A Survey of Behavioral-Level Partitioning Systems

Frank Vahid

Technical Report #91-71
October 30, 1991

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-8059

vahid@ics.uci.edu

Abstract

Many approaches have been developed to partition a system's behavioral description before a structural implementation is synthesized. We highlight the foundations and motivations for behavioral partitioning. We survey behavioral partitioning approaches, discussing abstraction levels, goals, major steps, and key assumptions in each.

Contents

1 Introduction **1**

2 Behavioral Partitioning: Foundations and Motivations **2**
2.1 Basic Partitioning . . . 2
2.1.1 Definitions and Terminology . . . 2
2.1.2 Partitioning Algorithms . . . 3
2.2 Basic High-level Synthesis . . . 6
2.3 Motivations for Behavioral Partitioning . . . 7
2.3.1 Partitioning for Tractability . . . 7
2.3.2 Partitioning for Packaging-Constraint Satisfaction . . . 8

3 A Survey of Behavioral Partitioning Systems **10**
3.1 YSC – The Yorktown Silicon Compiler . . . 11
3.2 BUD – Bottom-Up Design . . . 13
3.2.1 Synthesis by Delayed Binding of Decisions . . . 16
3.3 APARTY – Architectural Partitioning . . . 16
3.4 Workbench Behavioral Transformations . . . 20
3.5 Vulcan – Partitioning of Functional Models . . . 20
3.6 CHOP . . . 22
3.7 SpecPart – Specification Partitioning . . . 23
3.8 SPARTA and SLIP . . . 28

4 Summary of Three Important Aspects **29**

5 Conclusions **29**

6 References **30**

A Appendix **31**
A.1 Partitioning for Tractability: Allocation Example . . . 31

List of Figures

1 Behavioral-partitioning abstraction levels vs. goals . . . 1
2 Graphs and partitions . . . 2
3 Building a cluster tree during pairwise clustering . . . 4
4 Group migration's local solution-space search strategy . . . 5
5 These seemingly equal moves can be distinguished using a group migration "look-ahead" extension 6
6 A sample mapping of a behavior to CDFGs . . . 7
7 A simple behavior for which functional units must be allocated . . . 8
8 Behavioral vs. structural partitioning . . . 9
9 Comparison of behavioral-partitioning approaches . . . 10
10 Partitioning before logic synthesis . . . 12
11 Building a cluster tree in BUD for behavioral operations . . . 15
12 Selecting a partitioning and creating partitioned structure in BUD . . . 15
13 Multistage clustering . . . 17
14 The extended hypergraph model in Vulcan . . . 21
15 Specification partitioning . . . 25
16 Decomposing a behavior for finer-granularity partitioning . . . 26
17 A refined specification resulting from partitioning . . . 26
18 Incorporating performance constraints in SpecPart . . . 27

1 Introduction

High-level synthesis (HLS) converts a behavioral specification to a structure (usually a controller/datapath pair, or CU/DP). Partitioning the structure can improve floorplanning and other intrachip tasks, or can create structure subgroups that meet chip size and pin constraints. Recent work has focused on partitioning behavior before obtaining structure, not only because there are fewer objects to deal with, but also because results of such partitioning can be used to influence later HLS tasks.

There are essentially two levels at which behavioral partitioning can be performed. At the *operation* level, dataflow-level operations such as addition and subtraction, belonging to a single sequential behavior, are grouped. At the *algorithmic* level, entire program-grained computations such as processes and procedures, making up a set of sequential and concurrent behaviors, are grouped. We can also distinguish between *intrachip* and *interchip* partitioning goals. Goals may also involve performance constraints.

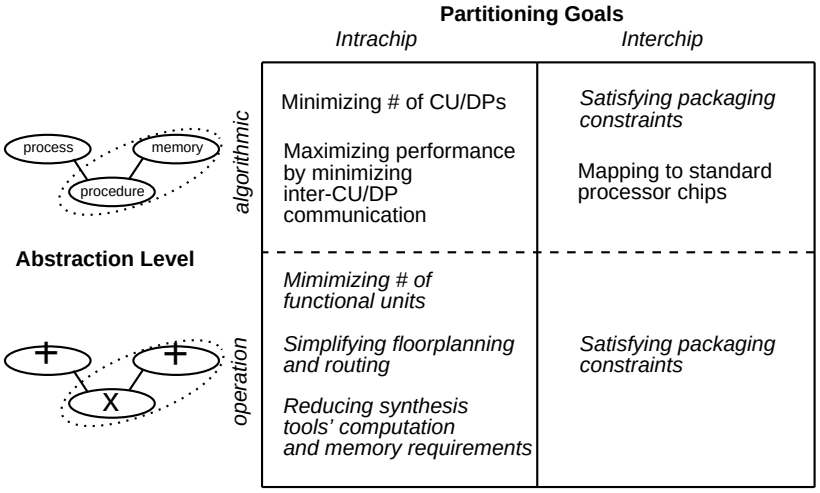


Figure 1: Behavioral-partitioning abstraction levels vs. goals

Figure 1 summarizes partitioning abstraction-levels and goals. Goals which are addressed by approaches discussed in this report are shown in italics. Current behavioral partitioning approaches are at the operation level with intrachip goals, at the operation level with the chip-packaging goal, or at the algorithmic level with the chip-packaging goal.

Publications discussing individual approaches use varying terminology and focus on different aspects, making comparison of the approaches difficult. In addition, several key assumptions are made in each approach that are sometimes not emphasized or made explicit; knowing these assumptions is crucial to understanding the applicability of each partitioning approach. For these reasons, a survey of behavioral partitioning approaches is necessary.

In this report, we first summarize the graph-theoretic foundations of partitioning. We discuss two main behavioral-partitioning goals. We then survey behavioral-partitioning systems, focusing on the mapping of the behavioral problem to a graph problem, on the algorithms used for partitioning, and on the uses of the partitioning results. Important assumptions are made explicit throughout.

2 Behavioral Partitioning: Foundations and Motivations

Partitioning is a long-studied problem, and is usually formulated using graphs. We provide a brief overview of graph definitions and basic partitioning algorithms that have proven useful in practice. We assume a familiarity with high-level synthesis; thus we simply define several terms that we will use. We then discuss the two main motivations for behavioral partitioning. The terms and algorithms introduced in this section will be used extensively throughout the remainder of this report.

For a detailed discussion of basic partitioning definitions and algorithms, see [1]. For an introduction to high-level synthesis, see [2, 3, 4].

2.1 Basic Partitioning

2.1.1 Definitions and Terminology

We define a **graph** as a set V of vertices v_i and a set E of edges $e_{i,j}$ connecting exactly two different vertices v_i, v_j. Each vertex has a value $area(v_i)$, and each edge has a value $weight(e_{i,j})$. A **hypergraph** is defined as above, except each edge connects *two or more* vertices, and is called a hyperedge. A hyperedge is denoted as $e_{i,j,...,m}$. Thus each hyperedge is a subset of two or more vertices. Hyperedges are often called nets. Hypergraphs are often called circuits or networks. **Partitioning** is the grouping of vertices of a graph or hypergraph into disjoint sets $V_1, V_2, ...V_m$. Each set is called a *partition* or *group*. We define:

$area(V_i)$	Area of a partition, equal to the sum of the areas of the vertices in V_i.
$cutsize(V_i)$	Sum of the weights of all hyperedges which connect a vertex in V_i to a vertex not in V_i.
P	The set of partitions $V_1, V_2, ...V_k$.
$\|P\|$	The number of partitions in P.

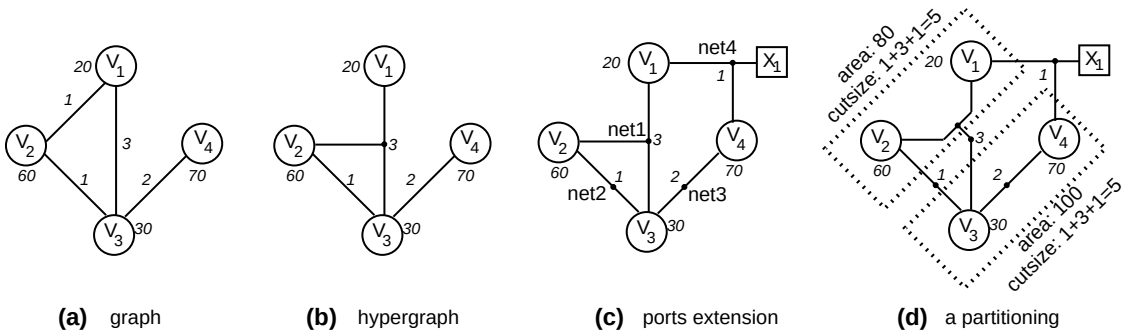


Figure 2: Graphs and partitions

Partitioning is usually subject to a set of constraints and/or an objective function. There are two types of constraints: (1) Hard constraint– a partitioning that doesn't meet the constraint is *invalid*. Although it may serve as an intermediate partitioning, it is not a valid output partition. (2) Soft constraint– a partitioning that doesn't meet the constraint is undesirable, but can be an output partition. Common hard constraints include maximum partition area ($area(V_i) < maxarea$ for all i) and maximum partition cutsize ($cutsize(V_i) < maxcutsize$ for all i), where $maxarea$ and $maxcutsize$ are area and cutsize constraints for *each partition*.

We define an objective function ($OBJFCT$) as an evaluation of a partitioning. $OBJFCT$ is usually a function of the number of partitions, the cutsizes and areas of each partition, and various other factors. Soft constraints are incorporated into $OBJFCT$. We shall assume throughout this report that we want to *minimize* the $OBJFCT$ value.

We extend hypergraphs with a set X of external ports x_i. We extend a hyperedge to be a subset of $V \cup X$. This only affects the cutsize definition. Partitioning is still a grouping of vertices only.

2.1.2 Partitioning Algorithms

There are only two basic types of partitioning algorithms:

- Constructive: creates a partitioning of a graph or hypergraph
- Iterative: improves a partitioning

We shall discuss two of the most commonly used constructive algorithms, and three of the most commonly used iterative algorithms.

Constructive Algorithms

Random Constructive Partitioning: Given a desired number of partitions, this algorithm randomly places each vertex into one of the partitions.

Algorithm 2.1 : Random Constructive Partitioning

for $i = 1$ *to numpartitions*
  initialize V_i to $\emptyset$, add V_i to P
for each v_i not in any V_j
  add v_i to a randomly selected V_j
return P

This is the fastest and simplest constructive algorithm, but results in very poor partitions. It is used to create an initial partition as input to an iterative algorithm.

Clustering Constructive Algorithm: this algorithm merges vertices connected by the heaviest edges, continuing to merge until some termination criteria is met. An edge weight can be thought of as the *closeness* between two vertices. Several clustering variations exists; the following is one common algorithm:

Algorithm 2.2 : Pairwise Cluster Partitioning
for each v_i
  initialize V_i to v_i, add V_i to P
while the **termination criteria** is not met
  find i, j with largest $weight(e_{i,j})$
  merge V_i, V_j into a new node V_{ij}
  remove all edges involving V_i or V_j
  add an edge between V_{ij} and every other V_k
  set the weight of each new edge to **newweight(i,j,k)**
return P

Common **termination criteria** include:

- $|P| \leq$ a constant
- $weight(e_{i,j}) \leq$ a constant for all i, j (*edge-weight threshold*)

Common **newweight(i,j,k)** functions include:

- $MIN(weight(e_{i,k}), weight(e_{j,k}))$
- $MAX(weight(e_{i,k}), weight(e_{j,k}))$
- $AVERAGE(weight(e_{i,k}), weight(e_{j,k}))$
- $SUM(weight(e_{i,k}), weight(e_{j,k}))$
- recomputation in the same manner as was done to obtain original edge weights

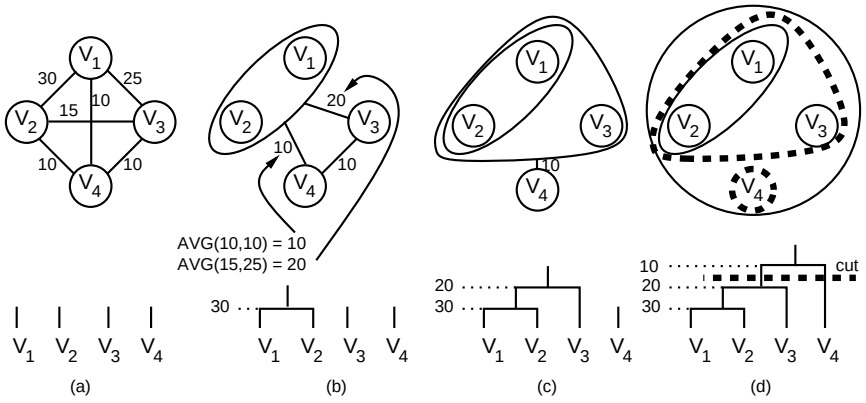


Figure 3: Building a cluster tree during pairwise clustering

Often various alternative partitionings must be explored rapidly. Repeatedly performing clustering with a different termination criteria to obtain alternative partitionings can be computationally expensive. A cluster tree can be used to reduce computation. Building a cluster tree is done by first making the termination criteria $|P| = 1$. The initial vertices are mapped to tree leaves. The above clustering algorithm is modified so each merge of v_i, v_j creates a $node_{ij}$ in the tree with children $node_i, node_j$. $weight(e_{i,j})$ becomes the distance of $node_{ij}$ from the root. A cut across the final tree (Figure 3(d)) defines a partitioning. In this way various alternative partitionings can be found by selecting alternative cut-lines, without requiring reclustering.

