning*. We first synthesized the entire VHDL specification to a controller block and a datapath of interconnected RT-level objects. We chose to partition the structure at the RT-level, rather than the gate level, in order to obtain reasonably equivalent granularities for functional and structural partitioning. Going to the gate level would have introduced an order of magnitude more objects, which might have prevented partitioning heuristics from finding good solutions, thereby accounting for most of the difference between structural and functional partitioning approaches.

We then converted the RT-level structure to a hypergraph. We created a hypergraph node for each RT object (each register, multiplexor, functional unit, etc.). The controller block was assigned to its own node. We assigned a weight to each node, corresponding to the size of each object when synthesized into a Xilinx library; various object sizes are shown in Table II. We assigned a weight to each hyperedge, corresponding to the number of bits transferred over that edge; memory accesses were encoded as address bits plus data bits. Finally, we input the hypergraph into GPP and applied simulated annealing. The average hypergraph size was 115 nodes. The simulated annealing cooling schedule was chosen so that GPP would run for about 20 minutes to partition the hypergraph. The cost function was a weighted sum of size and I/O violations. We ran four trials for each hypergraph, in which we weighed the size term of the cost function by 1, 5, 15, and 20.

Table III. Xilinx XC4000 FPGAs

Device	XC4008	XC4010/10D	XC4013	XC4025
Gate count	8,000	10,000	13,000	25,0000
Number of IO's	144	160	192	256

Table IV. Functional vs. Structural Partitioning: I/O and Size

Sys.	*Unpartitioned*	*Functional Partition.*			*Structural Partition*			
		no bus	bus	p.c.	Resource Sharing			
					w 1	w 5	w 10	w 15
2p-fact								
Chip 1	99/19697	199/7711	134/7117	102/6875	112/19396	210/15270	140/4854	355/12263
Chip 2	-	102/8398	38/8188	70/7836	14/301	112/4427	236/14843	389/7434
Violation (160/10000)		39/0	0/0	0/0	0/9396	50/5270	76/4843	424/2263
rsa								
Chip 1	132/22614	328/12786	164/11846	100/11604	101/814	169/4250	831/11887	692/13311
Chip 2	-	134/10705	38/9765	102/9249	262/21810	426/18374	832/1073	693/9313
Violation (192/13000)		136/0	0/0	0/0	70/8810	234/5374	1279/0	1001/311
chinese								
Chip 1	99/28471	502/1991	131/17284	99/17042	603/21424	603/21424	603/21424	603/21424
Chip 2	-	325/14211	37/11578	69/11820	699/7047	699/7047	699/7047	699/7047
Violation (256/25000)		315/0	0/0	0/0	790/0	790/0	790/0	790/0
vol								
Chip 1	110/17028	211/12040	191/12008	125/11766	157/13697	217/3772	207/12590	217/12906
Chip 2	-	133/10278	103/10246	135/10488	188/3331	154/13256	285/4438	263/4122
Violation (192/13000)		19/0	0/0	0/0	0/697	25/256	108/0	96/0

3.2 Experiments

We used the MEBS behavioral synthesis tool [Hsu et al. 1994] to synthesize structure in both the functional and structural partitioning approaches. MEBS converts a VHDL process into a finite-state machine (FSM) controller and a connection of RT-level datapath components. MEBS invokes Berkeley's SIS [Brayton and Rudell 1987] tools to implement the controller and can then map the structure into a Xilinx technology library for FPGA implementation.

For both functional and structural partitioning, we used Xilinx XC4000 FPGAs as the implementation components. Size and I/O constraints for these chips are shown in Table III. (We consider the gate listings as usable gates. If usable gates are lower, we can use another package with more usable gates, but the idea is the same.)

Results are shown in Table IV. The *unpartitioned* column shows the I/O and size when synthesizing the entire example into one design (i.e., assuming implementation on a single part). The next three columns show results of functional partitioning. The *no bus* column shows the I/O and size

of each chip after functional partitioning without any shared buses created after partitioning. The *bus* column shows I/O and size when sequential communications between the two chips are assigned to a single bus, thus reducing I/O [Vahid 1997]. The *p.c.* (with port calling) column shows data when accesses to external ports by a particular chip are distributed to the other chip and transmitted over the bus by introducing functions that are called to access ports, allowing better balancing of I/O between the chips [Vahid 1997]. The last four columns show structural partitioning results. The *w1* column represents an even weighing of size and I/O in the cost function used during partitioning. The *w5* represents a weighing of the size term by a factor of 5 more than the I/O term, thus striving for a better balancing of size. The *w10* and *w15* columns represent factors of 10 and 15.

