Experiments on functional partitioning for packaging constraints and synthesis tool performance

Frank Vahid Thuy dm Le Yu-chin Hsu

Technical Report CS-95-03
October 4, 1995

Department of Computer Science
University of California
Riverside, CA 92521

We describe several significant advantages of incorporating functional partitioning into a synthesis methodology. In particular, we examine the problem of partitioning to meet packaging size and I/O constraints. We show that we can obtain better performance while using far fewer packages by functionally partitioning a behavioral specification and then synthesizing structure for each part, as opposed to the current approach of synthesizing structure first and then partitioning the structure. In addition, we highlight results that show how functionally partitioning a behavioral specification before synthesis can dramatically reduce synthesis runtime. The substantial improvements described imply that the development of automated functional partitioning tools can vastly improve the quality and practicality of a synthesis environment.

Contents

List of Tables

List of Figures

.

1 Introduction

The existence of behavioral specifications for newly-designed digital systems is becoming commonplace. A behavioral specification is a program-like description of the system's functionality. Advantage of creating such a specification include the ability to simulate the system early to eliminate functional errors, to use synthesis to reduce design time, and to create precise, unambiguous documentation.

Many recent research efforts focus on functionally partitioning such specifications among software (standard processor) and hardware (custom processor) components, to explore cost/performance tradeoffs between hardware and software [1, 2, 3, 4, 5, 6, 7]. However, functional partitioning has two other very important uses. First, functional partitioning can be used to satisfy the packaging constraints of implementation components. For example, a design being implemented with FPGA's must satisfy the I/O and size constraints of each FPGA. Automated approaches to such packaging-constraint satisfaction focus on partitioning structure (e.g., [8]). However, with the existence of a behavioral specification, it might now be better to focus on partitioning functionality rather than structure. We show that such a focus can lead to far better results. In addition, such a focus is consistent with how system designers often partition manually so lends itself to interactive techniques. Second, functional partitioning can greatly improve the performance of synthesis tools. In particular, a single process in a behavioral specification is often large, causing synthesis tools to use long CPU times and sometimes to produce inferior results; we show that functionally partitioning can solve this problem. In this paper, we describe each of these uses, and provide results of experiments showing the substantial benefits of applying functional partitioning.

2 Functional vs. structural partitioning for package constraints

Figure 1 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 that contain an interconnection of register-transfer (RT) and gate level objects. One then partitions the structure among system components. In our terminology, a *system component* may be an FPGA, an ASIC chip, a block on an ASIC, or even a part of a multi-chip module. In contrast with structural partitioning, in functional partitioning we first partition the functions of the behavioral specification into groups, where each group represents a system component. Any communication between components is achieved using a high-level transfer protocol such as a handshake. We then implement each component's functions as structure, optionally repeating the functionally partitioning within each component. (In hardware/software partitioning, the implementation of the component's function is achieved by synthesizing software rather than hardware).

Functional partitioning requires estimations of performance, I/O, and hardware size for

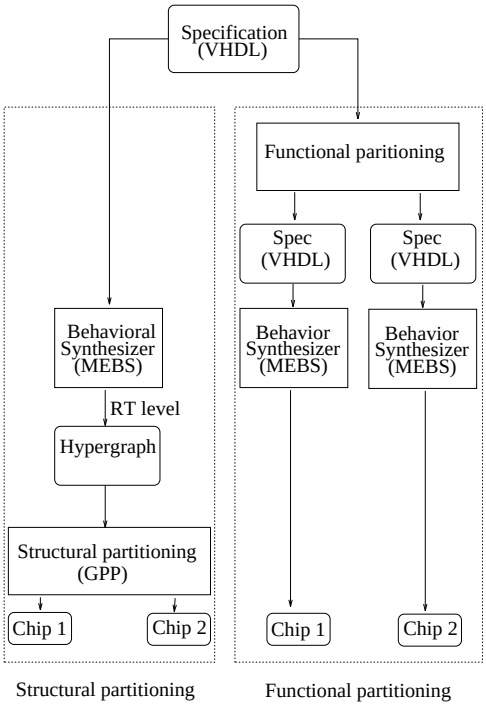

Figure 1: Functional vs. structural partitioning approaches.

any given set of functions, making such partitioning more challenging to automate since 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 inter-component 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 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.

2.1 Method

We now describe the method used in our experiments in detail.