Iterative Algorithms

Pairwise Exchange Iterative Partitioning: this algorithm simply swaps the vertex pair that gives the largest decrease in the objective function value. This swapping is repeated until no swap gives a decrease. One problem with this algorithm is that it is trapped in the first local minimum.

Group Migration Iterative Partitioning: this algorithm is an enhancement to pairwise exchange. It permits a "bad" swap if a subsequent swap will result in a lower overall $OBJFCT$ value. The algorithm swaps the vertex pair that gives the largest decrease *or the smallest increase* in the $OBJFCT$ value. This swapping is repeated considering only vertices that have *not* been part of a swap, until there are no more vertices to consider. The minimum $OBJFCT$ value is recorded. Starting with the initial partitioning, the swap sequence is repeated until the minimum $OBJFCT$ value is reached. The entire process is then repeated until the minimum $OBJFCT$ value achievable through swapping is not lower than the value without swapping.

A group migration algorithm is shown below for improving two-way partitions ($P = \{V_1, V_2\}$). A vertex that has been part of a swap is said to be *fixed*. The algorithm uses the following variables:

- $initval$: the $OBJFCT$ value before swapping

- $minval$: the minimum $OBJFCT$ value encountered during swapping
- $bestswap$: records the vertices of the best swap encountered at a particular stage and the corresponding $OBJFCT$ value

Algorithm 2.3 : Group Migration

```
loop /* main loop */
    initval = evaluate OBJFCT for current partitioning
    minval = ∞
    create a copy P_orig of P          while all v_k in V_1, V_2 are not fixed
        bestswap.val = ∞
        for each pair v_i, v_j, where v_i ∈ V_1, v_j ∈ V_2
            currval = evaluate OBJFCT if v_i, v_j swapped
            if currval < bestswap.val
                bestswap.(i, j, val) = (i, j, currval)
        endfor
        push bestswap on a queue,
        swap v_{bestswap.i}, v_{bestswap.j}, fix both vertices
        minval = MIN(minval, bestswap.val)
    endwhile
    set P = P_orig
    if initval ≤ minval
        return P
    repeat
        pop bestswap off of queue
        swap v_{bestswap.i}, v_{bestswap.j}
    until bestswap.val = minval
endloop /* main loop */
return P
```

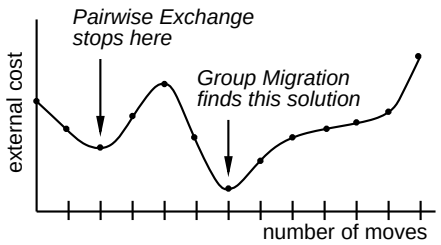


Figure 4: Group migration's local solution-space search strategy

The original Kernighan/Lin algorithm used the increase in cutsize as the $OBJFCT$. Fiduccia/Mattheyses modified the algorithm by: (1) moving a single vertex at a time, thus allowing for unbalanced partitions and non-uniform partition areas, (2) extending the external cost calculation for hypergraphs, and (3) selecting vertices in a time-saving manner. Their modifications changed the algorithm from $O(n^2log(n))$ complexity to a linear complexity. Krishnamurthy extended the algorithm to include look-ahead. For example, in Figure 5, moving v_2 to the other partition has the same effect on partition cutsizes as would moving v_1. However, moving v_2 enables a subsequent move of v_5 to reduce the cutsize. Two subsequent moves would be needed to reduce the cutsize if v_1 was first moved. Krishnamurthy's extension captures this subsequent-move information in the objective

function. Other extensions have been proposed for multi-way partitions, and for different objective functions.

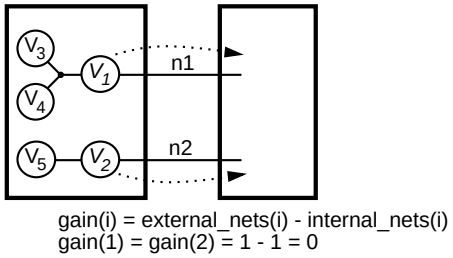


Figure 5: These seemingly equal moves can be distinguished using a group migration "look-ahead" extension

Simulated Annealing Iterative Partitioning: group migration achieves good results because it accepts a bad "move" (e.g. swap or any other change in the partitioning) if the move is part of a sequence of moves that leads to a better overall partitioning. To limit the computational complexity, the sequence of moves is limited by "fixing" each vertex after it is part of a move. The simulated annealing algorithm also accepts bad moves, but limits the sequence of moves in a different way. The tolerance for accepting bad moves is simply decreased over time.

The basic idea of the algorithm is to generate random moves, initially accepting and making many "bad" moves (i.e. those which increase the $OBJFCT$ value), and rejecting more bad moves as time proceeds, until only good moves are accepted and no further good moves are found. The initial acceptance of bad moves is intended to bring the partitioning out of local minimums. The algorithm is computationally expensive, so is usually used when partitioning quality is more important than computer runtime. Details are beyond the scope of this report; see [1].

2.2 Basic High-level Synthesis

High-level synthesis (HLS) converts a sequential behavioral specification into a structural design which implements that behavior. Common tasks are:

- *Scheduling:* Determining in which control step to perform each behavioral operation (e.g. addition, multiplation, comparison).
- *Allocation:* Designating which physical functional units (e.g. adders, comparators, registers, buses), and how many of each, to use in the structural design, and assigning behavioral operations to specific physical units (including buses).
- *Control creation:* Generating the design controller, using microcode and/or random logic, and optimizing the logic.

The structure usually consists of a **control unit** and a **datapath (CU/DP)**, with one or the other possibly being empty or very small. Follow-up tasks may include technology mapping, floorplanning, placement, and routing. A "sequential behavior" is a behavior describable by sequential program control constructs, such as loop and case statements, and procedure calls. It corresponds to a *single process* behavior, as opposed to a multiple process behavior, in hardware description language terminology.

A behavior is usually specified using a hardware description language, such as VHDL. Most HLS systems convert this to an internal representation of the behavior, called a **control/data-flow graph (CDFG)**. Figure 6 shows a behavioral specification and a corresponding sample CDFG. The control portion of the graph contains square and triangular shaped nodes in our notation, and directed arcs.

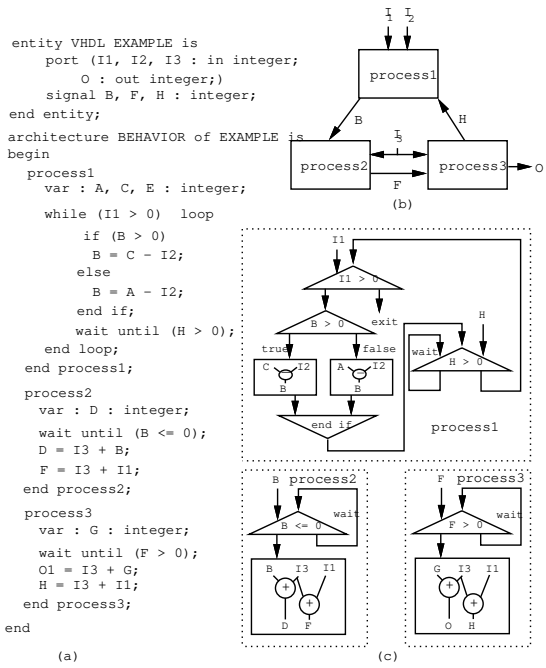


Figure 6: A sample mapping of a behavior to CDFGs

The dataflow portion is shown with circular nodes and undirected edges. The graph formed by the dataflow portion only is refered to as a dataflow graph **(DFG)**.

Partitioning can be performed at various levels of abstraction. It can be applied to the synthesized structure, to the synthesized logic, to the DFG only, to the DFG with consideration of information in the CDFG, to the CDFG, or to specification itself.

2.3 Motivations for Behavioral Partitioning

To date, behavioral-partitioning systems have focused on one of two goals:

- *Tractability (Intrachip):* Converting difficult HLS or follow-up problems into manageable ones.
- *Packaging-constraint satisfaction (Interchip):* Creating structure which can be implemented with a specific chip technology

2.3.1 Partitioning for Tractability

The goal is to tradeoff the solution space size with CPU time and memory requirements. The approach is to divide a problem with a large solution space into several problems with smaller solution spaces, the totality of the smaller spaces being substantially smaller than the large space. For example, consider an algorithm of computational complexity $O(n^k)$, where k is a small constant, intended to solve a particular problem such as scheduling, allocation, or logic synthesis. Assume this means that n^k computations are performed by the algorithm. If the problem is divided into p parts, then $p(\frac{n}{p})^k$ computations are required. Thus the ratio of the computations performed before dividing the problem to those performed after is: $\frac{n^k}{p(\frac{n}{p})^k} = p^{k-1}$ This is a constant factor, so the *theoretical* computational complexity is unchanged. However, this a very significant *practical* change in the number of computations required. For example, consider using an n^3 algorithm ($k = 3$). The "speedup" obtained by dividing a problem into four parts ($p = 4$) is: $4^{3-1} = 16$. This can mean the difference between 15

minutes and 4 hours. Of course, the computations required to perform the partitioning must also be considered. Therefore, partitioning for tractability will usually use fast partitioning algorithms, such as clustering or group migration.

CPU time and memory can also be reduced by using less complex, inferior algorithms on the original, unpartitioned problem. The result is a tractable problem but less of a chance of finding a good solution. Partitioning permits better algorithms to be applied. But a poor partition, e.g. random, also results in a tractable problem with less of a chance of finding a good solution. The former may search a large solution space unthoroughly, the latter may search a greatly reduced solution space very thoroughly. Both may not result in good solutions.

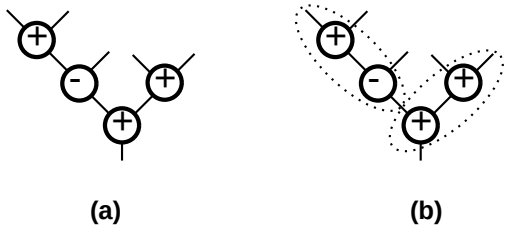


Figure 7: A simple behavior for which functional units must be allocated

Consider a functional unit allocation algorithm that is independent of scheduling algorithms. In the example of Figure 7(a), a behavior with three addition operations and one subtraction operation is shown. If a component library contains an adder, a subtractor, and an adder/subtractor, any number and combination of which may be used, then the algorithm must choose between approximately 1148 reasonable valid allocations (see Appendix A.1). By "reasonable" we mean allocations which don't have excess functional units, such as three adders and two subtractors. If the operators are clustered into two groups as in Figure 7(b), and then the allocation algorithm is performed separately on each group, there are only $10 + 14 = 24$ reasonable possibilities. This is because operators in separate clusters can't share the same functional unit, which greatly reduces the number of possible combinations of units. The clustering algorithm should thus insure that operators are separated into different clusters only if their sharing a functional unit would be an inferior solution. Conversely stated, the algorithm should attempt to group operators in the same cluster that can beneficially share a functional unit. An important consideration of partitioning for tractability is thus to partition in such a manner that a good solution still exists in the reduced search space. If this is done, then it is much more likely that this good solution will be found, so partitioning may actually *improve* the design quality. The partitioning for tractability approaches discussed in this report concentrate and differ from one another primarily in the specific method of trying to keep a good solution in the reduced solution space resulting from partitioning.