3.3 Analysis

3.3.1 *I/O and Size*. Functional partitioning led to better satisfaction of I/O and size constraints.

Structural partitioning could not satisfy I/O and size constraints at the same time. With an even weighing of those constraints, I/O was satisfied, but sizes were grossly unbalanced and size violations were huge. With heavier weighing of the size constraint, better size balancing was obtained but at the cost of large I/O violations. Functional partitioning nearly satisfied both constraints in all examples. In cases where the I/O constraint was slightly violated, merging communications into buses eliminated the violation [Vahid 1997]. *Note that such merging after partitioning is very difficult, if not impossible, in structural partitioning*—since scheduling communications over wires was already determined during design of the structure. In functional partitioning, communication is still represented as high-level data transfers, so we can merge transfers onto a single bus, introduce arbiters (which is not necessary in our examples, since we only merged sequential communication), and even serialize the data transfers.

We investigated the possibility that resource sharing in the datapath was causing the excessive I/O in structural partitioning. In particular, if numerous registers were being fed to a single input on a register or functional unit, or if a single output of a functional unit was being fed to numerous registers, then that input or output would be a net that would very likely have to be cut during partitioning. On the other hand, if there were fewer shared resources, then there would be fewer such nets. Thus, we resynthesized the examples with no resource sharing—if such a design didn't improve I/O (at the expense of increased size, of course), then no partial resource sharing option likely would. We found only tiny improvements in I/O over all the cases. Therefore, resource sharing did not seem to be the problem with structural partitioning.

When faced with such constraint violations during structural partitioning, the only alternative is to add more parts, leading to much higher-cost designs. Thus we see that functional partitioning can lead to much lower-cost designs by using far fewer parts.

Table V. Functional vs. Structural Partitioning: Performance

Examples	2p-fac	8-bits RSA	chin thm	vol
τ_{sp}	78 ns	315 ns	74 ns	66 ns
t_{sp}	39000 ns	157500 ns	37000 ns	33000 ns
τ_{fp}	47 ns	305 ns	66 ns	54 ns
h	10 clk cycles	7 clk cycles	6 clk cycles	4 clk cycles
t_{fp}	23970 ns	154635 ns	33396 ns	27216 ns
Speed up	1.63	1.02	1.12	1.21

3.3.2 *Performance*. Functional partitioning led to better performance than structural partitioning. To analyze performance, we must look at two factors: (1) the number of clock cycles, n, to execute the specification (say on the average), and (2) the clock period, τ. The performance is then computed as $n \times \tau$. In functional partitioning, we introduce more clock cycles for data transfer, but the clock period stays the same. In structural partitioning, the number of clock cycles stays the same, but we must extend the clock period to account for each intercomponent delay (e.g., 7ns for the Xilinx FPGA) that occurs during any register-to-register transfer. For the examples, we optimistically assumed only one delay (7ns) increase in the clock period. Because of the data-dependent nature of each example, we simply report the case of each example requiring 500 clock cycles. Table V summarizes results. τ_{sp} and τ_{fp} are the clock periods for structural and functional partitioning, respectively, and t_{sp} and t_{fp} are the performance times of each. h is the number of clock cycles for high-level data transfer for the functional partition. Thus, we compute performance as:

$$t_{sp} = n \times \tau_{sp}$$

$$t_{fp} = (n + h) \times \tau_{fp}$$

Functional partitioning led to significant speedups over structural partitioning. Structural partitioning required a longer clock cycle. However, this longer cycle was only partly due to the 7ns added for intercomponent delay. Another factor adding to the longer cycle was the more complicated hardware obtained when synthesizing one large structure instead of two simpler ones. Functional partitioning, on the other hand, required only a few extra clock cycles for handshaked intercomponent data transfer. Speedups ranged from 1.02 to 1.63 over the performance achieved by structural partitioning. Note that if we assume more intercomponent delays or consider examples requiring more than 500 cycles, speedups become even greater. In other words, functional partitioning results in better asymptotic performance than structural partitioning.

