A Three-Step Approach to the Functional Partitioning of Large Behavioral Processes

Frank Vahid
Department of Computer Science and Engineering
University of California, Riverside, CA 92521

Abstract

Earlier work has demonstrated that partitioning one large behavioral process into smaller ones before synthesis can yield numerous advantages, such as reduced synthesis runtime, easier package constraint satisfaction, reduced power consumption, improved performance, and hardware/software tradeoffs. In this paper, we describe a novel three-step functional partitioning methodology for automatically dividing a large behavioral process into mutually-exclusive subprocesses, and we define the problems and our solutions for each step. The three steps are granularity selection, pre-clustering, and N-way assignment. We refer to experiments throughout that demonstrate the effectiveness of the solutions.

1 Introduction

Functional partitioning divides a system's functional specification into multiple sub-specifications. Each sub-specification represents the functionality of a system component, such as a custom-hardware or software processor, and is eventually synthesized down to gates or compiled down to machine code. While much earlier research focuses on the partitioning of a large number of processes among processors, we have found that many applications consist of only one or a small number of very large processes.

In such cases, we have found numerous benefits to automatically partitioning, before synthesis or compilation, one large process into smaller sub-processes that execute in mutual exclusion [1]. One benefit is an order-of-magnitude reduction in logic synthesis runtimes (tens of hours down to tens of minutes). Such reductions occur because synthesis tool heuristics are usually non-linear, so the sum of runtimes for synthesizing several small processes (where we assume the common situation of a synthesis tool synthesizing each process to its own custom processor) can be much less than the runtime for one large process. A related benefit is improved system performance sometimes achieved when the smaller processes can be synthesized into custom processors with shorter clock periods than one large processor, and when these shorter periods outweigh the overhead of inter-processor communication In fact, synthesis tool manuals often recommend that a designer functionally partition a system manually before synthesis for these very reasons. (Of course, further performance benefits can be gained by allowing the multiple processes to execute concurrently, a natural extension of the work in this paper). Another benefit is the improved satisfaction of input/output (I/O) and size capacity constraints on a package such as an FPGA, resulting from nearly order-of-magnitude reductions in inter-package signals compared to structural partitioning, which is known to be I/O limited [2]. Such improved satisfaction leads to fewer packages and thus smaller board sizes and lower cost systems, while also having implications for easing the mapping problem of logic emulators. Our recent experiments have demonstrated 50% reductions in power consumption achievable through functional partitioning, since only one of the small mutually-exclusive processors is active at a time, so the inactive processors consume almost no power. Some hardware/software codesign researchers have shown the benefits of partitioning a single process to reduce hardware cost or increase software speed (e.g., [3, 4]). Other potential benefits include concurrent synthesis or design, easier debug, and possibly fewer physical design problems when dealing with several smaller modules rather than one large one.

Our earlier research has focused on functional partitioning heuristics, rapid estimation during partitioning, transformations for improving partitioning results, and experiments demonstrating the benefits of such partitioning. In this paper, we describe a novel three-step functional partitioning methodology necessary to put together the earlier pieces into a coherent automated approach. We highlight experiments throughout. We also summarize related work.

2 Problem description

2.1 Model

The input consists of a single behavioral process X, such as a C program or a VHDL process. The process describes a complex repeating sequential computation, often consisting of numerous models of operation, and typically requiring many hundreds or thousands of lines of sequential program code. This process can be viewed as consisting of a set of procedures $F = \{f_1, f_2, ..., f_n\}$, with one representing a main procedure (in VHDL, the process body is the main procedure). A variable is treated as a simple procedure, with reads and writes being procedure calls. Execution of F consists of procedures executing sequentially, starting with the main procedure, which in turn calls other procedures; at any given time, one and only one procedure is active – in other words, the procedures are mutually exclusive.