Improving design quality is often the stated goal of partitioning. As noted above, such partitioning has its roots in tractability. Specifically, any solution achieved using a partitioning approach can theoretically also be achieved without partitioning, except that CPU and memory use would be unacceptable for practical use.

2.3.2 Partitioning for Packaging-Constraint Satisfaction

The structure output of HLS must eventually be implemented as a chip set. Chips have limited capacities of silicon area or available gates or transistors, and of the number of pins. If the structure will not fit on a single chip or requires too many pins, it must be partitioned. The goal of behavioral partitioning for packaging-constraint satisfaction is to enable HLS to output structural groups, each

group implementable as a single chip, as opposed to the structure itself being partitioned (see Figure 8).

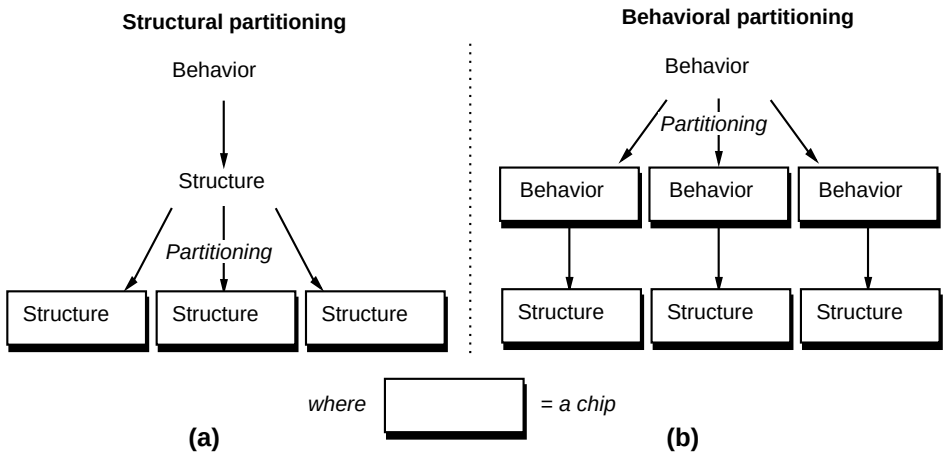


Figure 8: Behavioral vs. structural partitioning

Initially, this approach seems to be less desirable than structural partitioning since estimations of structure size are less accurate at higher abstraction levels. For example, consider a transistor schematic. Assuming custom layout implementation, the exact layout area needed for the schematic is unknown until layout is actually obtained, since placement, routing, and compaction all affect the layout area. Partitioning a transistor schematic among chips thus requires an area estimator. Now consider a register-transfer level netlist; increased area estimation error occurs from not knowing the exact results of technology mapping, logic optimization, and layout.

At the behavioral level, the results of scheduling, allocation, and controller generation are also unknown, leading to even greater estimation error than from the structural level. However, the significance of this error may be outweighed by several advantages to partitioning before structure. Such advantages involve:

- Architectural decisions: it is incorrect to assume that the structure generated by HLS is independent of that structure's distribution among chips. Scheduling and allocation are influenced by the partitioning of the behavior into chip behaviors. For example, allocation may use fewer or smaller functional units to meet a partition's area constraint, or it may choose a faster unit to make up for interchip communication delay with another partition. Non-essential behavior may actually be modified to account for pin constraints; for example, a parallel data transfer between two partitions may be changed to two transfers of half the data, using a latch to store the first half. These tradeoffs are difficult to consider at the structural level.

- Inherent groupings: the specification may provide natural groupings (e.g. procedures) which may correspond to good partitions.

- Behavioral chip specifications: since the structure of each chip is generated from the behavior, this behavior serves as the chip specification. This can aid both functional testing and future redesign.

- Fewer objects: a behavioral description can capture a system with fewer objects than a structural description. This can speed partitioning and enable designer control over partitioning decisions.

3 A Survey of Behavioral Partitioning Systems

Several approaches exist for partitioning a behavior before the final structure is synthesized. The following is a survey of these approaches. For each system, we shall indicate the following information:

- *System context:* a brief description of the overall system in which the partitioning effort is embedded.
- *Subproblem and goal:* the subproblem in the system to which partitioning is applied, and the goal of that partitioning.
- *Approach:* a brief description of how partitioning is applied to the subproblem to obtain the goal.

- *Abstraction level:* a rough categorization of the level of abstraction on which partitioning is applied.
- *Mapping:* the mapping of the objects of the abstraction level to a graph or hypergraph model, or a variation of one of these models.
- *Algorithm:* the partitioning algorithm(s) used.
- *Notes:* any miscellaneous key information about the system.
- *References:* references to publications for the system

This is followed by a more detailed description of the partitioning approach, and is usually accompanied with an example. There are three important aspects of each approach that should be focused on: (1) the input level, (2) the mapping of input to partitioning objects, and (3) the use of the partitioning results. Figure 9 summarizes these three aspects for the approaches surveyed in this report; we shall refer again to this figure at the end of the survey.

	Input level of abstraction	Partitioning objects	Use of partitioning results
SPECPART	behaviorally-hierarchical specification*	behaviors and storage elements	info. to specification refinement tool
VULCAN	sequential, hierarchical CDFG	CDFG nodes**	info. to HLS tool
APARTY	sequential CDFG	DFG and CFG operations	info. to HLS tool
BUD	sequential CDFG (single procedure)	DFG operations, using CFG information	estimation and info. to HLS tool
YSC	logic w/ DFG-like operations	logic and DFG-like operations	divides input of logic synthesis tool
CHOP	DFG (acyclic) and memories	DFG operations and memories	info. to DFG HLS tool

* a hierarchy of sequential and concurrent behaviors, (e.g. processes, procedures, substates) plus storage elements
** requires combined control/datapath target architecture

Figure 9: Comparison of behavioral-partitioning approaches

3.1 YSC – The Yorktown Silicon Compiler

- *System context:* To synthesize a sequential behavior into structure consisting of storage units and combinational logic.
- *Subproblem and goal: Tractability* – reduce the runtime and memory requirements of logic synthesis applied to the combinational logic, and possibly improve follow-up floorplanning quality
- *Approach:* The combinational logic is partitioned, with logic synthesis then run on each partition separately.
- *Abstraction level:* Combinational logic, containing atomic operations from the behavior (AND, ADD, SHIFT, EQUAL, etc.).
- *Mapping:* Graph model, where each vertex represents an operation, and each edge represents a closeness.
- *Algorithm:* Clustering
- *Notes:* The partitioning uses knowledge of the logic optimization capability of each pair of operations ("similarity information") to ensure that the reduced search space contains good solutions.
- *References:* [5, 6, 7]

In Figure 10(a), a sample behavioral input and the resulting structure is shown. The behavioral language is not any one in particular. Note that the synthesized structure can be divided into three parts: ports (a, b, clk, y, z), storage (x, c), and logic operations $(+, =, -, <)$, as shown in Figure 10(b). Logic synthesis maps the operations to gates available in the target technology, such as two-input NOR gates, and optimizes the logic. An example of optimization is converting two AND gates that have the same inputs into a single gate.

Partitioning the logic improves the tractability of the logic synthesis problem. The approach is to cluster the logic into groups, and apply logic synthesis to each group separately. To attain a good solution, the closeness function attempts to merge highly connected pieces of logic while maintaining balanced partition areas. The function is:

$$C(V_i, V_j) = \left(\frac{k_1 \times inputs(V_i, V_j) + wires(V_i, V_j)}{maxall\,(k_1 \times inputs(V_x, V_y) + wires(V_x, V_y))} \right)^{k_2} \times \left(\frac{limit}{min\,(size(V_i), size(V_j))} \right)^{k_3} \times \left(\frac{limit}{size(V_i) + size(V_j)} \right) \quad (1)$$

where:

V_i	the i'th group of operations
$inputs(V_i, V_j)$	the number of common inputs shared by groups V_i and V_j,
$wires(V_i, V_j)$	the number of output to input and input to output connections between groups V_i and V_j,
$size(V_i)$	the estimated size of group V_i (in number of transistors),
$maxall(e(V_x, V_y))$	the maximum value of expression e for all group pairs $V_x, V_y, x \neq y$,
$min(x, y)$	the minimum of x and y,
$limit$	a desired size limit constant,
k_1, k_2, k_3	constants

The first term of the equation favors merging groups which share common data, i.e. are highly connected. The second term favors merges that involve a small group, which aids in creating balanced partitions. The third term attempts to prevent any single partition from greatly exceeding a given limit.

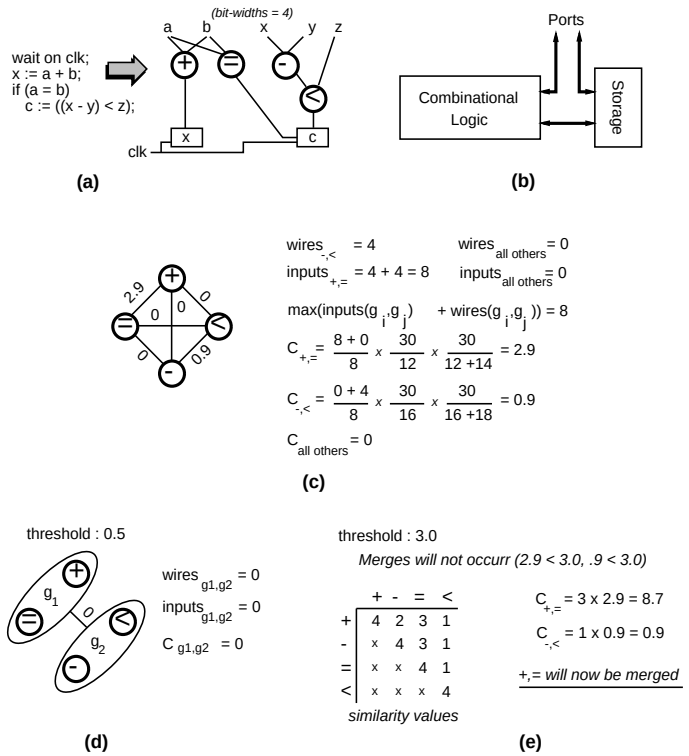


Figure 10: Partitioning before logic synthesis

As given above, the partitioning considers only the structural aspects of the logic. Knowledge of the logic synthesis task can be used to improve the overall results. Specifically, particular pairs of operations are more amenable to logic synthesis than other pairs. For example, consider an = operation which compares two bit-vectors for equality. Equality can be determined by ANDing the complements of the exclusive-OR of all bit pairs. A + operation might be implemented using an exclusive-OR to generate the sum value for each bit pair. If the two operations have the same inputs, then logic optimization would ideally share the exclusive-OR gates of these two operations, reducing the logic required.

Similarity of two atomic operations is thus defined as the amenability of the operation pair to successful logic optimization. Similarity can be computed by attempting logic synthesis for various configurations of each pair. Results can be stored as values in a similarity table. During partitioning, the closeness values as determined above can be multiplied by this similarity value, i.e.:

$$C(V_i, V_j)' = similarity(V_i, V_j) \times C(V_i, V_j) \tag{2}$$

As an example, consider Figure 10(c). The closeness value computations are shown for the example in Figure 10(a) using the original cost function, which does *not* consider similarity information. The graph model on which clustering will be applied is also shown. In Figure 10(d), clustering for an edge weight threshold of 0.5 is shown. Note that operations + and = were grouped, based *solely* on their interconnectivity and size. Consider if the threshold was increased to 3.0 (or conversely, if the size limit was reduced). Since no closeness value exceeds this, clustering would not have grouped

any operations. However, we intuitively know that + and = are excellent candidates for merging, as discussed above. This knowledge is accounted for by using the similarity value. In Figure 10(e), each closeness value is multiplied by the corresponding similarity value. This results in the closeness value of + and = to exceed 3.0, so that a subsequent clustering would group these two operations, as desired.

