A Comparison of Functional and Structural Partitioning

Frank Vahid Thuy Dm Le Yu-chin Hsu
Department of Computer Science
University of California, Riverside, CA 92521
vahid@cs.ucr.edu

Abstract

Incorporating functional partitioning into a synthesis methodology leads to several important advantages. In functional partitioning, we first partition a functional specification into smaller sub-specifications 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 that of greatly improved partitioning among a given set of packages with size and I/O constraints, such as FPGA's or ASIC blocks, resulting in better performance and far fewer required packages. A second advantage is that of greatly reduced synthesis runtimes. We present results of experiments demonstrating these important advantages, leading to the conclusion that further research focus on functional partitioning can lead to improved quality and practicality of synthesis environments.

1 Introduction

Functional specifications, consisting of a machine-readable program-like description of a system's desired behavior, are becoming commonly available with digital systems. They can be input to simulators for early verification of behavior, as well as to synthesis tools for automatic structural design. In addition, those specifications can be functionally partitioned to solve numerous problems, including exploring hardware/software tradeoffs, satisfying hardware packaging constraints, and reducing synthesis runtimes.

Recent research has focused on the first problem of exploring tradeoffs of cost and performance through functional partitioning among software and custom-hardware processor components [1, 2, 3, 4, 5, 6, 7]. The latter problems are also important, but their functional partitioning solutions have received far less attention.

The problem of satisfying hardware packaging constraints, such as size and I/O (input/output pin) constraints on ASIC or Field-Programmable Gate Array chips (FPGA's), has received extensive attention by researchers for over two decades [8]. The focus, though, has been on partitioning already-designed structure. Such structural partitioning is I/O dominated because a typical package's I/O is usually relatively scarce compared to gates, leading to more packages, power, and execution delay [9]. While new techniques multiplex wires to reduce I/O [9], the problem is still very hard to solve structurally. Designers have long solved this problem manually by performing functional partitioning, where a system's functions are first partitioned among packages, and then each package's functions are implemented. 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 [10, 11, 12, 13, 14]. This paper provides empirical results supporting that hypothesis.

Intuitively, functional partitioning excels 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. 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 drawbacks. First, functional partitioning must be guided by estimates of size, I/O and performance for possibly thousands of examined partitions, but obtaining good size and performance estimates quickly 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 can help alleviate this drawback [15]. Second, a functionally partitioned system often (though not always, as we shall see) uses more gates, since hardware objects aren't shared by functions on different parts. However, since partitioning is often I/O dominated, this increase is often insignificant.

Another problem solvable with functional partitioning is reducing synthesis tool runtime. In particular, one large process in a specification can result in excessive tool CPU time and memory use and can lead to an inefficient design. By functionally partitioning the specification into smaller ones, and inputing each to the synthesis tool independently, runtimes can be reduced by an order of magnitude. An analogous problem and solution were presented in [16] for partitioning logic equations before logic synthesis; in this paper, we describe results of functionally partitioning a specification before behavioral synthesis.

This paper is organized as follows. We describe experiments in Section 2 demonstrating functional partitioning's advantages for satisfying package constraints, and in Section 3 for reducing synthesis runtimes. We discuss future work in Section 4, and provide conclusions in Section 5.

0-8186-7563-2/96 $5.00 © 1996 IEEE

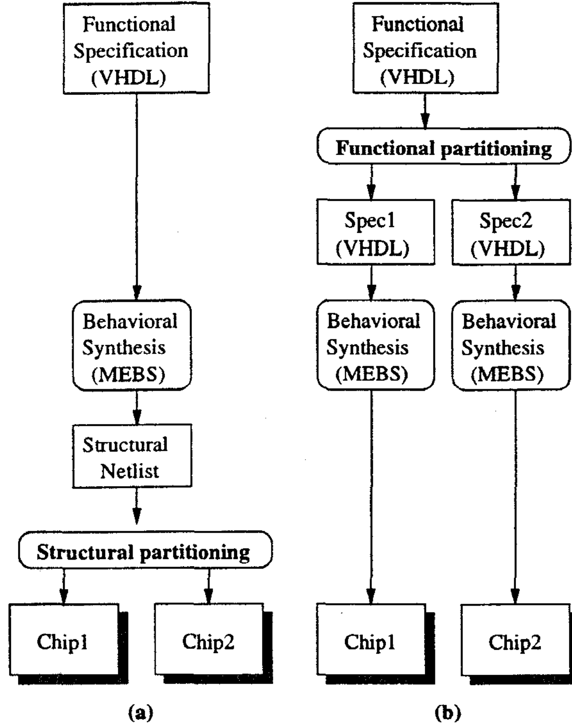

Figure 1: Partitioning approaches: (a) structural, (b) functional.