Consider asymptotic performance in a more general manner. Consider the ideal situation in which functional partitioning results in *constant* communication clock cycles. This is achieved by avoiding splitting up any loops when we functionally partition the specification. In particular, each

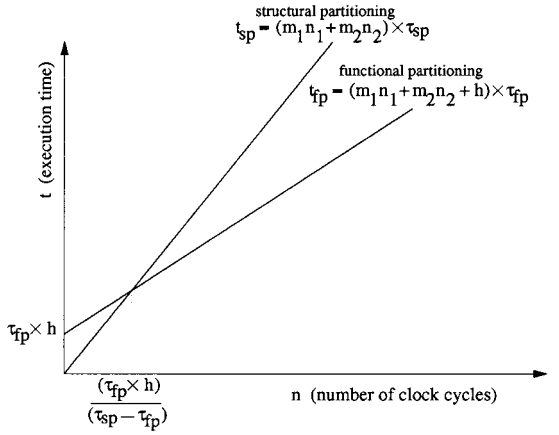

Fig. 6. Asymptotic comparison of functional vs. structural partitioning; (a) constant communication time; (b) linear communication time.

loop of the specification will be completely inside only one part of the partition. On the other hand, constant communication clock cycles may not be possible to achieve because we have to split a loop to balance the partition area. In this case the functional partitioning might result in *linear* communication clock cycles.

To illustrate the asymptotic performance of functional partitioning versus structural partitioning in the case of constant communication time, we consider a specification with two loops, say A and B, where loop A has m_1 iterations, with each iteration taking n_1 clock cycles, and loop B has m_2 iterations, with each taking n_2 clock cycles. Suppose we want to partition the specification between two parts, and we place loop A in part 1 and loop B in part 2. The equations for the specification's execution time after functional partitioning and structural partitioning are shown in Figure

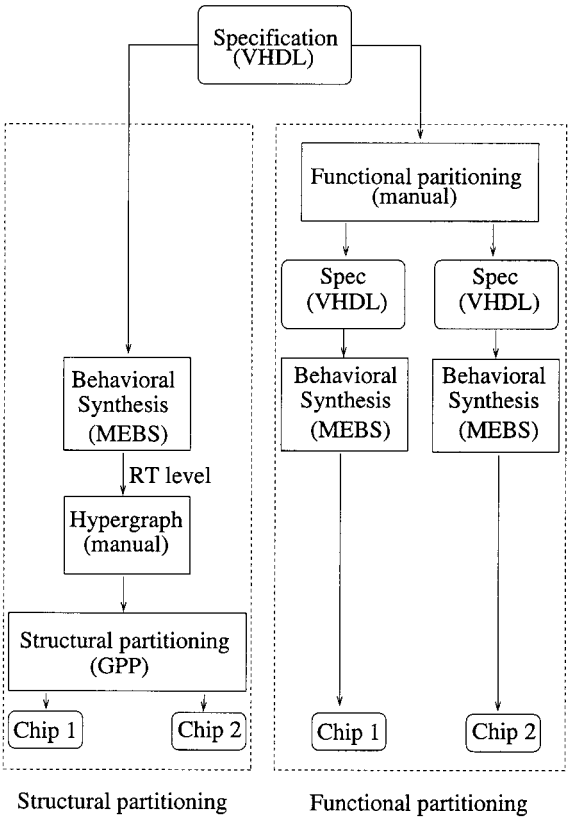

Fig. 7. Method for functional vs. structural partitioning experiments.

6(a), where h is the number of clock cycles for communication after functional partitioning, and τ_{fp} and τ_{sp} are the clock periods obtained after functional and structural partitioning, respectively. If the two loops have only a few iterations, the communication overhead, h, results in a longer execution time from functional partitioning. However, if the loops have many iterations, then the smaller clock period, τ_{fp}, outweighs the communication overhead, leading to shorter execution time. The tradeoff point is $n \geq (\tau_{fp} \times h)/(\tau_{sp} - \tau_{fp})$, where $n = m_1 n_1 + m_2 n_2$.

Linear communication time arises when a loop must itself be split among parts. For example, assume that loop A is larger than the size constraints of parts 1 and 2, so we put half of loop A in one part and half in the other. In this case the execution time equations are shown in Figure 6(b). Note that we now have h cycles of communication overhead for *each* iteration of A. t_{fp} is less than t_{sp} when the indicated relation between τ_{fp} and τ_{sp} holds; in many examples, this relation does hold.

3.3.3 *Notes on the Experimental Method.* A few comments are necessary on our experiments comparing functional and structural partitioning. The

goal of the experiments was to determine whether functional partitioning, only recently enabled, led to a substantially better starting point than structural partitioning, perhaps an order of magnitude better—and thus worthy of investigation and development. To achieve a fair comparison, we had to control other variables that could have led to differences in the results, such as heuristic specialization and quantity of objects.