3.2 BUD – Bottom-Up Design

- *System context:* To synthesize a sequential behavior into structure consisting of a control unit and one or more datapaths.
- *Subproblem and goal: Tractability*, leading to better designs – use estimations of the eventual area/time characteristics of the structure as an aid to making scheduling and allocation decisions.

- *Approach:* Behavioral operations are partitioned, with allocation performed separately on each group. The area for each group is estimated, and the area of the entire structure estimated through floorplanning. Scheduling is done, a clock is determined, and the average cycle time is computed. Area and time characteristics for various partitionings can be rapidly evaluated in this manner. A partitioning essentially determines a scheduling and allocation, thus conveniently encapsulating the key HLS decisions.
- *Abstraction level:* DFG, using CDFG information to guide the partitioning.
- *Mapping:* Graph model, where each vertex represents an operation that must be bound to a functional unit, and each edge represents a closeness.
- *Algorithm:* Clustering, making use of a cluster tree.
- *References:* [8, 9, 10]

The motivation for developing BUD is that physical design characteristics, such as placement and routing, play an essential role in the area and delay of a structural design. System designers, it is observed, make heavy use of such information in making design decisions. Therefore, synthesis tools should also incorporate such "bottom-up" information when transforming behavior into structure. The goal is to provide accurate estimates of the area and delay for a given behavior throughout HLS. The issue is then to perform HLS tasks in a manner that permits accurate estimation. For example, given only a set of functional-units, interconnect estimations will be extremely inaccurate due to the large number of possible bindings of operations to functional units; hence an approach which first selects functional units for the entire behavior is not amenable to estimation.

The approach taken in BUD is to perform HLS tasks by partitioning the behavioral operations of a CDFG. Since BUD's allocation algorithm is simple and is applied to each partition separately, *the partitioning decisions encompass the major tradeoffs of the design.* In addition, the partitions make likely structural objects, so the area of each object, as well as that of the collection of these objects, can then be estimated with accuracy. By choosing various partitions, a portion of the design space can be rapidly explored.

BUD takes as input a CDFG and an area/time $OBJFCT$, among other items. The input CDFG consists of a single acyclic CDFG ("procedure" or "vtbody"). BUD also has access to a database which returns detailed functional unit structural information, such as area, height, width, and delay. BUD's overall algorithm is described below. The loop can be exited at any time.

- **STEP 1**: Select the DFG operations that must be bound to a functional unit (e.g. $+, -, =$). Make each operation a graph vertex. For every vertex pair, create an edge with the weight being the closeness $C(v_i, v_j)$ (defined below).

- **STEP 2**: Build a cluster tree, using $AVERAGE$ as the *newweight* function.
- **STEP 3**: For each tree level, starting at the root, loop
  - Set the partitions to those determined by a cut at this level.
  - Estimate area/time for this partitioning (see below).
  - Calculate the $OBJFCT(area, time)$ value. Store this information.
- **STEP 4**: Choose the best partitioning and generate the output structure through scheduling and allocation.

The closeness function is defined as:

$$C(v_i, v_j) = \left(\frac{fcost(v_i) + fcost(v_j) - fcost(v_i, v_j)}{fcost(v_i, v_j)} \right) + \left(\frac{commconn(v_i, v_j)}{totalconn(v_i, v_j)} \right) - N \times (par(v_i, v_j)) \quad (3)$$

where:

v_i	the i'th operation,
$fcost(v_i, v_j, v_k, ...)$	the cost, based on delay and area, of the minimal number of functional units needed to perform all the given operations,
$commconn$	the number of dataflow connections shared by v_i and v_j,
$totalconn$	the total dataflow connections to either v_i or v_j,
$par(v_i, v_j)$	1 if v_i and v_j can be done in parallel, 0 otherwise,

The first term in the equation favors merging operations which have lower area/delay cost using the same functional unit than when using separate units. For example, a two's complement adder/subtractor merely complements one input to change an addition to a subtraction. Thus the area of such a unit is less than the sum of the areas of a separate adder and subtractor. The second term favors merging operations which use common data. This reduces routing area. The third term tries to avoid merging operations that can be executed concurrently, since BUD's scheduler always schedules merged operations sequentially.

The equation can be enhanced by weighing each term by its significance to the overall design. The first term can be multiplied by the area of the functional unit needed to perform operations v_i, v_j divided by the total area of the design, so that large scale merges are more likely than small ones. For example, a merge which results in a $1000mm^2$ functional unit has more effect on the overall area than a merge which results in a $10mm^2$ unit. The third term of the equation can be multiplied by the probability of either v_i or v_j being executed in one major cycle of the hardware divided by the average number of steps in the cycle. This relaxes the parallelism goal for seldomly used operations, since they have little effect on the average cycle time.

Given a partitioning, BUD allocates each partition using the minimal number of function units needed to perform all the operations in that partition. The CDFG is scheduled with the restriction that operations in the same cluster can not be scheduled into the same control step (unless chaining is used). Given this information, estimations are possible. The details of estimation are beyond the scope of this report. However they are an integral part of BUD so we shall briefly overview the approach used.

The registers needed to hold values between control steps are determined. The length and width, and hence the area, are determined from the registers, functional units, multiplexers, and wiring

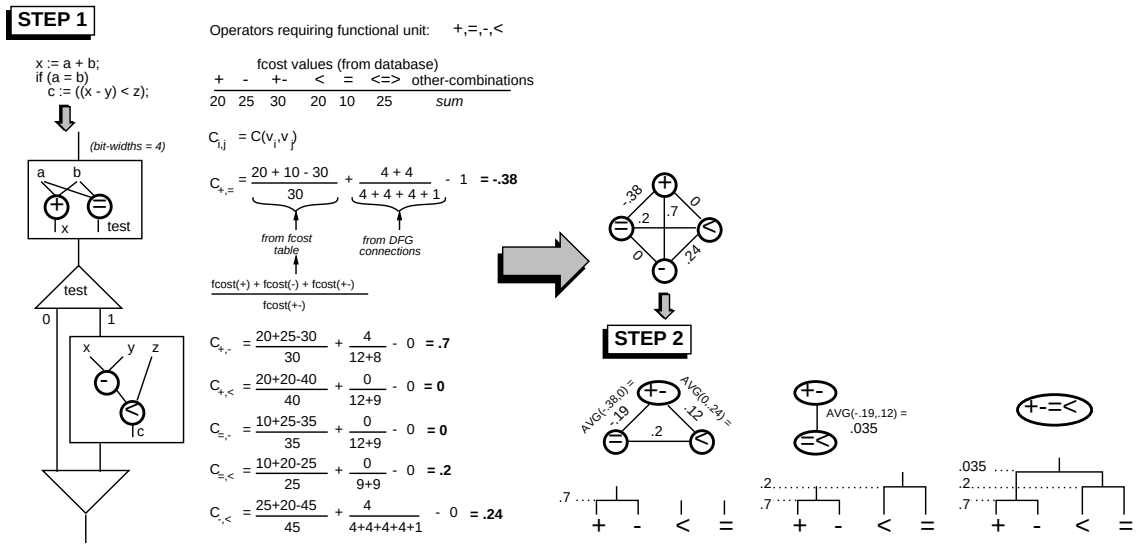
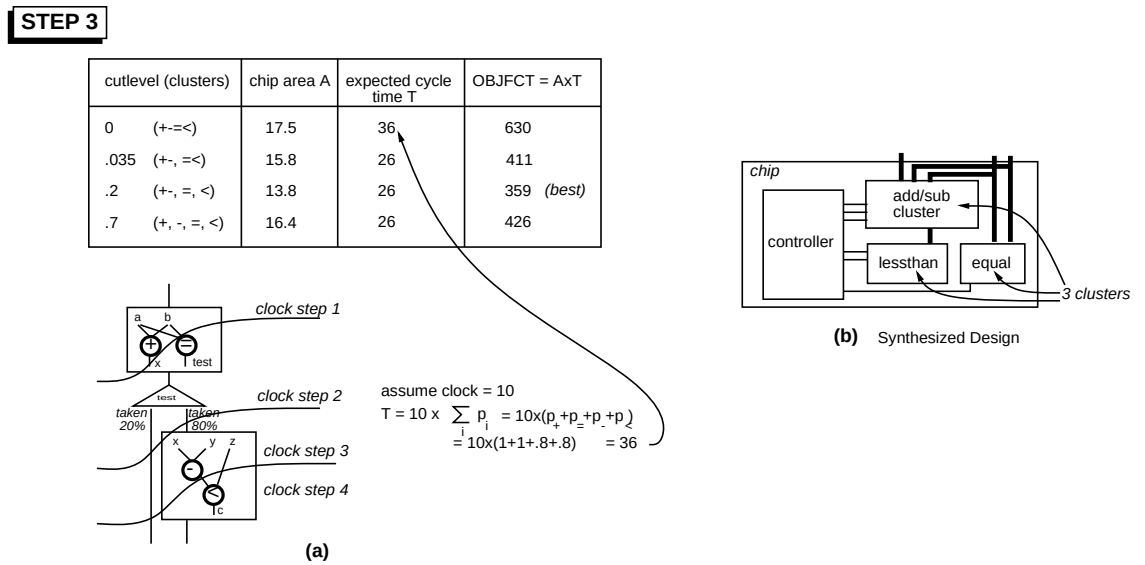


Figure 11: Building a cluster tree in BUD for behavioral operations

needed within the cluster, assuming a particular layout style. A floorplanner places these cluster objects on a chip. The bounding box of these placed objects is the estimated area A.

The clock cycle is estimated as the maximum delay through the datapath for any control step. Contributors to delay include functional units, multiplexors, registers, and wires. The average cycle time T is then $clock \times \sum_i p_i$, where p_i is the probability that control step i is executed in a major cycle of the design. These probabilities are determined from branch probabilities provided as input to BUD. Once A and T have been estimated, $OBJFCT(A,T)$ can be evaluated.



cutlevel (clusters)		chip area A	expected cycle time T	OBJFCT = AxT	
0	(+-=<)	17.5	36	630	
.035	(+-, =<)	15.8	26	411	
.2	(+-, =, <)	13.8	26	359	*(best)*
.7	(+, -, =, <)	16.4	26	426	

Figure 12: Selecting a partitioning and creating partitioned structure in BUD

In Figure 11, a simple behavior is described, and the first two steps of BUD's algorithm are shown. In Figure 12, the third step is shown, which involves computing the value of $OBJFCT$ for various partitions determined by different cuts of the cluster tree, where the details of estimation have been omitted. As an example of expected cycle-time estimation, consider the first row of the table in Figure 12. Since all operations are in the same cluster, none will share the same clock step. Thus

four cycles are needed. The expected cycle time of 36 was computed using the fact that the right branch of the CDFG is taken 80% of the time. For the other three rows of the table, + and = are in separate clusters so can be done in parallel, resulting in a shorter expected cycle time. For the given $OBJFCT$, it is found that clustering + and − is beneficial, due to the sharing of a functional unit and reduced wiring. However, merging = and < is not beneficial, since the advantage of using a single functional unit is outweighed by the need for additional multiplexors, the wiring between clusters, and the resulting floorplan aspect ratio.

3.2.1 Synthesis by Delayed Binding of Decisions

BUD's clustering approach was also used in [11] as part of an effort to perform HLS without making important decisions concerning scheduling and allocation independently. The clustering serves to prune the solution space to promising designs, therefore improving design quality through tractability. Minor variations from BUD's approach include differing assumptions of functional unit sharability and consideration of more than one "procedure".