0-8186-8623-5/98 $10.00

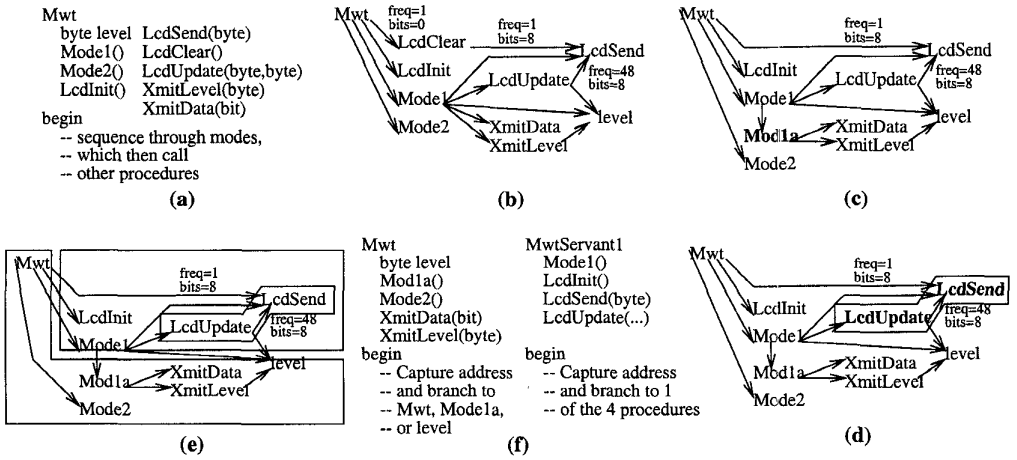

Fig. 1: The three partitioning steps on a simple example: (a) input specification with numerous procedures, (b) access graph, (c) granularity selection (using exlining), (d) pre-clustering, (e) N-way assignment (among 2 parts).

Functional partitioning creates a partition P consisting of a set of parts $\{p_1, p_2, ...p_m\}$, such that every procedure f_i is assigned to exactly one part p_j, i.e., $p_1 \cup p_2 \cup ...p_m = F$ and $p_i \cap p_j = \emptyset$ for all $i, j, i \neq j$. Each p_j represents the functions to be implemented on a single processor. Execution of F is the same as above. Since only one procedure is active at a time during execution, then only one processor will be active at a time, so the processors are mutually exclusive.

Each part p_j is converted to a single process before synthesis; this process consists of a loop that detects a request for one of the part's procedures, receives the necessary input parameters, calls the procedure, and sends back any output parameters. All parameter passing occurs over a single bus between the processors, called a FunctionBus, consisting of shared address/data lines, an address request control line, and a data request control line. The protocol is one of putting the destination procedure's address on the bus and pulsing the address request, then putting each parameter on the bus and pulsing the data request, possibly sending a parameter in several chunks if its size exceeds the bus size.

Synthesis converts a process into a custom *processor* component c_i; for the applications we target, c_i consists of a non-trivial datapath and a complex controller with hundreds of states. A procedure on c_i may be implemented using any of various techniques, such as a control subroutine, a datapath component, or even inlined. Synthesis may implement some of a process' procedures in parallel, as long as data dependencies are not violated – therefore, while the procedures are not necessarily mutually exclusive after partitioning, the processors still are mutually exclusive. Note that many processors will likely co-exist on a single package (e.g., an ASIC).

2.2 Tasks

Five tasks are necessary to achieve a good functional partitioning. *Model creation* converts the input to an internal model; we use a call graph model. For example, Figure 1(a) shows an extremely simplified input (the example comes from a 720 VHDL example of a microwave-transmitter controller) with one process *Mwt*, one global variable, and numerous procedures. Figure 1(b) shows the call graph, where each node represents a procedure, and each edge a procedure call. A call graph easily handles multiply-called and deeply nested procedures, a common situation often overlooked. *Allocation* is the task of instantiating processors of varying types; we assume this is done beforehand. *Partitioning* is the task of dividing the input process among the allocated processors – the focus of this paper. *Transformation* modifies the input process into one with a different organization but same overall functionality, leading to a better partition, and is also mentioned in this paper. *Estimation* provides data used to create values for design metrics. We use two types of estimation. Pre-estimation computes estimation information for a given graph node (procedure) or edge, and is usually done once, before partitioning heuristics are applied. Examples of pre-estimated data include the number of bytes, gates, or clock cycles for a functional object to execute on each possible component, e.g., Figure 1(b) shows values for calling frequency and bits per transfer for three edges. Online-estimation determines actual metric values for a given implementation by combining pre-estimated values, and is usually done hundreds or thousands of times during application of a partitioning heuristic. It may involve sophisticated data structures that can be incrementally updated as objects are moved. We have described estimation techniques elsewhere [5, 6].

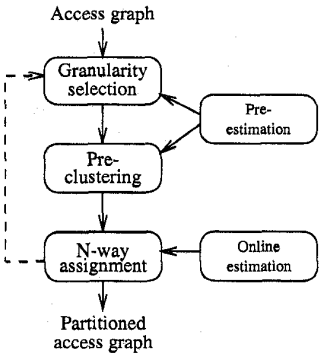

Fig. 2: Sequence of partitioning steps.