Heuristic specialization is the degree to which a heuristic is advanced for either structural or functional partitioning. Comparing the most advanced heuristics for each would not be a fair, as structural partitioning approaches have had decades to develop, while functional partitioning methods are just appearing, though advances are already being reported. For example, much work in structural partitioning has focused on replication of gates [Johannes 1996], while replication techniques for functional partitioning are just appearing [Vahid 1997]. To control this variable, we used the GPP partitioning engine for both approaches, rather than a specialized netlist partitioner for structural partitioning. We point out that the number of I/O achieved using GPP for structural partitioning roughly agrees in magnitude to results described in structural partitioning literature, such as Tessier et al. [1994]. In particular, in the above research, structural partitioning between five or ten thousand gate parts requires between several hundred and a thousand I/O (thus in turn requiring many more parts than the gate count implies). While logic replication techniques report good I/O reductions (18% in Hwang et al. [1995]), these reductions are small compared to those achieved by functional partitioning.

Quantity of objects refers to the number of objects that the partitioning heuristic must deal with. Partitioning heuristics are nonoptimal, and thus the number of objects can greatly affect the quality of the results. Since functional partitioning of coarse-grained functions yields hundreds of objects, we chose to partition a structure at the RT level, which also yields hundreds of objects. If, instead, we partitioned structure at the gate level, there would be thousands or tens of thousands of objects, likely leading to inferior results. Perhaps a gate-level approach is better for comparison of a statement or arithmetic-level functional partitioning approach (which we do not advocate).

4. PARTITIONING FOR SYNTHESIS TOOL PERFORMANCE

4.1 Method

In this section, we evaluate improvements in synthesis tool performance gained with functional partitioning. Note that the resulting partitioned design may still be implemented on a single package.

We evaluate synthesis tool performance using three factors. (1) *Synthesis time*: The CPU time (on a Sparc 10) required for the synthesis tool to convert the functional specification into structure. As shown in Figure 8, we compare the time for synthesizing the entire VHDL specification with the sum of the times for synthesizing each specification after partitioning.

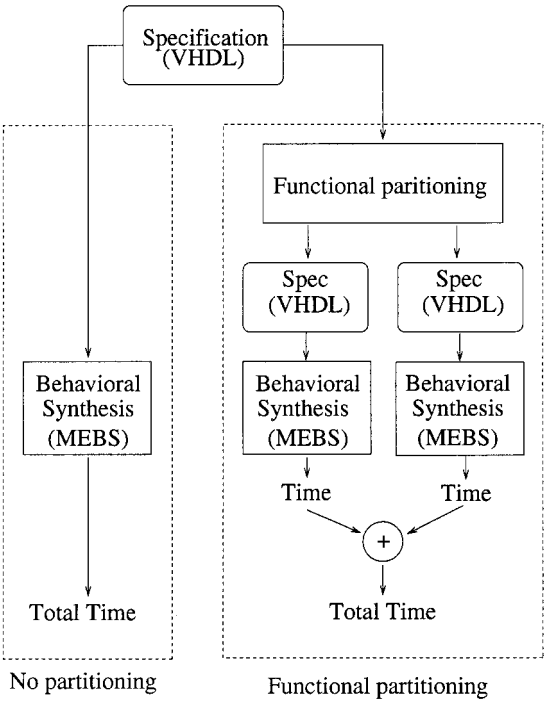

Fig. 8. Method for synthesis tool evaluation experiments.

(2) *Output size*: The total size of the output structure, measured in equivalent gates using the Xilinx XC4000 technology library. The sizes are compared in a manner identical to synthesis times. (3) *Memory use*: The maximum amount of memory used at any time by the synthesis tool.

Our experiments show that memory use is linearly proportional to the size of the input, so we do not report memory use in subsequent tables. In our examples, the maximum memory used during synthesis of any one example is 300 Mb. Since this amount of memory is much less than our available memory, partitioning did not yield significant improvements in memory use. However, in cases where available memory is scarce, partitioning could ensure that the maximum memory is not exceeded.

4.1.1 *Examples*. We use the same examples as in the previous section. However, we want to measure the effect of input size on tool performance. But if we compare tool performance on different examples of different sizes, we cannot determine whether the variations in performance result from the different sizes factor or from other factors arising from the different computations in each example. For example, a large example might require more synthesis time than a small example, not because of its size but because of some small portion of the large example requiring substantial scheduling and binding. To eliminate such additional factors, we created large examples by *duplicating* smaller examples a number of times. The

Table VI. Synthesis Time Speedups (Optimal Mode)