3.3 APARTY – Architectural Partitioning

- *System context:* To synthesize a sequential behavior into structure consisting of one or more control units and one or more datapaths.
- *Subproblem and goal: Tractability* leading to better designs, indirectly addressing *packaging constraints*— create structural partitions from a behavior to guide HLS tasks in a manner which improves area and performance of the resulting design.
- *Approach:* Behavioral operations are clustered, using an algorithm in which clustering and tree-cutting can be iterated. During each iteration, the clusters resulting from the previous cut serve as the input to the next clustering. Any one of several closeness functions can be used during an iteration. Simple allocation and/or scheduling can be done to estimate cluster area, interconnection requirements, or schedule length. Various partitionings can be rapidly evaluated in this manner.
- *Abstraction level:* CDFG
- *Mapping:* Graph model, where each vertex represents an operation that must be bound to a functional unit or a control branch operation, and each edge represents a closeness.
- *Algorithm:* Clustering, making use of a cluster tree, and permitting multiple stages of clustering with different closeness functions.
- *References:* [12, 13, 14, 15]

There are two apparent limitations in the BUD approach which APARTY seeks to overcome:

1. As a cluster tree is built, the closeness of two clusters will contain some error since it is not actually recomputed, but is instead taken as an average of the weights between the cluster members.

2. The criteria of interconnect, functional-unit sharability, and potential parallelism are all incorporated as terms of a single closeness function. It may be difficult or impossible to balance the relative weights of these terms to achieve the desired design.

The first limitation is solved by clustering in stages. Each time a cutline is selected, the resulting clusters can serve as the basis of a new clustering. Thus, a new graph model must be created, with edge weights corresponding to the closeness between a pair of clusters. This requires that closeness be defined between *groups* of operations (clusters), rather than just between pairs of operations. An

example of this extension to traditional clustering is shown in Figure 13. Figure 13(a) is taken directly from Figure 3(d).

The second limitation is addressed by providing a variety of closeness functions, each of which concentrates on a specific criteria (possibly different from those in BUD). Since clustering is now done in stages, each stage can use any one of these functions. In addition, different $OBJFCT$s can be applied at each stage.

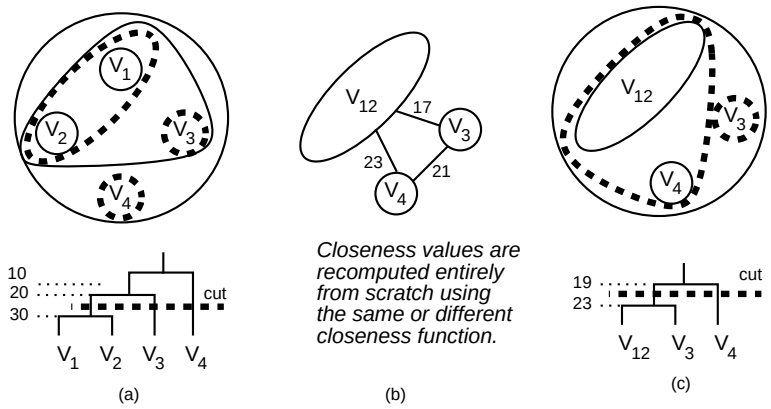


Figure 13: Multistage clustering

APARTY takes as input a CDFG and a set of physical constraints, among other items. The input CDFG consists of a set of acyclic CDFG's ("procedures" or "vtbodies"), and their possibly cyclic relationships. Constraints are also specified (e.g. maximum partition area or maximum schedule length). APARTY has access to technology information, such as area per bit used for an operation. The partitioning methodology is described below.

- **STEP 1**: Choose a closeness function $C(v_i, v_j)$ (described below). Make each object (clusters if the CDFG is already partitioned, operations otherwise) a graph vertex. For every vertex pair, create an edge with the weight being the closeness $C(v_i, v_j)$.
- **STEP 2**: Build a cluster tree, using MIN or MAX as the *newweight* function.
- **STEP 3**: For each tree level, usually starting at the leafs, do:
  - Set the partitions to those determined by a cut at this level.
  - Choose a cutline criteria (area, interconnect, or time), and estimate its value (see below).
  - If the estimated value exceeds a physical constraint, go to the next level. Otherwise, calculate the $OBJFCT(criteria)$ value. The actual $OBJFCT$ details are up to the designer to decide.
- **STEP 4**: Choose the "best" partitioning. Repeat STEP 1 using the partitioned CDFG, or terminate if decided by the designer. The output structure can be generated using HLS tools that use the partitioning information as a guide.

APARTY's closeness functions focus on one of three goals:

- *Control Transfer Reduction:* Reduce the number of times that control is passed between partitions, thus improving performance if interpartition delay is large. This assumes that multiple controllers can be generated for the partitions.
- *Data Transfer Reduction:* Reduce the interconnections required for data transfer between clusters, thus reducing the data lines between partitions and perhaps improving performance.

- *Hardware Sharing:* Reduce the overall hardware used by sharing functional units.

There are five such functions defined in APARTY:

Control closeness of operations:

- *Goal: Control Transfer Reduction.*
- *Closeness Function:*

$$C(v_i, v_j) = P(v_j|v_i) \tag{4}$$

  where $P(v_i|v_j)$ is the probability that operation j is executed given that operation i is executed, and both operations belong to the same acyclic CDFG. For example, in Figure 12, $P(v_+, v_=) = 1$ and $P(v_+, v_-) = .8$.
- *Favors merging:* operations that are likely to both be executed in a single pass through an acyclic behavior.

Data closeness of clusters:

- *Goal: Data Transfer Reduction.*
- *Closeness Function:*

$$C(V_i, V_j) = \frac{commconn(V_i, V_j)}{totalconn(V_i) + totalconn(V_j)} \tag{5}$$

  where *commconn* and *totalconn* are defined as in Section 3.2, extended for a pair of operation groups, rather than just a pair of operations. Note that the function is $C(V_i, V_j)$, and not $C(v_i, v_j)$, since we are dealing with clusters, not operations. See Figure 11 for an example.
- *Favors merging:* clusters which would otherwise require many data lines between them for passing data.

Control closeness of clusters:

- *Goal: Control Transfer Reduction.*
- *Closeness Function:*

$$C(V_i, V_j) = P(V_i \cap V_j) = P(V_i) \times P(V_j|V_i) \tag{6}$$

  $P(V_i)$ is the probability that an operation in cluster V_i is activated, where each operation may belong to any of the acyclic CDFG's.
- *Favors merging: clusters* (as opposed to operations) that are likely to both be executed in a single pass of the sequential behavior.
- *Notes:* (1) This function considers cluster pairs rather than just operation pairs. (2) There may be cyclic relationships between any acyclic CDFG's. Thus, the additional $P(V_i)$ factor is required to more heavily weight clusters that contain commonly executed operations. (3) $P(V_i \cap V_j)$ also equals $P(V_j) \times P(V_i|V_j)$. This value may differ from that given above, since calls between procedures are not necessarily symmetric. APARTY uses the maximum of the two possible values. (4) Since APARTY does not currently take branch probabilities as an input, the above probabilities are estimated statically.

Parameter data closeness of clusters:

- *Goal: Data Transfer Reduction.*
- *Closeness Function:*

$$C(V_i, V_j) = \frac{CommCalls(V_i, V_j)}{\sum_k ExternCalls(V_i, p_k) + \sum_k ExternCalls(V_j, p_k)} \tag{7}$$

  where $CommCalls(V_i, V_j)$ is the number of procedures called by both V_i and V_j, p_k is a procedure, and $ExternCalls(V_i, p_k)$ is the *total* number of calls, made from *anywhere*, to the procedure p_k, if p_k is called in V_i (otherwise it is zero).

- *Favors merging:* clusters which would otherwise require many data lines for passing procedure parameters between themselves or to another cluster
- *Notes:* the denominator terms decrease the closeness value if a common procedure is also called from many other clusters. Conversely stated, the closeness of two clusters is increased if some procedure is called only by those two clusters.

Functional unit sharability of operations:

- *Goal: Hardware Sharing.*
- *Closeness Function:*

$$C(V_i, V_j) = \frac{\sum_{v_k \in V_i} \left(\bigvee_{v_l \in V_j} D(v_k, v_l) \right) size(V_i) \; + \; \sum_{v_k \in V_j} \left(\bigvee_{v_l \in V_i} D(v_k, v_l) \right) size(V_j)}{size(V_i) + size(V_j)} \tag{8}$$

where:

$D(v_k, v_l)$	$f(v_k, v_l) \wedge g(v_k, v_l)$
$f(v_k, v_l)$	1 if v_k, v_l are scheduled into different control steps, 0 otherwise
$g(v_k, v_l)$	1 if v_k, v_l can share a functional unit, 0 otherwise

- *Favors merging:* Operators that can share the same functional unit. It discourages merging operations that are scheduled concurrently, since otherwise the operations would have to be rescheduled sequentially to execute on the same functional unit, thus negatively affecting performance.
- *Notes:* The CDFG must have been preliminarily scheduled.

Several possible $OBJFCT$s can be used to select a cutline, based on estimates of the area per cluster, the cluster interconnect, or the schedule length. *Area evaluation:* Each cluster is assumed to use the minimum required number of functional units. Contributors to a cluster's estimated area are the functional units and multiplexors. A maximum area constraint is an input to the system. *Interconnect evaluation:* Calculated as the average number and size of external data values accessed per cluster. Clusters whose area is less than a minimum area constraint are ignored. Low interconnect is prefered. *Schedule length:* Each cluster is assumed to use the minimum required number of functional units. The partitioning information is considered when scheduling. A maximum schedule length constraint is an input to the system.

For APARTY's built in $OBJFCT$s, if more than one cutline is valid for area or schedule length evaluation, then the highest one is chosen. This assumes that higher cuts will encourage shared functional units and thus result in lower overall area, assuming that interconnect area can be ignored.

Partitioning in APARTY consists of choosing the number of clustering stages, and selecting a closeness function and $OBJFCT$ to be applied at each stage. The standard configuration is:

1. Closeness Function: Control closeness of operations
   $OBJFCT$: highest cutline meeting maximum area constraint
2. Closeness Function: Data closeness of clusters
   $OBJFCT$: highest cutline meeting maximum area constraint with minimum average data interconnections per partition
3. Closeness Function: Control closeness of clusters
   $OBJFCT$: highest cutline meeting maximum area constraint
4. Closeness Function: Parameter data closeness of clusters
   $OBJFCT$: highest cutline meeting maximum area constraint

5. Closeness Function: Functional unit sharability of operations
   $OBJFCT$: highest cutline meeting maximum area constraint

This obtains what is often called instruction-set partitioning. Alternatively, data partitioning can be achieved by using data closeness functions only, perhaps ending with the functional-unit sharability function to reduce hardware. Data partitioning focuses on reducing data interconnect.

The CDFG is passed to HLS tools along with the partitioning information. Currently, the tools used will generate a single controller for the design. Ideally, partitions might contain their own controllers. This requires general transformations to be applied to the CDFG such that a partition is made into a separate process. No such transformations currently exist.

3.4 Workbench Behavioral Transformations

- *System context:* To synthesize a sequential behavior into structure consisting of one or more control units and one or more datapaths.
- *Subproblem and goal: Tractability* leading to better designs, and *packaging constraints* – To divide the sequential behavior into concurrent behaviors (processes) such that HLS can accommodate behavioral chip partitioning or improve area and performance of the resulting design.
- *Approach:* Partitioning choices are entirely up to the designer. The system provides a set of CDFG transformations for converting a a subset of possible CDFG partitionings into processes. If an original single process is too large to fit on a chip, transformations can create multiple smaller processes to be distributed among multiple chips. Another transformation organizes processes in a manner that indicates to the scheduler that each process is a stage in a pipeline, which may improve performance.
- *Abstraction level:* CDFG
- *Mapping:* None – partitioning is up to the designer.
- *Algorithm:* None – partitioning is up to the designer.
- *Notes:* This research effort focuses on the details of the CDFG transformations to create partitions that represent processes and pipestages, and on the usefulness of such transformations. Deciding what the partitions should be is a task left to the designer. Although the transformations are quite useful, they are beyond the scope of this survey, which focuses on the task of partitioning decisions. Note that the transformations are not general enough to be used to generate processes for all possible CDFG partitionings that may be output by a CDFG partitioning system.
- *References:* [15, 16, 17]

3.5 Vulcan – Partitioning of Functional Models

- *System context:* To synthesize a sequential behavior into structure consisting of multiple interconnected combined controller/datapaths.
- *Subproblem and goal: Packaging constraints* – partition the behavior such that the structure synthesized for each partition meets chip-packaging constraints while also meeting an overall schedule length constraint.
- *Approach:* In the representation used, CDFG control nodes can be hierarchically composed of CDFG's. By assuming a combined controller/datapath target architecture, the hierarchical CDFG maps easily to a hierarchical hypergraph model, which is then partitioned using modified hypergraph partitioning algorithms. Focus is on creating $OBJFCT$'s which efficiently recompute area, pin, and schedule length estimates after each move in an algorithm. Initially, vertices of a single hierarchical level in the hypergraph are considered; if constraints can not be met, a large vertex is decomposed and iterative partitioning is applied.