2 Partitioning for packages

We describe our functional and structural partitioning approaches, as illustrated in Figure 1, and compare the resulting system performance, gates, and I/O for several examples partitioned among two FPGA's.

2.1 Examples

Four VHDL design descriptions were used in the experiments. *2p-fact* is a decryption example, which attempts to find two prime numbers who's product equals some given value. *Chinese* is the application of "Chinese Remainder Theorem" [17], which finds the value of x such that it satisfies three congruent equations. *8-bits rsa* is a simple version of the RSA cryptography system [17], which performs encryption or decryption with the given public and private keys. *Vol* is a volume-measuring medical instrument controller, which receives sonar data from which an object's volume is computed. All examples were written with algorithmic-level VHDL, as opposed to a state-machine or RT level, and their sizes were 161, 168, 169, and 229 lines.

2.2 Functional partitioning

We first decomposed the specification into a set of functional objects to be assigned to system components. The object granularity was procedures and variables. Arguments for this granularity, as opposed to finer-granularities like statements, can be found in [2, 5, 18, 19]. Techniques in [19] can be used to group a procedure's statements into sub-procedures when the user-written procedures would be too coarse-grained.

We then partitioned the functional objects among two groups, using both automated and manual techniques. We applied the prototype automated partitioner in SpecSyn [20]. 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 use the simulated annealing heuristic in this case. We also independently partitioned the examples manually, using rough hand-calculated estimates of size, performance and I/O to guide decisions. The automated and manual partitions were usually very close.

After choosing a partition, we manually rewrote the specification as two processes, each process containing a subset of the original procedures and 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 1 hour per example, and we typically performed two iterations. (SpecSyn can automatically partition and rewrite the specification in just a few minutes, so iteration times could be greatly reduced).

2.3 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 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 caused partitioning heuristics to find inferior 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, mux, 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 1. We assigned a weight to each hyperedge, corresponding to the number of bits being 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. The cost function was a weighted sum of size and I/O violations. We ran 4 trials for each hypergraph, in which we weighed the size term of the cost function by 1, 5, 15 and 20.

2.4 Experiments

We used the MEBS behavioral synthesis tool [21] to synthesize structure in both the functional and structural partitioning approaches. MEBS converts a VHDL

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
...	...	...

Table 1: Partial size library

process into a finite-state machine (FSM) controller and a connection of RT-level datapath components. MEBS invokes Berkeley's SIS [22] tool to implement the controller, and then maps the structure into a Xilinx technology library for FGPA implementation.

For both functional and structural partitioning, we used Xilinx XC4000 FPGA's as the implementation components. Size and I/O constraints for these chips are shown in Table 2.

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

Table 2: Xilinx XC4000 FPGA's

Results are shown in Table 3. The *unpartitioned* column shows the I/O and size when synthesizing the entire example into one design (i.e., assuming implementation on a single chip). 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 additional 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. The *p.d.* (ports distributed) column shows size and I/O when accesses to external ports by a particular chip are distributed to the other chip and transmitted over the bus, allowing better balancing of I/O between the chips. The last four columns show structural partitioning results. The *w 1* column represents an even weighing of size and I/O in the cost function used during partitioning. The *w 5* 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 *w 10* and *w 15* columns represent factors of 10 and 15.

2.5 Analysis

2.5.1 I/O and size

Functional partitioning led to much 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 in the cost function, 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. *Note that such merging after partitioning is very difficult during structural partitioning*, since scheduling of communications over wires was already determined during design of the structure. In functional partitioning, communication still represents high-level data transfers, so we can merge transfers onto a single bus, introduce arbiters (which was not necessary in our examples since we only merged sequential communication), and even serialize the data transfers.

We investigated the possibility that the excessive I/O achieved during structural partitioning was caused by too much sharing of resources in the datapath. We re-synthesized the designs, telling MEBS not to share any registers or functional units in the datapath. Obviously this resulted in a much larger total size (more register and functional units, though fewer muxes). After applying structural partitioning, we found only minor improvements in I/O satisfaction. For example, in the *2p-fact* example, the packaging violation was 0 pins and 14043 gates (or 0/14043).