2P-FACT						
Dup.	*Unpart.*	*Functional part.*			*Spd. Up*	
		Part 1	Part 2	Total	S.	P.
1	00:48	00:32	00:05	00:37	1.3	1.5
2	19+ hours	01:10	00:15	01:25	15.2	17.3
3	19+ hours	01:21	00:31	01:52	12.5	15.7
4	19+ hours	02:18	00:43	03:01	6.3	8.7
CHINESE THM						
Dup.	*Unpart.*	*Functional part.*			*Spd. Up*	
		Part 1	Part 2	Total	S.	P.
1	05:20	00:23	00:31	00:54	5.9	10.3
2	07:28	02:22	01:03	03:25	2.4	3.3
3	15:19	03:09	02:23	05:32	2.9	4.9
4	16+ hours	03:25	04:58	07:49	2.1	3.2
VOL						
Dup.	*Unpart.*	*Functional part.*			*Spd. Up*	
		Part 1	Part 2	Total	S.	P.
1	00:15	00:06	00:02	00:08	1.9	2.5
2	01:03	00:12	00:05	00:17	3.7	5.25
3	05:26	00:35	00:22	00:57	5.7	9.3
4	09:06	01:29	01:06	02:35	3.5	6.1

duplication method follows. First, we duplicated the ports, variables, and procedures $N - 1$ times, creating new identifiers for each duplicated object. Then we duplicated the process' statements $N - 1$ times, where each duplication accessed its own copy of ports, variables, and procedures. We created four versions of each example, corresponding to an N of 1 (the original version), 2, 3, and 4 (the largest version of the example, roughly four times bigger than the original version).

4.1.2 *Synthesis*. We again used MEBS to perform synthesis. The MEBS synthesis tool divides behavior synthesis into two subtasks: high-level synthesis and logic synthesis. High-level synthesis involves a sequence of subtasks: compilation, scheduling, allocation, and binding. MEBS' logic synthesis has three modes: no optimization, fast, and optimal. In this experiment, we use only the fast mode and the optimal mode.

The number of hardware units, with respect to the technology library, and synthesis times are reported upon the completion of the synthesis process.

4.2 Results

Table VI compares the results of synthesizing the unpartitioned and functionally partitioned examples using optimal mode logic synthesis. The

Table VII. Synthesis Time Speedups (Fast Mode)

2P-FACT						
Dup.	*Unpart.*	*Functional part.*			*Spd. Up*	
		Part 1	Part 2	Total	S.	P.
1	890s	12s	3s	15s	59.3	74.2
2	2675s	23s	9s	32s	83.6	116.3
3	3400s	786s	12s	798s	4.3	4.3
4	5500s	1414s	120s	1534s	3.5	3.8
8-bits RSA						
Dup.	*Unpart.*	*Functional part.*			*Spd. Up*	
		Part 1	Part 2	Total	S.	P.
1	1216s	7s	2s	9s	135.1	173.7
2	3759s	71s	4s	75s	52.9	52.9
3	4937s	610s	8s	618s	8.1	8.1
4	7597s	1314s	9s	1323s	5.8	4.4
VOL						
Dup.	*Unpart.*	*Functional part.*			*Spd. Up*	
		Part 1	Part 2	Total	S.	P.
1	13s	11s	3s	14s	0.9	1.2
2	1260s	20s	79s	99s	12.7	15.9
3	1364s	35s	970s	1005s	1.3	1.4
4	1694s	51s	1138s	1189s	1.4	1.5

Dup column represents the number of duplications for a given example, as described earlier. The *Unpart* column represents the CPU times, in hours and minutes, for synthesizing the unpartitioned example. The *Part1* and *Part2* columns are the CPU times for synthesizing each part of the functionally partitioned specification, and the *Total* column is the sum of those two times. The *S* column shows the speedup obtained by partitioning. The *P* shows the speedup if we assume that the two parts of the partitioned specification can be synthesized in parallel. (Only three of the four examples are shown in each table, as the synthesis tool's limitations at the time of the experiments prevented completion of some duplicated examples.)

Table VII is identical to Table VI, except that it shows results using the fast logic synthesis mode. Finally, Table VIII shows size results for the fast logic synthesis mode. The last column of the table indicates the ratio of the unpartitioned design size over the total size of the partitioned design from the fast mode. (We do not have complete size data for the optimal synthesis mode because several synthesis jobs did not complete in the 19 hours we allocated for each example).