- *Abstraction level:* CDFG
- *Mapping:* Hierarchical hypergraph model with edges and hyperedges. Each hierarchical CDFG node is mapped to a hierarchical vertex, and each control or data dependency edge is mapped to an edge. Hardware sharing between CDFG nodes is represented as a hyperedge.
- *Algorithm:* Modified group migration and simulated annealing
- *Notes:* In this synthesis approach, a different CDFG representation is used. Each CDFG node is synthesized into a separate finite-state machine, thus generating multiple combined controller/datapaths.
- *References:* [18, 19]

In Figure 14, a sample CDFG is shown. The HLS approach used maps each CDFG control node to a controller/datapath pair. Each controller is either *waiting*, *active*, or *done*. When *waiting*, the controller waits for all controllers of predecessor CDFG nodes to be *done*. At this time it becomes *active* and begins executing its corresponding operation. When the operation is complete, it is *done*. It only returns to *waiting* if the entire CDFG is *reset* (e.g. after one major cycle of the hardware).

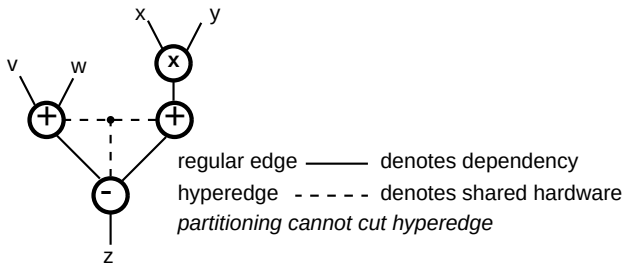


Figure 14: The extended hypergraph model in Vulcan

Creating a separate controller per CDFG node is relevant because each node represents an entity with a specific interface (*done* and *reset* control signals, and data). Thus each node has an estimated size and an interconnect with other nodes, so maps easily to a hypergraph partitioning model.

The inputs to VULCAN are a CDFG, area, pin and schedule length constraints, a clock cycle time, and sets of nodes which share the same hardware, among other things. The overall algorithm is as follows:

- **STEP 1**: Map the hierarchical CDFG to a hierarchical graph model. If several CDFG nodes will be implemented with the same hardware, create a hyperedge between the corresponding vertices.
- **STEP 2**: Estimate the area of each vertex, the wire width (i.e. weight) of each edge, and the schedule length of the hypergraph (see below).
- **STEP 3**: Apply a two-way hypergraph partitioning algorithm, using a modified $OBJFCT$ (see below). If area constraints can not be met, partition the subgraph of the largest vertex. Repeat until constraints are met. Multi-way partitions are achieved by repeated two-way partitioning. The vertices of a hyperedge must always be in the same partition, since they will share the same hardware.
- **STEP 4**: Synthesize structure for each partition. Each partition's structure corresponds to a chip.

Bottom-level CDFG nodes represent combinational logic blocks, so the area of the corresponding vertex is estimated as a function of the number of literals. The areas of relevant bottom-level vertices

are summed to obtain the area of a higher-level (i.e. *complex*) vertex. The exception is for vertices incident to a hyperedge; in this case, only one vertex per hyperedge contributes to the area, since all of these vertices will share the same hardware. The area of a partition is estimated in a similar manner. Edge weights are estimated as the number of wires needed for control (to indicate *done* and *reset*) plus the number of wires needed for data.

Vertices that represent combinational blocks are assigned a delay equal to the estimated number of clock cycles required for signals to propagate through the block. A complex vertex's delay is then the longest delay through all paths of its subgraph, plus one clock cycle for every edge on the path that crosses between partitions, representing interchip delay. The schedule length T is then the delay of the CDFG's longest path.

The partitioning is subject to:

- $area(V_i) \leq maxarea$ and $cutsize(V_i) \leq maxcutsize$ for all i, and $T < maxschedule$ (hard constraints)
- $OBJFCT = k_1 avgcut + k_2(T - maxschedule)$

where $avgcut$ is the average cutsize of all V_i, and k_1, k_2 are constants. Given a partitioning that meets the hard constraints, the above $OBJFCT$ attempts to minimize the average number of pins per chip and the overall schedule length. Variations of this $OBJFCT$ have also been proposed.

During partitioning, objects are tentatively moved among partitions. It is inefficient to recompute the values of all partition areas and cutsizes as well as the schedule length for each move. Incremental modifications to those values will save computation. Doing so for area is simple: simply subtract the areas of the moved objects from the source partition, and add it to the destination partition. Cutsize is also straightforward. Schedule length, however, can not be incrementally modified. Thus VULCAN uses an approximation of the change, and only recomputes the actual schedule length when an actual move (as opposed to a tentative one) is made.

3.6 CHOP

- *System context:* To synthesize a DFG and memories into structure consisting of one control unit and one or more datapaths.
- *Subproblem and goal: Packaging constraints* – partition the DFG and memories such that the structure synthesized for each partition meets chip packaging and schedule length constraints.
- *Approach:* Permit rapid estimation for a given DFG partitioning. Feasible implementations for each partition are estimated, and then a feasible overall implementation is selected. Area, pin, and delay estimates are derived from this.
- *Abstraction level:* DFG and memories
- *Mapping:* Graph model, mapping DFG nodes to vertices, DFG edges to edges.
- *Algorithm:* None – partitioning is up to the designer.
- *Notes:* CHOP assumes behavior is described as a DFG, as opposed to a CDFG, which restricts its usefulness to a small subset of applications (acyclic data-operations only). CHOP actually performs estimation (not partitioning), most of the details of which are beyond the scope of this survey.
- *References:* [20, 21]

The inputs to CHOP are a DFG and memories, area, pin, and schedule length constraints, a partitioning, and technology information, among other things. The overall algorithm is as follows:

- **STEP 1**: Create a set of feasible implementations for each partition. Various possible allocation and pipelining choices are considered, among other things. Estimate the area, cutsize, and schedule length of each feasible implementation.
- **STEP 2**: Choose a feasible implementation for each partition, considering area constraints for each partition and the overall schedule constraint.
- **STEP 3**: Permit the designer to repartition and return to STEP 1. Conversely, apply a synthesis tool to generate structure for each partition, passing the feasibility implementation information to the tool.

3.7 SpecPart – Specification Partitioning

- *System context:* To synthesize a a set of concurrent and sequential behaviors into structure consisting of one or more control units and datapaths.
- *Subproblem and goal: Packaging constraints* – partition the behavior such that the structure synthesized for each partition meets chip-packaging constraints, while also meeting an overall system performance constraint, and retaining the ability to modify the behavior after partitioning.
- *Approach:* Partition the specification itself, as part of a partitioning/communication-tradeoffs design iteration loop. Entire specification portions (behaviors and storage elements) are treated as the objects considered for grouping. Each object has an estimated area determined by treating each as a CU/DP; a system clock is estimated and scheduling and allocation are performed on each object. Each object communicates with others through data and control ports. Objects are partitioned into chips. The area of each chip and the pins required for interchip communication are estimated. Expected execution time of each object is estimated, based on both computation and communication times (including off-chip delays). Various partitionings can be rapidly evaluated in this manner.
- *Abstraction level:* Specification objects (language-imposed behavioral groupings and storage elements)
- *Mapping:* Hypergraph model, where specification objects (procedures, substates, processes, storage) are mapped to nodes. Estimated control and data lines between objects are represented as hyperedges. Extension: special directed edges are added to represent estimated on-chip/off-chip communication times.
- *Algorithm:* Any hypergraph partitioning. Currently clustering, group migration, and manual. A modified $OBJFCT$ is used for the extended hypergraph.

The motivation for developing SpecPart is the elevation of behavioral partitioning from the operation level to the algorithmic level (see Section 1), where a behavioral specification is viewed as a set of behaviors, such as processes, procedures, substates, and other code groupings imposed by the language, and a set of storage elements, including registers, memories, stacks, and queues. These behaviors and storage elements are then grouped into chips. This approach is in contrast to converting a description to a CDFG, and then grouping the data and control nodes (i.e. operation-level partitioning).

A focus is on determining how to obtain estimates of the area and pins of a group. Several behaviors that belong to a group may be sequential to one another, meaning that their structural implementation may share a single CU/DP. The effect of this sharing on structure area can be considered by applying to each group an area estimator which assumes the implementation will use a minimal number of CU/DPs. This estimation method is only feasible when considering a small number of possible groupings (e.g. when using a cluster tree). To consider more possible groupings, which is necessary to explore area/pin/performance tradeoffs thoroughly, a faster estimation method is needed. Each object

is treated as an individual CU/DP, and hence has its area estimated only once, before partitioning. A group's area is then the sum of its members' areas.

A second focus is on refining the original behavioral specification with the chip structure determined by partitioning, as opposed to combining the addition of chip structure with structural synthesis of the behavior. This is seen as necessary since the specification may be changed by the designer after partitioning, as is commonly done in practice.

A third focus is on incorporating performance constraints for multiple behaviors into partitioning. This is done by extending the hypergraph model, by using an abstraction in which communication is modeled as protocols, and by modifying the objective function.

SpecPart takes as input a hierarchical behavioral specification (in the SpecCharts language [22]), an $OBJFCT$ based on area, pins, performance and the number of chips, and a set of soft constraints on these metrics, among other things. It also has access to area, wire/pin, and execution-time estimators. The overall algorithm is described below (for the moment we shall ignore system performance constraints).

- **STEP 1**: Select the specification objects to be considered for partitioning (see below). Convert each object into a new concurrent process that communicates with other processes through connected ports (see below).
- **STEP 2**: Map each object to a vertex. Estimate the area of each object assuming a single CU/DP implementation, making this the vertex area. For each set of connected ports, add a hyperedge between the corresponding vertices, with a weight equal to the estimated wire-width of the ports.
- **STEP 3**: Partition the hypergraph, using hypergraph algorithms or manually, and provide evaluation metrics. Go to step 1, step 3, or step 4, based on the designer's choice.
- **STEP 4**: Create a refined specification in the original language, containing chip modules and their interconnection, and the behavioral specification of each chip.

Treating sequential behaviors as separate CU/DPs may lead to slightly inaccurate area estimates. Thus, a goal of object selection is to choose the minimal number of behaviors (i.e. those high in the specification hierarchy which encompass many sub-behaviors and storage elements) such that objects are still fine-grained enough to enable a satisfactory partitioning.

Each object is moved to a concurrent process which communicates with other processes through ports only. A behavior's original location will only contain actions which activate/deactivate the new process. All access to memories is through address and data ports; for registers, access is simply through data ports. The area estimator provides an area for each process. All ports are of one-dimensional type, since memories are accessed with address and data ports; thus the wire-width of each connection of ports is estimated simply as the number of bits required.