3 Partitioning methodology

We apply a three-step partitioning methodology to obtain an implementation. They occur in sequence as shown in Figure 2, though iterating over the sequence is certainly desirable. We now describe each step.

3.1 Step 1: granularity selection

The goal of this partitioning step is to extract procedures from the specification, which are to be assigned to processors during subsequent N-way assignment. The essential feature of this step is that it determines the granularity of subsequent N-way assignment. Granularity is a measure of the complexity of each procedure. Fine-granularity means many procedures of low complexity, while coarse granularity means few procedures of high complexity.

Improper granularity selection can have negative effects on subsequent partitioning results. First, fine-granularity means little useful pre-estimation can be done, which in turn means online estimation may be less accurate. Alternatively, online estimation may be made more complex to achieve accuracy, requiring more time and thus prohibiting use of powerful N-way assignment heuristics that need many estimates.

Second, fine-granularity means there will be numerous procedures, preventing the use of powerful partitioning heuristics that have high, such as quadratic, runtime complexities.

Third, coarse-granularity means that many behaviors are grouped together into an inseparable unit, so any possible solution that separates those behaviors is excluded. If coarse-grained procedures are not created carefully, they can prevent good solutions.

From the above effects, we can see the importance of a distinct partitioning step that chooses the procedures very carefully. In our approach to this step, we treat each statement as an atomic unit. Granularity selection is thus the problem of partitioning statements into procedures, such that: (1) procedures are as coarse-grained as possible, to enable maximum pre-estimation and application of powerful N-way heuristics, and (2) statements are grouped into a procedure only if their separation would yield inferior solutions.

The most straightforward heuristic for this partitioning step is to choose a particular specification construct to represent a procedure, such as each statement or block. In our approach, we start by selecting each user-defined procedure (along with each global and non-scalar variable) as a procedure for partitioning. User-defined procedures tend to be excellent coarse-grained groupings of statements, but there is often a need for some improvement, which can be obtained through transformations.

Procedure inlining is the well-known transformation of replacing a procedure call by the procedure's contents, thus making granularity coarser (the inlined procedure disappears, while all calling procedures are made more complex).

Procedure cloning makes a copy of a procedure for exclusive use by a particular caller. This transformation is a compromise between inlining and not inlining. A multiply-called procedure when inlined might lead to excessive size growth, but if not inlined might become a communication bottleneck. Cloning eliminates the bottleneck while avoiding excessive size growth. Prior experiments demonstrate performance and I/O improvements obtainable through cloning [7].

Procedure exlining, as its name implies, is the inverse of inlining. It replaces a subsequence of a procedure's statements by a call to a new procedure containing only that subsequence. This transformation makes the granularity finer, as it essentially replaces one procedure by two simpler ones. We have isolated two forms of exlining. *Redundancy exlining* seeks to replace two or more near-identical sequences of statements by one procedure. We use an interactive approach based on approximate-string matching, in which each statement is encoded into a set of characters and appended to a string represent the entire program, and then near-redundant strings are searched for. *Distinct-computation exlining* seeks to divide a large sequence of statements into several smaller procedures such that statements within a procedure are tightly related and would thus never be separated in a good N-way assignment solution. Once again, we make statements our atomic unit; an approach at the arithmetic-operation level is described in [8]. We transform the statements into a tree, and then perform one of three heuristics (varying in their complexity) that insert procedure nodes into the tree with the goal of minimizing a cost function including procedure size and communication. Further exlining details can be found in [9].

Figure 1(c) illustrates sample call-graph modifications made during granularity selection on the earlier example. The procedure *Model* is large and accesses pro-

cedures that access both an external LCD and a transmitter. Exlining introduces a new procedure *Mod1a*, which makes the granularity finer and expanding the partitioning solution space to hopefully include better possible solutions. In particular, it separates the original procedure's statements into those that access the LCD and those that access the transmitter. On the other hand, *LcdClear* is very small, consisting only of one call to *LcdSend* with a particular parameter value. Thus, *LcdClear* is inlined, making the granularity coarser and thus eliminating hopefully inferior partitioning solutions. Experiments illustrating the partitioning improvements obtained by exlining can be found in [9].