4.3 Analysis

4.3.1 *Synthesis Time*. Functional partitioning yields very substantial and practical reductions in synthesis times. When using optimal logic

Table VIII. Size Outputs

2P-FACT					
Dup.	*Unpart.*	*Functional part.*			*Ratio*
		Part 1	Part 2	Total	
1	13312	6347	7431	13778	1.11
2	24271	8410	10279	18689	0.77
3	34160	11431	13628	25059	0.77
4	45033	14717	15219	29936	0.67
8-bits RSA					
Dup.	*Unpart.*	*Functional part.*			*Ratio*
		Part 1	Part 2	Total	
1	17275	14200	3558	17758	1.11
2	25655	21640	6525	28165	1.11
3	33435	28994	10814	398098	1.25
4	43182	34635	11690	46325	1.11
VOL					
Dup.	*Unpart.*	*Functional part.*			*Ratio*
		Part 1	Part 2	Total	
1	11996	11884	6856	18740	1.6
2	19489	19449	9861	29310	1.4
3	27911	23043	13989	37032	1.25
4	35661	29306	17806	47112	1.25

synthesis mode, speedups are excellent, sometimes over 10. We observe reductions from over 8 hours down to just 1 to 2 hours, thus converting an overnight job into one that can be done during a work day. When using fast logic synthesis mode, we find even larger speedups, in some cases near 100, although unpartitioned specification synthesis time was reduced compared to optimal mode from roughly 10 hours to 1 hour. Note that as the example size increased (denoted by the duplication amount), the speedups tends to decrease. Thus, for large examples, it would likely be beneficial to partition the specification into more than just two parts.

The observed improvements are likely due to polynomial-time heuristics in the synthesis tools. Partitioning the specification thus has a nonlinear effect on the tool's CPU time. Optimal logic-synthesis time is dominated by control unit synthesis, and thus strongly affected by the number of states and transitions, whereas fast logic-synthesis time is dominated by data-path binding and thus is affected by the number of operations.

4.3.2 *Design Size*. One concern is that partitioning will lead to much larger designs caused by inability to share functional units across parts and to extra hardware for communication between parts. However, Table VIII shows that there is usually only a slight increase in size, roughly 10-20%. The biggest increase occurs in the *vol* example, since instead of

just one multiplier we need two multipliers, one for each part. In many cases, the sizes are nearly equal, and in some cases, there was actually a *decrease* in size, most likely attributable to simpler control logic and multiplexing.

Finally, we note that synthesis tool documentation often encourages specification writers to functionally partition the input manually, by writing processes that are no larger than some specified criteria.

4.3.3 *Predictor*. The number of states/transitions and functional units seem to predict the synthesis time. When using optimal logic synthesis, synthesis time is dominated by synthesis of the control unit. The number of states and transitions increases the complexity of control units. Therefore, we can use the number of states and transitions as the predictor for synthesis time in optimal mode. On the other hand, when using fast logic synthesis, most time was spend performing data path binding (in MEBS). Thus the number of operations becomes the predictor for synthesis time.

5. FUTURE WORK

Functional partitioning can yield big advantages. But the problem is a difficult one, and much research is needed in several key areas before automated functional partitioners become truly practical.

First, fast and accurate estimators must be developed. These estimators will likely need to be closely integrated with commercial synthesis tools, in order to provide accurate prediction of synthesis results.

Second, good partitioning and transformation heuristics must be developed. We recently extended the Kernighan/Lin heuristic, which had proven itself fast, yielding good results for structural partitioning, to the problem of functional partitioning [Vahid and Le 1997]. We are also investigating transformations that could lead to improved results, such as cloning of shared procedures [Vahid 1997] or parallelization of sequential procedure calls. Integration of partitioning and transformation should also be examined.

Third, techniques for interfacing functions on different components must be developed. We are currently developing communication libraries for such a purpose [Vahid and Tauro 1997]. Techniques for generating a highly readable and refined specification must also be developed.

Many of the above problems were initially addressed in Gajski's SpecSyn tool [Gajski 1994] at UC Irvine.

6. CONCLUSIONS

We have shown the importance of functional partitioning in a synthesis environment. Functionally partitioning a specification among hardware blocks yields far better satisfaction of I/O and size constraints on those blocks than does the current approach of structural partitioning, while also yielding improvements in performance . Functionally partitioning a system before synthesis can yield an order of magnitude improvement in synthesis

runtimes. These findings suggest the need for further investigation and development of automated functional partitioning tools, in order to meet packaging constraints, improve synthesis performance, as well as perform hardware/software partitioning. Such tools can substantially improve the usefulness of automated synthesis environments.

References

ANTONIAZZI, S., BALBONI, A., FORNACIARI, W., AND SCIUTO, D. 1994. A methodology for control-dominated systems codesign. In *Proceedings of the International Workshop on Hardware-Software Co-Design*, 2–9.