2.1.1 Examples

Experiments were performed on four VHDL examples: (1) *2p-fact* – this example, given a number n, will find the prime numbers p and q such that $n = pq$; (2) *chin thm* – this example is the application of "Chinese Remainder Theorem", which finds the value of x such that it satisfies three congruent equations, (3)*8-bits rsa* – a simple version of the RSA encryptography system, and (4)*vol*, a volume-measuring medical instrument controller. All examples were written at the algorithmic level, as opposed to a state-machine or RT level. The average lines of four VHDL examples is 200 lines.

2.1.2 Functional partitioning

We first decomposed the specification into a set of functional objects to be assigned to system components. The granularity of these objects were procedures and variables. Arguments in favor of this granularity, as opposed to say statements, for functional partitioning can be found in [2, 6, 9, 7]. Techniques in [10] can be used to group statements into procedures when the user-written procedures would have proven too coarse-grained.

We then partitioned the functional objects among two groups. All examples were partitioned manually, using mental estimates of size, performance and I/O to guide decisions. We also applied a prototype automated partitioner and estimator (SpecSyn [11]), which in turn created an internal model and passed it to a partitioning engine (GPP – General Purpose Partitioner) for partitioning by simulated annealing, and we then compared the resulting partition with the manual partition; they were usually very close, and on two occasions the slight differences suggested improvements to the manual partition.

After choosing a partition, we then manually rewrote the specification as two processes, each process containing a subset of the original set of procedures and variables. These processes communicated via global signals, where some signals were used for data, and others for control handshaking. The manual 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 when the tool matures, iteration time may be greatly reduced).

2.1.3 Structural partitioning

We first synthesized the entire VHDL specification to a controller block and interconnected RT-level objects. We chose to partition structure at the RT-level, rather than the gate level as assumed in many structural partitioning techniques, 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 is 115 nodes. The simulated annealing cooling schedule was chosen so that GPP would run about 20 minutes to partition the hypergraph. The cost function tried to minimize size and pin 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.

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 list of size library

2.1.4 Behavioral synthesis

We used the MEBS behavioral synthesis tool [12] 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 [13] tools to implement the controller, and can then map the structure into a Xilinx technology library for FGPA implementation.

2.1.5 Constraints

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.

2.2 Results

Results are shown in Table 3. The *unpartition* column shows the I/O and size when synthesizing the example 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

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

after functional partitioning without any additional buses created after partitioning. The *bus* column shows data when sequential communications between the two chips are assigned to a single bus, resulting in smaller I/O. The *p.d.* (ports distributed) column shows data 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.

Sys.	*Unpartition*	*Functional Partition.*			*Structural Partition*			
2p-fact		no bus	bus	p.d.	Resource Sharing			
					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	–	102/8188	38/7594	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	–	134/8705	38/7765	102/8249	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)			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

2.3 Analysis

2.3.1 I/O and size

Functional partitioning led to better satisfaction of I/O and size constraints, and thus to fewer FPGA's in the implementation and hence lower cost. 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. *Note that such merging after partitioning is not possible in 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 partitioning, we found only minor improvements in I/O satisfaction. For example, in *2p-fact* example, the packaging violation is 0 pin and 14043 gates (or 0/14043).

When faced with such constraint violations during structural partitioning, the only alternative is to add more FPGA's, leading to much higher-cost designs. Thus we see that functional partitioning can lead to much lower-cost designs by using far fewer FPGA's. In addition, port distribution, which is possible when functionally partitioning, can reduce the I/O even further, thus enabling use of even lower-cost FPGA's.

2.3.2 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., 7 ns for the Xilinx FPGA) that occurs during any register-to-register transfer. For the examples, we optimistically assumed only a one delay (7 ns) 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 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 compute performance as:

$$t_{sp} = 500 \times \tau_{sp}$$
$$t_{fp} = (500 + h) \times \tau_{fp}$$

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

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. Apparently, 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 transferring data between components using handshaking. Speedups ranged from 1.02 to 1.63. Note that, if we assume more intercomponent delays, or consider examples requiring more than 500 cycles, speedups would be even greater.

3 Partitioning for synthesis tool performance

3.1 Method

We evaluate the 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. As shown in Figure 2, 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.