3.2 Step 2: pre-clustering

Given a set of procedures, the goal of pre-clustering is to reduce the number of procedures for subsequent N-way assignment, by merging procedures whose separation among parts would never represent a good solution.

This step is distinguished from granularity selection by the fact that the procedures being clustered here may not be such that they could have been exlined into a single new procedure; i.e., the calls to these procedures do not appear as adjacent statements. Typically, calls to such procedures are scattered throughout the specification. This step is distinguished from N-way assignment by the fact that each cluster does not represent a processor (many clusters may be assigned to each processor), so this step cannot be guided by direct design metric estimates as can N-way assignment.

We perform pre-clustering by using a hierarchical clustering. The procedures created after granularity selection are each converted to a graph node, and edges are created between every pair weighed by the closeness of the nodes. The closest pair of nodes is merged into a new (hierarchical) node, and the merging repeats until no pair of nodes exceeds a minimum closeness threshold. Closeness is defined using a weighted sum of several normalized closeness metrics. We must take care to define these metrics as a normalized number (between 0 and 1), because leaving such normalization to the user through the selection of weights, as many approaches require, is an extremely difficult if not impossible user task. We describe our closeness metrics in [10]. Some of those metrics are similar to those for logic-operation clustering [11] and arithmetic-operation clustering [12, 8].

Figure 1(d) illustrates results of pre-clustering on the earlier example. Two procedures *LcdUpdate* and *LcdSend* communicate very heavily: 48 times per call to *LcdUpdate*, transmitting 8 bits each time. These two procedures should probably never be separated. Since *LcdSend* actually appears 48 times inside *LcdUpdate*, inlining during granularity selection was not a reasonable option. We merge these two procedures during pre-clustering; they are subsequently treated as one object.

We demonstrated in [10] that pre-clustering can reduce the runtime of N-way assignment using simulated annealing or Kernighan/Lin by between 25% and 30%. For N-way greedy heuristics, pre-clustering can actually improve the final results significantly. To illustrate the benefits of pre-clustering, we experimented with two-way partitioning of an Ethernet coprocessor example, consisting of 125 procedures to be partitioned. The tradeoffs between the amount of pre-clustering versus the subsequent runtime of the Kernighan/Lin heuristic and the cost of the final results are shown in Figure 3. Cost is unit-less number representing a weighted sum of normalized size, I/O and performance metrics; smaller means better. We see that when pre-clustering merges until there are only 85 objects, runtime is decreased by 30%, and a better-cost partition is found by the Kernighan/Lin heuristic. Merging to less than 55 objects yielded much higher costs, due to distant objects being forcibly merged.

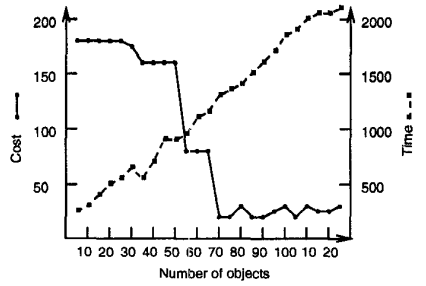

Fig. 3: Pre-assignment clustering effects on iterative-improvement runtime and resulting partition cost.

3.3 Step 3: N-way assignment

The goal of N-way assignment is to distribute the procedures among the given set of processors. Granularity selection and pre-clustering should have already created a good set of procedures.

Constructive heuristics are used to create an initial partition, and include random distribution, as well as clustering using the same closeness metrics defined above. One additional metric is used: *Balanced size* is the size of an implementation of both sets of nodes, divided by the size of all nodes. This metric favors merging small sets over large ones, thus preventing one set from getting too large and defeating the purpose of functional partitioning.

Iterative improvement heuristics improve a given partition by moving procedures from one part to another in order to reduce the value of a cost function. We use three heuristics. Greedy improvement is a linear-time heuristic that moves nodes that reduce cost until no further such node is found. Simulated annealing uses randomized hill climbing to avoid local minima

at the expense of potentially long runtimes. An extended version of Kernighan/Lin uses a more restricted method of hill climbing to avoid some local minima, along with a tightly-coupled data structure, yielding only $O(n \log(n))$ runtime complexity. We demonstrated in [5] that this last heuristic obtains results nearly as good as simulated annealing, but in just a few seconds rather than many minutes.