When faced with such constraint violations during structural partitioning, the alternatives are to add more FPGA's or use more expensive FPGA's with greater capacities, leading to much higher-cost designs. Thus, we see that functional partitioning can lead to much lower-cost designs by using fewer or cheaper FPGA's.

2.5.2 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., 7 ns for the Xilinx FPGA) that occurs during any register-to-register transfer. For the examples, we optimistically assumed only one delay (7 ns) increase in the clock period. Table 4 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 can 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 7 ns added for intercomponent delay. A second factor leading to the longer cy-

Example	*Unpartitioned*	*Functional partitioning*			*Structural partitioning*			
		I/O reduction technique			Resource sharing			
2p-fact		none	buses	port dist.	w 1	w 5	w 10	w 15
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	832/10737	693/9313
Violation (192/13000)		136/0	0/0	0/0	70/8810	234/5374	1279/0	1001/311
chinese								
Chip 1	99/28471	502/19917	131/17284	99/17042	184/819	208/4014	212/4063	212/4063
Chip 2	–	325/14211	37/11578	69/11820	824/27652	912/24457	916/24408	916/24408
Violation (256/25000)		315/0	0/0	0/0	568/2652	656/0	660/0	660/0
vol								
Chip 1	110/17028	211/12040	191/12008	125/11766	183/14792	174/13907	223/3366	315/4003
Chip 2	–	133/10278	103/10246	135/10488	168/2236	196/3121	165/13662	168/13035
Violation (192/13000)		19/0	0/0	0/0	0/1792	4/907	31/662	123/25

Table 3: Functional vs. structural partitioning

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

Table 4: Performance comparison

cle 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 inter-component data transfer. Speedups ranged from 1.02 to 1.63 over the performance achieved by structural partitioning. More intercomponent delays after structural partitioning, which are commonly the case, would lead to even greater speedups.

3 Partitioning for synthesis

In this section, we evaluate synthesis tool performance improvements 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 behavioral specification into structure. We compare the time for synthesizing the entire VHDL specification with the sum of the times for synthesizing each specification after partitioning. (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 showed that memory use was 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 was 300 Mb. Since this amount of memory was much less than our available memory, partitioning did not yield significant improvements with regard to memory use. However, in cases where available memory is scarce, partitioning could ensure that the maximum memory amount was not exceeded.

3.1 Examples

We used the same examples as above. However, in this case, we needed to measure the effect of input size on tool performance. If we compare tool performance on different examples of different sizes, we can not determine whether the variations in performance result from the different sizes or from other factors. For example, one example might require more synthesis time than another not because of its size, but because of some piece requiring substantial scheduling and binding. To eliminate such additional factors, we created large examples by *duplicating* smaller examples a number of times. The duplication method was as 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).

3.2 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's logic synthesis has three modes: no optimization, fast, and optimal. In this experiment, we use the fast mode and the optimal mode.

2p-fact						
Dup.	Unpartit.	Functional partitioning			Speedup	
		Part1	Part2	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						
Dup.	Unpartit.	Functional partitioning			Speedup	
		Part1	Part2	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	08:23	1.9	3.2
Vol						
Dup.	Unpartit.	Functional partitioning			Speedup	
		Part1	Part2	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

Table 5: Synthesis times (optimal mode)

2p-fact						
Dup.	Unpartit.	Functional partitioning			Speedup	
		Part1	Part2	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
RSA						
Dup.	Unpartit.	Functional partitioning			Speedup	
		Part1	Part2	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.	Unpartit.	Functional partitioning			Speedup	
		Part1	Part2	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

Table 6: Synthesis times (fast mode)

3.3 Results

Table 5 provides a comparison of the results of synthesis of the unpartitioned and functionally partitioned examples, using optimal mode logic synthesis. The *Dup* column represents the number of duplications for a given example, as described earlier. The *Unpartit* 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 the experiments were performed prevented completion of some duplicated examples.)