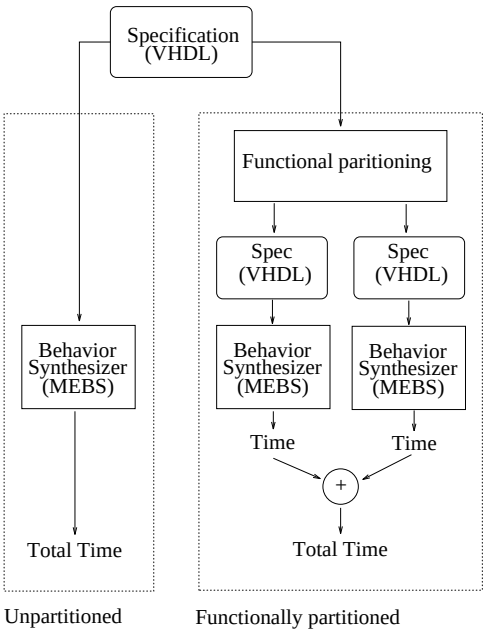

Figure 2: Method for synthesis tool evaluation

3.1.1 Examples

We used the same examples as discussed in the previous section. However, we wanted to measure the effect of input size on tool performance. But, 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 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 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).

3.1.2 Functional partitioning

Functional partitioning was performed in the same manner as described in the previous section.

3.1.3 Behavioral 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 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.

3.2 Results

Table 5 provides a comparison of the results of synthesis of the unpartitioned and partitioned examples, in which the optimal logic mode was used during synthesis. The first column represents the number of duplications for a given example, as described earlier. The second column represents the CPU time for synthesizing the unpartitioned example. The third and fourth columns are the CPU times for synthesizing each part of the partitioned specification, and the fifth column is the sum of these two times. The sixth column shows the speedup obtained by partitioning. The last column shows the speedup if we assume that the two parts of the partitioned specification can be synthesized in parallel. (Due to duplication size of the examples and the synthesis tool limitation, Table 5 and Table 6 report only the results of three examples each.)

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

3.3 Analysis

3.3.1 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 the existence of polynomial time heuristics

2P-FACT						
Dup.	*Unpart.*	*Functional part.*			*Spd. Up*	
		Chip 1	Chip 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*	
		Chip 1	Chip 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*	
		Chip 1	Chip 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

Table 5: Synthesis times (optimal mode)

2P-FACT						
Dup.	*Unpart.*	*Functional part.*			*Spd. Up*	
		Chip 1	Chip 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*	
		Chip 1	Chip 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*	
		Chip 1	Chip 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

Table 6: Synthesis times (fast mode)

2P-FACT					
Dup.	*Unpart.*	*Functional part.*			*Ratio*
		Chip 1	Chip 2	Total	
1	13312	6347	7431	13778	0.9
2	24271	8410	10279	18689	1.3
3	34160	11431	13628	25059	1.3
4	45033	14717	15219	29936	1.5
8-bits RSA					
Dup.	*Unpart.*	*Functional part.*			*Ratio*
		Chip 1	Chip 2	Total	
1	17275	14200	3558	17758	0.9
2	25655	21640	6525	28165	0.9
3	33435	28994	10814	39808	0.8
4	43182	34635	11690	46325	0.9
VOL					
Dup.	*Unpart.*	*Functional part.*			*Ratio*
		Chip 1	Chip 2	Total	
1	11996	11884	6856	18740	0.6
2	19489	19449	9861	29310	0.7
3	27911	23043	13989	37032	0.8
4	35661	29306	17806	47112	0.8

Table 7: Size outputs

in the synthesis tools. Dividing the specification thus has a non-linear effect on the tool's CPU time.

3.3.2 Design size

One concern is that 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%. 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.

3.3.3 Predictor

The number of states/transitions and functional units seemed to predict the synthesis time. When using optimal logic synthesis, synthesis time was 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.

4 Conclusion

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

References

[1] 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.

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

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

[5] 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.

[6] 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.

[7] F. Vahid and D. Gajski, "Incremental hardware estimation during hardware/software functional partitioning," in *IEEE Transactions on Very Large Scale Integration Systems*, pp. 459–464, 1995.

[8] Y. Kirkpatrick and C. Cheng, "Ratio cut partitioning for hierarchical designs," *IEEE Transactions on Computer-Aided Design*, vol. 10, no. 7, pp. 911–921, 1991.

[9] 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.

[10] F. Vahid, "Procedure exlining: A transformation for improved system and behavioral synthesis," in *International Symposium on System Synthesis*, 1995.

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

[12] 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.

[13] 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.