BRAYTON, R., RUDELL, R., SANGIOVANNI-VINCENTELLI, A., AND WANG, A. 1987. MIS: A multiple-level logic optimization system. *IEEE Trans. Comput.-Aided Des. Integr. Circuits 6* (Nov.), 1062–1080.

CAMPOSANO, R. AND BRAYTON, R. 1987. Partitioning before logic synthesis. In *Proceedings of the International Conference on Computer-Aided Design* (ICCAD).

CAMPOSANO, R. AND VAN EIJNDHOVEN, J. 1987. Partitioning a design in structural synthesis. In *Proceedings of the International Conference on Computer Design*.

CHEN, Y., HSU, Y., AND KING, C. 1992. MULTIPAR: Behavioral partitioning for synthesizing application-specific multiprocessor architectures. In *Proceedings of the European Conference on Design Automation*, 14–18.

CHEN, Y., HSU, Y., AND KING, C. 1994. MULTIPAR: Behavioral partition for synthesizing multiprocessor architectures. *IEEE Trans. Very Large Scale Integr. Syst. 2*, 1 (Mar.), 21–32.

CORMEN, T. T., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

ELES, P., PENG, Z., AND DOBOLI, A. 1992. VHDL system-level specification and partitioning in a hardware/software co-synthesis environment. In *Proceedings of the International Workshop on Hardware-Software Co-Design*, 49–55.

ELES, P., PENG, Z., KUCHCINSKI, K., AND DOBOLI, A. 1996. Hardware-software partitioning with iterative improvement heuristics. In *Proceedings of the International Symposium on System Synthesis*, 71–76.

ERNST, R., HENKEL, J., AND BENNER, T. 1994. Hardware-software cosynthesis for microcontrollers. *IEEE Des. Test* (Dec.), 64–75.

FIDUCCIA, C. AND MATTHEYSES, R. 1982. A linear-time heuristic for improving network partitions. In *Proceedings of the Conference on Design Automation*.

GAJSKI, D. D., DUTT, N. D., WU, A. C.-H., AND LIN, S. Y.-L. 1992. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, Hingham, MA.

GAJSKI, D. D., VAHID, F., NARAYAN, S., AND GONG, J. 1994. *Specification and Design of Embedded Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ.

GUPTA, P., CHEN, C.-T., DESOUZA-BATISTA, J. C., AND PARKER, A. C. 1994. Experience with image compression chip design using unified system construction tools. In *Proceedings of the 31st Annual Conference on Design Automation* (DAC'94, San Diego, CA, June 6–10, 1994). ACM Press, New York, NY, 250–256.

GUPTA, R. K. AND DE MICHELI, G. 1993. Hardware-software cosynthesis for digital systems. *IEEE Des. Test 10*, 3 (Sept.), 29–41.

GUPTA, R. AND DEMICHELI, G. 1990. Partitioning of functional models of synchronous digital systems. In *Proceedings of the International Conference on Computer-Aided Design*, 216–219.

GUPTA, R. AND DEMICHELI, G. 1992. System-level synthesis using re-programmable components. In *Proceedings of the European Conference on Design Automation*, 2–7.

HAREL, D. 1987. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program. 8*, 3 (June 1), 231–274.

HOARE, C. 1978. Communicating sequential processes. *Commun. ACM 21*, 8, 666–677.

HSU, Y., LIU, T., TSAI, F., LIN, S., AND YU, C. 1994. Digital design from concept to prototype in hours. In *Proceedings of the Asia-Pacific Conference on Circuits and Systems*.

HWANG, L. AND GAMAL, A. E. 1995. Min-cut replication in partitioned networks. *IEEE Trans. CAD 14* (Jan.), 96–106.

ISMAIL, T. B., O'BRIEN, K., AND JERRAYA, A. 1994. Interactive system-level partitioning with PARTIF. In *Proceedings of the European Conference on Design Automation* (EURO-DAC '94, Grenoble, France, Sept. 19–23, 1994). IEEE Computer Society Press, Los Alamitos, CA.

JOHANNES, F. 1996. Partitioning of VLSI circuits and systems. In *Proceedings of the 33rd Conference on Design Automation*.

KALAVADE, A. AND LEE, E. 1994. A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem. In *Proceedings of the International Workshop on Hardware-Software Co-Design*, 42–48.

KERNIGHAN, B. AND LIN, S. 1970. An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J.* (Feb.).

KIRKPATRICK, S., GELATT, C. D., JR., AND VECCHI, M. P. 1983. Optimization by simulated annealing. *Science 220*, 4598 (May), 671–680.