Each process is mapped to a hypergraph vertex, and each port interconnection to a hyperedge. Any hypergraph partitioning algorithm can then be applied. An existing example is a clustering algorithm using SUM as the $newweight$ function. Group migration has also been incorporated, using the following straightforward $OBJFCT$:

$$OBJFCT = k_1 \sum_i \left(100 \times \frac{excessarea(V_i)}{maxarea}\right)^2 + k_2 \sum_i \left(100 \times \frac{excesscutsize(V_i)}{maxcutsize}\right)^2 + k_3 \left(100 \times \frac{excesschips}{maxchips}\right)^2 \quad (9)$$

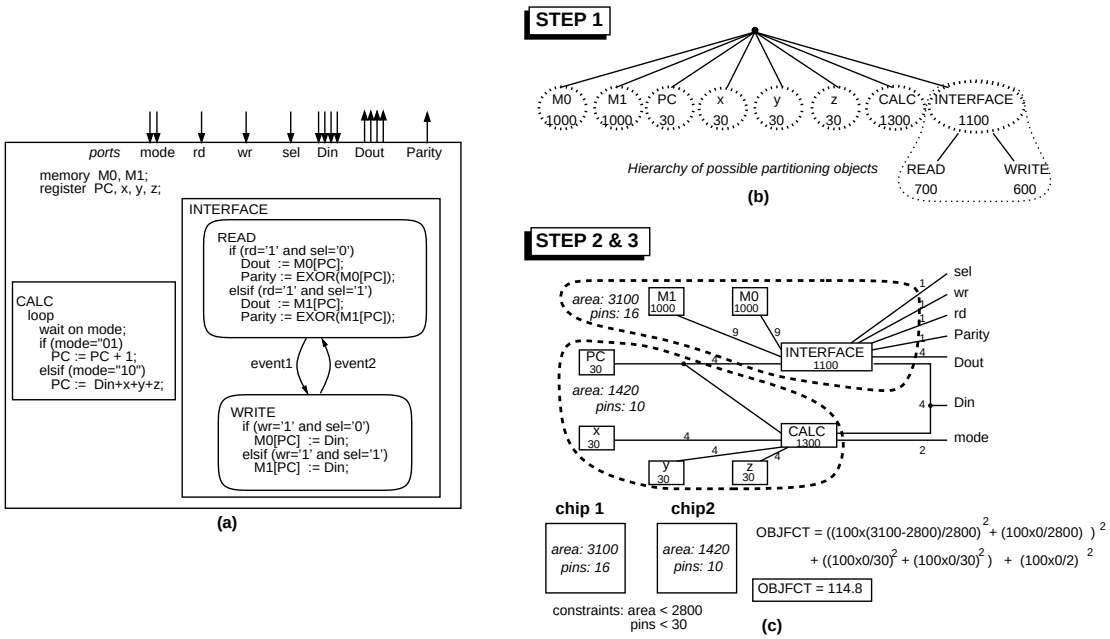


Figure 15: Specification partitioning

This function attempts to minimize constraint violations. If a metric does not exceed a constraint, then its excess value is zero, rather than being a negative value. Thus, any partitionings that meet all constraints are considered equal. Multiplying each term by 100 makes the term a percentage by which the actual value is greater than the constraint. Squaring of terms is done to favor balanced over unbalanced excesses. Other objective functions can also be used.

When a satisfactory partitioning is found, a new specification is generated. At its top-level will be processes representing the chips, with connections between these for communication. The goal of refinement is to minimize the change from the original specification; hence a behavior or storage element is only converted to a process if the partitioning requires it. For example, a memory that was originally declared as a global memory need only be converted to a process if it is accessed off-chip; otherwise it is a global memory in the chip on which it exists. Similarly, a behavior in the hierarchy is only converted to a process if its ancestor behavioral object is on a separate chip; otherwise it will appear in the ancestor's hierarchy just as it did in the original specification.

In Figure 15(a), a behavior with two processes, each represented as a box, is shown (in no particular language). The second process consists of two sub-behaviors that could be processes, procedures, or substates. Two memories and four registers are also declared. In the example, only the objects at the top of the specification hierarchy are selected; thus, INTERFACE is considered in its entirety. Figure 15(c) shows the hypergraph model created for this example, including estimates of the number of wires between objects. Note that nine wires are estimated for communication between INTERFACE and M1. Four are for address, four for data, and one to indicate writing or reading. A partitioning is also shown. Figure 15(d) shows the partitioning evaluation. The only constraint exceeded is the area

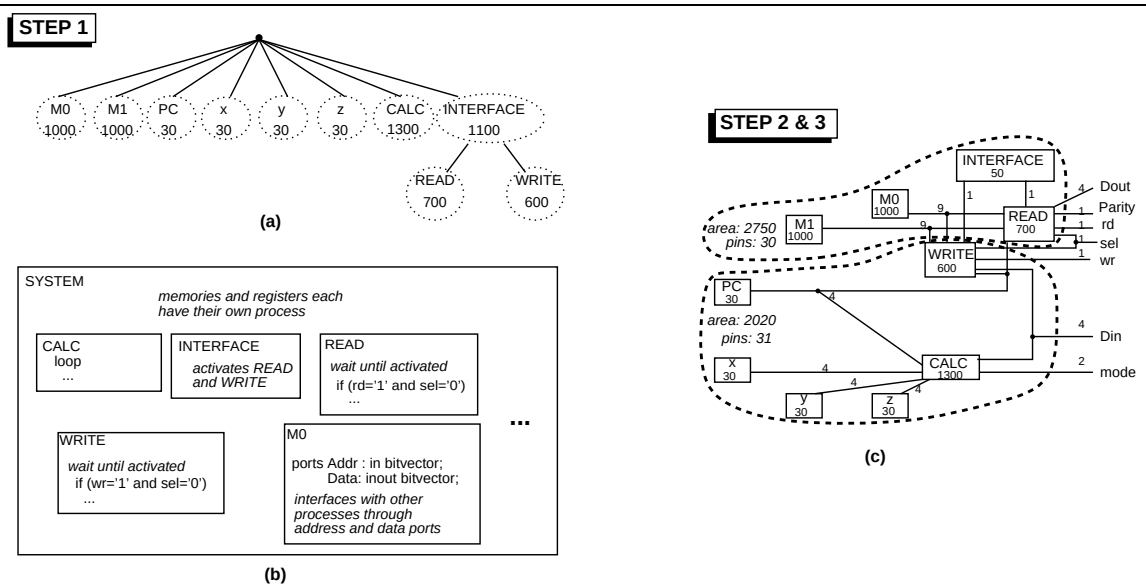


Figure 16: Decomposing a behavior for finer-granularity partitioning

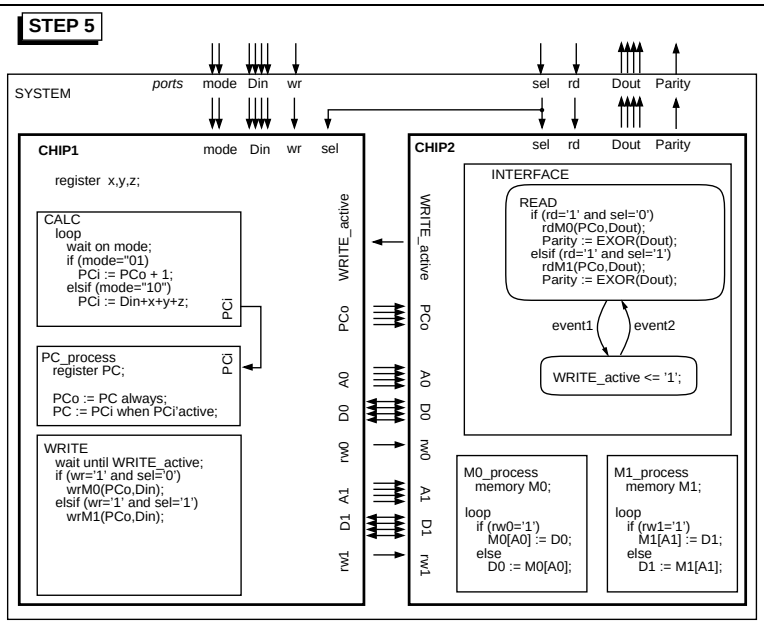


Figure 17: A refined specification resulting from partitioning

constraint of the first chip, which leads to a positive value for $OBJFCT$.

In Figure 16(a), an alternative object selection is shown in which READ and WRITE are also selected. In Figure 16(b), the results of converting to processes are shown. Note that READ and WRITE are concurrent processes, and that INTERFACE consists only of a simple behavior that controls those two processes. Also note that the memories have their own processes (as was also true for the previous example). The partitioned hypergraph is shown in Figure 16(c). The second chip exceeds the pin constraint by only one. Area constraints are now met for both chips. Respecification can also be used reduce chip pin-counts by creating behaviors which use a different set of ports.

The refined specification in Figure 17 is derived from the partitioning of Figure 16(c). At the top level are two concurrent behaviors which represent two chips. Note that x, y, z are accessed only on CHIP1, so they are defined as registers global to that chip. On the other hand, PC is accessed by both chips, so is converted to a process which communicates through ports; likewise for M0 and M1. The WRITE process communicates with M0 through address, data, and control ports. Assume that the procedures $wrM0$ and $wrM1$ exist, and that they implement one half of the communication protocol by placing the address and data parameters on the appropriate buses, and set the rw line to the value for writing.

Note that since READ and INTERFACE were grouped to the same chip, READ appears as a sub-behavior of INTERFACE as it was in Figure 15(a). On the other hand, WRITE is replaced in INTERFACE by a behavior which merely activates the WRITE process through a port to the other chip.

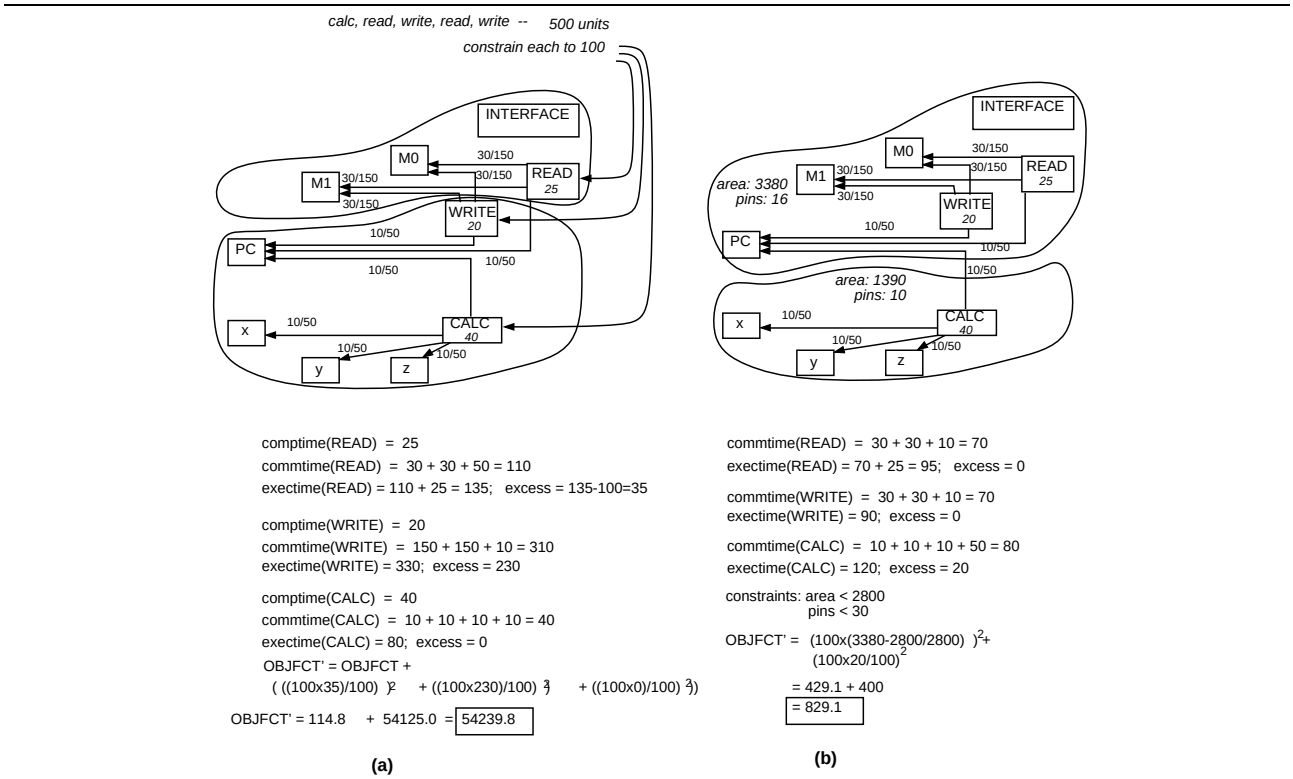


Figure 18: Incorporating performance constraints in SpecPart

The above is extended to consider performance. Each behavior in the specification may optionally have an expected execution-time constraint. This is the average time it takes to execute the behavior from start to finish. This constraint is associated with the corresponding hypergraph vertex as