Table 6 is identical to Table 5 except that it shows results using the fast logic synthesis mode. Finally, Table 7 shows the size results. The last column of that

2p-fact					
Dup.	Unpartit.	Functional partitioning			Ratio
		Part1	Part2	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
RSA					
Dup.	Unpartit.	Functional partitioning			Ratio
		Part1	Part2	Total	
1	17275	14200	3558	17758	1.11
2	25655	21640	6525	28165	1.11
3	33435	28994	10814	39808	1.25
4	43182	34635	11690	46325	1.11
Vol					
Dup.	Unpartit.	Functional partitioning			Ratio
		Part1	Part2	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

Table 7: Size outputs

table indicates the ratio of the unpartitioned design size over the total size of the partitioned design from the fast mode.

3.4 Analysis

3.4.1 Synthesis time

Functional partitioning yields very substantial and practical reductions in synthesis times. When using optimal logic synthesis mode, speedups were excellent, sometimes over 10. We observe reductions in some cases from over 8 hours down to just 1-2 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 synthesis time of the unpartitioned specification was reduced compared with optimal mode from roughly 10 hours to 1 hour.

Note that as the example size increased (denoted by the duplication amount), the speedups tended to decrease. In such cases, 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 non-linear effect on the tool's CPU time. Optimal logic-synthesis times were dominated by control unit synthesis and thus were strongly affected by the number of states and transitions, whereas fast logic-synthesis times were dominated by datapath binding and thus were affected by the number of operations.

3.4.2 Design size

One concern is that functional partitioning would lead to much larger designs caused by the inability to share functional units across parts and to extra hardware for communication between parts. However, Table 7 shows that there is usually only a slight increase in size, roughly 10-20% (which is quite negligible considering that structural partitioning cannot utilize even more

gates since it is I/O dominated). The biggest increase occurred in the *vol* example, since instead of just one multiplier we needed two multipliers, one for each part. In many cases, the sizes were 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 Future work

Functional partitioning can yield big advantages. However, 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 are presently extending the Kernighan/Lin heuristic, which has proven to be fast and yield good results for structural partitioning, to the problem of functional partitioning. We are also investigating transformations that could lead to improved results, such as cloning of shared procedures or parallelization of sequential procedure calls. The integration of partitioning and transformation must also be examined.

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

Most of the above problems were directly addressed in Gajski's SpecSyn tool [20] from UC Irvine.

5 Conclusions

We have shown the importance of functional partitioning in a synthesis environment. Functionally partitioning a specification among hardware packages yields far better satisfaction of I/O and size constraints than does the current approach of structural partitioning, while also yielding much better performance. Functionally partitioning a system before synthesis can yield an order of magnitude improvement in synthesis runtimes. These findings suggest the need for significant research focus on functional partitioning.

References

[1] S. Antoniazzi, A. Balboni, W. Fornaciari, and D. Sciuto, "A methodology for control-dominated systems codesign," in *International Workshop on Hardware-Software Co-Design*, pp. 2–9, 1994.

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

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

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

[5] D. Thomas, J. Adams, and H. Schmit, "A model and methodology for hardware/software codesign," in *IEEE Design & Test of Computers*, pp. 6–15, 1993.

[6] F. Vahid and T. Le, "Towards a model for hardware and software functional partitioning," in *International Workshop on Hardware-Software Co-Design*, pp. 116–123, 1996.

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

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

[9] R. Tessier, J. Babb, M. Dahl, S. Hanono, and A. Agarwal, "The virtual wires emulation system: A gate-efficient asic prototyping environment," in *International Workshop on Field-Programmable Gate Arrays*, 1994.

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

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

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

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

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

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

[16] R. Camposano and R. Brayton, "Partitioning before logic synthesis," in *Proceedings of the International Conference on Computer-Aided Design*, 1987.

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

[18] P. Gupta, C. Chen, J. DeSouza-Batista, and A. Parker, "Experience with image compression chip design using unified system construction tools," in *Proceedings of the Design Automation Conference*, pp. 250–256, 1994.

[19] F. Vahid, "Procedure exlining: A transformation for improved system and behavioral synthesis," in *Proceedings of the International Symposium on System Synthesis*, pp. 84–89, 1995.

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

[21] Y. Hsu, T. Liu, F. Tsai, S. Lin, and C. Yu, "Digital design from concept to prototype in hours," in *Asia-Pacific Conference on Circuits and Systems*, Dec. 1994.

[22] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "MIS: A multiple-level logic optimization system," *IEEE Transactions on Computer-Aided Design*, vol. 6, pp. 1062–1080, November 1987.