We apply two transformations during N-way assignment to improve partitioning results. *Cloning*, a transformation defined earlier, can be performed selectively along with this partitioning step, rather than trying to clone only during granularity selection [7]. A second transformation, *port-calling*, can be used to better balance I/O among the parts and to reduce total I/O by eliminating the case where two parts have I/O accessing one external port. This transformation involves replacing all accesses to each external port by a port-call procedure, and then moving this procedure among parts just like any other procedure. The port-call procedure acts as a broker, accessing the port, and sending/receiving data over the FunctionBus to/from the original accessing procedure.

Figure 1(e) shows the earlier example's call graph after being partitioned into two parts. While not entirely obvious from the figure because not all annotations are shown, the partition minimizes communication, isolates external I/O access to one or the other part, and creates roughly balanced parts, each of which is roughly half the size of the original input.

We have performed numerous experiments illustrating the relative tradeoffs among various N-way heuristics [5, 10]. We have also performed experiments showing I/O and performance improvements obtained through cloning [7] as well as through port-calling [13].

4 Three-step experiments

The above-referenced experiments each illustrates the impact of a particular technique applied at a particular step. In this section, we demonstrate the impact of combining some of those techniques in the three-step methodology; a more extensive combination of all the techniques remains for future work.

We examined three examples, *ans* (an answering machine), *fuzzy* (a fuzzy-logic controller), and *mwt* (a microwave transmitter controller). Our goal was to partition each among a hardware/software architecture consisting of microcontroller and an FPGA, such that the FPGA size constraint was met and the example's execution time was minimized. For each example, we applied step 1 (granularity selection) to three different degrees, by applying exlining agressively *exl-2*, moderately *exl-1*, or not at all *exl-0*. Then, for each granularity, we applied step 2 (pre-clustering) to three different degrees, by applying clustering with a termination criteria of a closeness threshold of *0.6*, *0.7* or *0.8*. On the resulting

Version	ans	fuzzy	mwt
exl-0 0.6	739	60	1077
exl-0 0.7	290	60	1870
exl-0 0.8	274	60	96
exl-1 0.6	626	241	600
exl-1 0.7	32	48	6716
exl-1 0.8	738	49	600
exl-2 0.6	441	6769	1270
exl-2 0.7	169	78	676
exl-2 0.8	74	78	3800

Table 1: Three-step methodology experiments.

9 versions of each example, we applied step 3 (N-way assignment) using simulated annealing (start temperature 50, stop temperature 0, equilibrium condition 200 moves with no improvement, temperature decrease ratio 0.93). Results are summarized in Table 1. The numbers corresponds to normalized squared execution times beyond a target execution time for each example – in short, lower numbers mean better performance. We see that the best configuration of parameters (i.e., the degree of exlining, and the degree of clustering) during each step is highly example-dependent. Thus, a functional partitioning tool should allow tuning of those parameters for each example. Further work is needed to evaluate good configurations and relationships among the parameters.

5 Related work and limitations

Several researchers have developed functional partitioning techniques for partitioning among hardware packages. Most efforts partitioned arithmetic-level operations. Aparty [8] partitioned operations among datapath modules using multi-stage clustering. Vulcan [14] partitioned operations among packages using iterative improvement heuristics. Chop [15] partitioned operations among packages, focusing on providing a suite of feasible solutions for each package that would satisfy overall constraints. Multipar [16] and Gebotys' techniques [17] partitioned operations among packages simultaneous with scheduling and allocation, using integer-linear programming. Other efforts partitioned procedural-level operations. SpecPart [6] partitioned procedures among packages using clustering and iterative improvement. Peng and Kuchcinski [18] similarly partitioned among packages. None of these efforts applied a three-step methodology as described in this work, although the idea behind multi-stage clustering in Aparty is along similar lines, while being limited to just clustering for the partitioning heuristic and focusing on finer-grained operations.

Recently, much research effort has focused on developing functional partitioning techniques for hardware-software partitioning (for example, see [19]). Most of these techniques partition and possibly schedule multiple processes among a custom-hardware and software processor architecture [20, 3, 4, 6, 21], while some focus

on simultaneous processor allocation and partitioning [22]. Since these techniques focus on multiple processes, they do not apply directly to the single-process problem we address. However, Henkel and Ernst have investigated granularity during hardware-software partitioning and developed a technique to dynamically modify granularity during partitioning [23].

Our work can be viewed as filling a gap between these two research domains. Most functional partitioning work initially dealt with extremely fine-granularity when they were first developed as extensions of behavioral synthesis, but today most work focuses on very coarse granularity and deals primarily with hardware-software partitioning. We focus on medium to coarse granularity (procedures), with an emphasis on solving important synthesis problems.