$maxexectime(v_i)$. A behavior's execution time is viewed as the sum of its computation time and its communication time. The former is determinable by operator delays and branch probabilities. The latter requires that communication be modeled as protocols. A behavior can initiate a protocol (such as a memory read protocol), which will take a specific amount of time to complete. These times will differ for on-chip and off-chip accesses. A special directed *communication edge* is added between a behavior's vertex and all vertices with which the behavior initiates a communication protocol. The edge has two weights corresponding to an off-chip and an on-chip communication delay ($offchipdelay(e_{j,k})$ and $onchipdelay(e_{j,k})$), which is the protocol time multiplied by the estimated number of uses of this protocol in a single pass of the behavior. The total communication time of a vertex is the sum of the exiting communication edge weights, using the appropriate off-chip/on-chip edge weight for a given partitioning. The following is added to $OBJFCT$:

$$OBJFCT = ... + k_4 \sum_j \left(100 \times \frac{excessexectime(v_j)}{maxexectime(v_j)} \right)^2 \quad (10)$$

where:

$excessexectime(v_j)$	$= execime(v_j) - maxexectime(v_j)$, if $execime(v_j) > maxexectime(v_j)$, 0 otherwise,
$execime(v_j)$	$= commtime(v_j) + comptime(v_j)$,
$comptime(v_j)$	the expected time to execute a single pass of the behavior associated with v_j assuming communication times are 0,
$commtime(v_j)$	$= \sum_k commdelay(e_{j,k})$
$commdelay(e_{j,k})$	$= offchipdelay(e_{j,k})$ if v_j and v_k are not in the same partition V_i, $onchipdelay(e_{j,k})$ otherwise (recall that these delays incorporate expected frequency of use),
k_4	user-defined constant

Note that this term involves v_j, not V_i. Specifically, excess execution time is determined per vertex, not per partition.

As an example, consider the example of Figure 15. Suppose the designer knows that a typical sequence consists of executing CALC once, followed by two READs and two WRITEs. This average time to perform this sequence is constrained to 500 time units. One way to achieve this is to constrain each of the three behaviors to 100 time units. In Figure 16, partitioning was performed without incorporation of performance, and the minimal $OBJFCT$ value was found. In Figure 18(a), a graph model is shown with the communication edges and their estimated on-chip/off-chip delays, along with the estimated *comptime* for the three time-constrained vertices (hyperedges and vertex sizes are omitted for clarity). Extending the $OBJFCT$ to consider performance gives a very high value, indicating poor system performance due to excessive interchip communication time. A repartitioning finds the minimal value for the extended $OBJFCT$.

3.8 SPARTA and SLIP

Because the term "system partitioning" can refer to either the level of the input description or the level of the modules on which to implement hardware, the SPARTA (A System Partitioning Aid [23]) and SLIP (System Level Interactive Partitioning [24]) systems are commonly confused with behavioral-partitioning systems. SPARTA is used to evaluate *structural netlist* partitionings among chips or other

packages such that packaging constraints are met. It consists of spreadsheet-like software that is used to evaluate various chip metrics such as area, pins, and power. Traditional spreadsheets are extended to permit metric calculations using programs, not just arithmetic expressions. SLIP concentrates on providing a data model for a hierarchical structural partitioning and for packaging technologies.

4 Summary of Three Important Aspects

This report has demonstrated the various input levels assumed by behavioral-partitioning approaches. Inputs included a DFG plus memories, DFG operations plus logic, a single sequential CDFG procedure, a sequential CDFG with multiple procedures, a sequential hierarchical CDFG, and a behaviorally-hierarchical original specification.

We have also shown that the pieces of the input that are actually treated as partitioning objects (i.e. are grouped) varies between systems. The assumed target architecture affects the range of possible pieces. We demonstrated the manners in which these pieces are mapped to a graph model and then partitioned.

The use of the partitioning results also varies among approaches. The results can be used to obtain estimations, to divide the input to logic synthesis into several smaller inputs, to provide structural information to a high-level synthesis tool, or to add structure to the original specification which can then be further modified by other tools or designers.

Refer to Figure 9 for a summary of these three aspects for each approach considered in this report.

5 Conclusions

The various assumptions made in each behavioral-partitioning approach greatly affect an approach's domain of applicability, with the assumed input being the most important. An approach which assumes a behavior is described as a DFG cannot be used for behaviors which contain much control, as many behaviors do. An approach which assumes that behavior consists of registers and combinational logic can only be used *after* high-level synthesis. An approach which assumes that behavior is a sequential CDFG cannot be used for behaviors which contain concurrency. The input level also determines the type of performance constraints that can be specified.

As yet, no approach which assumes a CU/DP target architecture has shown the ability to create multiple concurrent controllers to reduce interchip delay. Instead, a single controller exists on one chip which controls datapaths on the same or separate chips. Conversely, approaches which assume a combined control/datapath target architecture can create multiple controllers. Hence the desired target architecture also plays an important role in the applicability of a partitioning approach.

In terms of results, intrachip behavioral partitioning for tractability has shown beneficial results by reducing computation time and/or improving the quality of structure and layout by reducing functional units, busing or overall routing area.

More research is needed in the area of interchip behavioral partitioning, whose usefulness has yet to be conclusively demonstrated. Real examples are needed, especially those with much control (i.e. not just DFGs) and concurrency. Comparison with structural approaches is also necessary. Other areas of future work include optimizing performance through the use of minimal CU/DPs and executing behaviors on standard processor chips.

6 References

[1] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout.* England: John Wiley and Sons, 1990.

[2] M. McFarland, A. Parker, and R. Camposano, "Tutorial on High-Level Synthesis," in *Proc. of the 25th Design Automation Conference*, 1988.

[3] R. Walker and R. Camposano, *A Survey of High-Level Synthesis Systems.* Massachusetts: Kluwer Academic Publishers, 1991.

[4] D.D. Gajski, et. al., *High-Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, 1991.

[5] R. Camposano and R. Brayton, "Partitioning Before Logic Synthesis," in *Proc. of the International Conference on Computer-Aided Design*, 1987.

[6] R. Camposano and J. van Eijndhoven, "Partitioning a Design in Structural Synthesis," in *Proc. of the International Conference on Computer Design*, 1987.

[7] D. Gajski, *Silicon Compilation.* Massachusetts: Addison-Wesley, 1988.

[8] M. McFarland and T. Kowalski, "Incorporating Bottom-Up Design into Hardware Synthesis," *IEEE Transactions on Computer-Aided Design*, September 1990.

[9] M. McFarland, "Computer-Aided Partitioning of Behavioral Hardware Descriptions," in *Proc. of the 20th Design Automation Conference*, 1983.

[10] M. McFarland, "Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions," in *Proc. of the 23rd Design Automation Conference*, 1986.

[11] J. Rajan and D. Thomas, "Synthesis by Delayed Binding of Decisions," in *Proc. of the 22nd Design Automation Conference*, 1985.

[12] E. Lagnese, *Architectural Partitioning for System Level Design of Integrated Circuits.* PhD thesis, Carnegie Mellon Unversity., March 1989.

[13] E. Lagnese and D. Thomas, "Architectural Partitioning for System Level Synthesis of Integrated Circuits," *IEEE Transactions on Computer-Aided Design*, July 1991.

[14] E. Lagnese and D. Thomas, "Architectural Partitioning for System Level Design," in *Proc. of the 26th Design Automation Conference*, 1989.

[15] D.E. Thomas, et. al., "The System Architect's Workbench," in *Proc. of the 25th Design Automation Conference*, 1988.

[16] R. Walker, *Design Representation and Behavioral Transformation for Algorithmic Level Integrated Circuit Design.* PhD thesis, Carnegie Mellon Unversity., April 1988.

[17] R. Walker and D. Thomas, "Behavioral Transformation for Algorithmic Level IC Design," *IEEE Transactions on Computer-Aided Design*, October 1989.

[18] R. Gupta and G. Micheli, "Partitioning of Functional Models of Synchronous Digital Systems," in *Proc. of the International Conference on Computer-Aided Design*, 1990.

[19] G. Micheli and D. Ku, "HERCULES - A System for High-Level Synthesis," in *Proc. of the 25th Design Automation Conference*, 1988.

[20] K. Kucukcakar and A. Parker, "CHOP: A Constraint-Driven System-Level Partitioner," in *Proc. of the 28th Design Automation Conference*, 1991.

[21] K. Kucukcakar and A. Parker, "CHOP: A Constraint-Driven System-Level Partitioner." University of Southern California, TR CEng 90-26, 1990.

[22] S. Narayan, F. Vahid, and D. Gajski, "System Specification and Synthesis with the SpecCharts Language," in *Proc. of the International Conference on Computer-Aided Design*, 1991.

[23] M. Resnick, "SPARTA: A System Partitioning Aid," *IEEE Transactions on Computer-Aided Design*, October 1986.

[24] M. Beardslee et. al., "SLIP: A Software Environment for System Level Interactive Partitioning," in *Proc. of the International Conference on Computer-Aided Design*, 1989.

A Appendix

A.1 Partitioning for Tractability: Allocation Example

The following table shows the number of possible allocations for the DFG of Figure 7(a), assuming any number of adders (A), subtractors (S), and adder/subtractors (AS) are available as functional units. Since there are only four operators that add or subtract, we know that there can be no more than four adder/subtractors allocated. Since there is only one subtraction operation, there can be no more than one subtractor allocated. Likewise, there can be no more than three adders used. All possible combinations of these selections are listed in the table.

Certain of these combinations are not feasible, meaning that they lack enough functionality (e.g. only one adder with no other functional units). Also, certain combinations provide excess functionality, e.g. three adders and four adder/subtractors. These selections are considered invalid, and marked with a '-' in the fourth column. For the valid component selections, the number of possible bindings of DFG operations to functional units is shown, computed by a program which exhaustively made all possible bindings.

```
#A  #S  #AS  num possible allocs
---------------------------------
0   0   0     -
0   0   1     1
0   0   2     16
0   0   3     81
0   0   4     256
0   1   0     -
0   1   1     2
0   1   2     24
0   1   3     108
0   1   4     -
1   0   0     -
1   0   1     8
1   0   2     54
1   0   3     192
1   0   4     -
1   1   0     1
1   1   1     16
1   1   2     81
1   1   3     -
1   1   4     -
2   0   0     -
2   0   1     27
2   0   2     128
2   0   3     -
2   0   4     -
2   1   0     8
2   1   1     54
2   1   2     -
2   1   3     -
2   1   4     -
3   0   0     -
3   0   1     64
3   0   2     -
3   0   3     -
3   0   4     -
3   1   0     27
3   1   1     -
3   1   2     -
3   1   3     -
3   1   4     -
---------------------------------
Totals: possible component selections: 19, possible allocs: 1148
```

The following tables show the possible allocations for the partitioned DFG of Figure 7(b). Note the substantial reduction in possibilities.

```
(Cluster 1)
#A  #S  #AS  num possible allocs
---------------------------------
0   0   0     -
0   0   1     1
0   0   2     4
0   1   0     -
0   1   1     2
0   1   2     -
1   0   0     -
1   0   1     2
1   0   2     -
1   1   0     1
1   1   1     -
1   1   2     -
---------------------------------
Totals: possible component selections: 5, possible allocs: 10


(Cluster 2)
#A  #S  #AS  num possible allocs
---------------------------------
0   0   0     -
0   0   1     1
0   0   2     4
1   0   0     1
1   0   1     4
1   0   2     -
2   0   0     4
2   0   1     -
2   0   2     -
---------------------------------
Totals: possible component selections: 5, possible allocs: 14
```