KIRKPATRICK, Y. AND CHENG, C. 1991. Ratio cut partitioning for heirarchical designs. *IEEE Trans. CAD 10*, 7 (July), 911–921.

KRISHNAMURTHY, B. 1984. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Trans. Comput.* (May).

KUCUKCAKAR, K. AND PARKER, A. 1991. CHOP: A constraint-driven system-level partitioner. In *Proceedings of the Conference on Design Automation*, 514–519.

LAGNESE, E. D. AND THOMAS, D. E. 1991. Architectural partitioning of system level synthesis of integrated circuits. *IEEE Trans. CAD 10*, 7 (July), 847–860.

LEE, E. AND MESSERSCHMITT, D. 1987. Synchronous data flow. *Proc. IEEE 75*, 9, 1235–1245.

MCFARLAND, M. AND KOWALSKI, T. 1990. Incorporating bottom-up design into hardware synthesis. *IEEE Trans. CAD 9*, 9 (Sept.).

SANCHIS, L. A. 1989. Multiple-way network partitioning. *IEEE Trans. Comput. 38*, 1 (Jan.), 62–81.

SECHEN, C. AND CHEN, D. 1988. An improved objective function for mincut circuit partitioning. *IEEE Trans. CAD*.

TESSIER, R., BABB, J., DAHL, M., HANONO, S., AND AGARWAL, A. 1995. The virtual wires emulation system: A gate-efficient asic prototyping environment. In *Proceedings of the Third International ACM Symposium on Field-Programmable Gate Arrays* (FPGA '95, Monterey, CA, Feb. 12–14, 1995). ACM Press, New York, NY.

THOMAS, D., ADAMS, J., AND SCHMIT, H. 1993. A model and methodology for hardware/software codesign. *IEEE Des. Test*, 6–15.

VAHID, F. 1997. I/O and performance tradeoffs with the FunctionBus during multi-FPGA partitioning. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays* (FPGA), 27–34.

VAHID, F. 1997. Port calling: A transformation for reducing i/o during multi-package functional partitioning. In *Proceedings of International Symposium on System Synthesis*.

VAHID, F. 1997. Procedure cloning: A transformation for improved system-level functional partitioning. In *Proceedings of the European Conference on Design and Test*, 487–492.

VAHID, F. 1995. Procedure exlining: A transformation for improved system and behavioral synthesis. In *Proceedings of the Eighth International Symposium on System Synthesis* (Cannes, France, Sept. 13–15, 1995). ACM Press, New York, NY, 84–89.

VAHID, F. AND GAJSKI, D. D. 1995. Incremental hardware estimation during hardware/software functional partitioning. *IEEE Trans. Very Large Scale Integr. Syst. 3*, 3 (Sept.), 459–464.

VAHID, F. AND GAJSKI, D. 1995. SLIF: A specification-level intermediate format for system design. In *Proceedings of the European Conference on Design and Test* (EDTC), 185–189.

VAHID, F. AND GAJSKI, D. D. 1992. Specification partitioning for system design. In *Proceedings of the 29th ACM/IEEE Conference on Design Automation* (DAC '92, Anaheim, CA, June 8-12). IEEE Computer Society Press, Los Alamitos, CA, 219–224.

VAHID, F. AND LE, T. 1997. Extending the Kernighan/Lin heuristic for hardware and software functional partitioning. *J. Des. Autom. Embedded Syst. 2*, 2, 237–261.

VAHID, F. AND LE, T. 1996. Towards a model for hardware and software functional partitioning. In *Proceedings of the International Workshop on Hardware-Software Co-Design*, 116–123.

VAHID, F., NARAYAN, S., AND GAJSKI, D. 1995. SpecCharts: A VHDL front-end for embedded systems. *IEEE Trans. CAD*, 694–706.

VAHID, F. AND TAURO, L. 1997. An object-oriented communication library for hardware-software co-design. In *Proceedings of the International Workshop on Hardware-Software Co-Design*, 81–86.

XIONG, X., BARROS, E., AND ROSENSTIEL, W. 1994. A method for partitioning UNITY language in hardware and software. In *Proceedings of the European Conference on Design Automation* (EURO-DAC '94, Grenoble, France, Sept. 19–23, 1994). IEEE Computer Society Press, Los Alamitos, CA, 220–225.

IEEE STANDARDS OFFICE, 1988. *IEEE Standard VHDL Language Reference Manual*. IEEE Standards Office, New York, NY.

Received: January 1996; revised: October 1996; accepted: July 1997