One limitation of the approach for partitioning before synthesis is that total hardware increase may be large for examples that have very simple controllers but large datapaths. Another limitation is that we don't address the problem of partitioning large numbers of small processes; that problem is very much a scheduling problem. Third, our model dictates that procedures on different processors do not execute in parallel. Allowing parallel execution could improve performance, but at the expense of introducing the need for bus arbitration and other complexities.

6 Conclusions

We described a three-step partitioning methodology for partitioning a large behavioral process among custom hardware or software processors. While earlier work has described detailed solutions to various functional partitioning problems, this work combines those solutions into a single coherent approach.

There are many parameters, described in this paper, that can be used to influence the results of each partitioning step. Finding the right parameter values can be hard, and modifying them when iterating the three steps can be challenging. Thus, an automatable "meta-algorithm" for setting and modifying those parameters during iteration among the steps may be an interesting research direction.

Acknowledgements: We wish to acknowledge Tony Givargis for formulating the experiments, and Roman Lysecky, Puneet Mehra, and Jason Villarreal for their assistance.

References

[1] F. Vahid, T. Le, and Y. Hsu, "A comparison of functional and structural partitioning," in *ISSS*, pp. 121–126, 1996.

[2] F. Johannes, "Partitioning of VLSI circuits and systems," in *DAC*, 1996.

[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] F. Vahid and T. Le, "Extending the kernighan/lin heuristic for hardware and software functional partitioning," *Journal of Design Automation of Embedded Systems, Kluwer*, vol. 2, no. 2, pp. 237–261, 1997.

[6] D. Gajski, F. Vahid, S. Narayan, and J. Gong, "Specsyn: An environment supporting the specify-explore-refine paradigm for hardware/software system design," *IEEE Transactions on VLSI*, p. to appear, 1998.

[7] F. Vahid, "Procedure cloning: A transformation for improved system-level functional partitioning," in *Proceedings of the European Design and Test Conference (EDTC)*, pp. 487–492, 1997.

[8] 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.

[9] F. Vahid, "Procedure exlining: A transformation for improved system and behavioral synthesis," in *ISSS*, pp. 84–89, 1995.

[10] F. Vahid and D. Gajski, "Clustering for improved system-level functional partitioning," in *ISSS*, pp. 28–33, 1995.

[11] R. Camposano and R. Brayton, "Partitioning before logic synthesis," in *ICCAD*, 1987.

[12] M. McFarland and T. Kowalski, "Incorporating bottom-up design into hardware synthesis," *IEEE Transactions on Computer-Aided Design*, pp. 938–950, September 1990.

[13] F. Vahid, "Port calling: A transformation for reducing i/o during multi-package functional partitioning," in *ISSS*, 1997.

[14] R. Gupta and G. DeMicheli, "Partitioning of functional models of synchronous digital systems," in *ICCAD*, pp. 216–219, 1990.

[15] K. Kucukcakar and A. Parker, "CHOP: A constraint-driven system-level partitioner," in *DAC*, pp. 514–519, 1991.

[16] Y. Chen, Y. Hsu, and C. King, "MULTIPAR: Behavioral partition for synthesizing multiprocessor architectures," *IEEE Transactions on VLSI*, vol. 2, pp. 21–32, March 1994.

[17] C. Gebotys, "Optimal synthesis of multichip architectures," in *ICCAD*, pp. 238–241, 1992.

[18] Z. Peng and K. Kuchcinski, "An algorithm for partitioning of application specific systems," in *Proceedings of the European Conference on Design Automation (EDAC)*, pp. 316–321, 1993.

[19] R. Gupta, "Special issue: Partitioning methods for embedded systems," *Journal of Design Automation of Embedded Systems, Kluwer*, vol. 2, no. 2, 1997.

[20] A. Kalavade and E. Lee, "A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem," in *CODES*, pp. 42–48, 1994.

[21] A. Balboni, W. Fornaciari, and D. Sciuto, "Partitioning and exploration strategies in the tosca co-design flow," in *CODES*, pp. 62–69, 1996.

[22] J. Hou and W. Wolf, "Process partitioning for distributed systems," in *CODES*, pp. 70–75, 1996.

[23] J. Henkel and R. Ernst, "A hardware/software partitioner using a dynamically determined granularity," in *DAC*, 1997.