VULCAN - A System for High-Level Partitioning of Synchronous Digital Circuits

Rajesh K. Gupta *and* Giovanni De Micheli

Technical Report: CSL-TR-91-471
April 1991

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University, Stanford, CA 94305-4055.

Abstract

Existing high-level synthesis techniques try to achieve a single-chip hardware structure from a given behavioral, i.e., *high-level* description. However, area limitations on a single chip may necessitate hardware sharing during synthesis process, which affects overall performance adversely. Therefore, system-level partitioning techniques are required in the synthesis of large digital systems in order to meet area and performance constraints. While considerable effort has been spent in the past in obtaining partitions at the structural description level, use of partitioning techniques at the behavioral level is emerging only recently. There are three major advantages of using partitioning techniques at functional abstraction level. First, scheduling techniques can be applied concurrently to partitioning. Therefore, partitioning under timing constraints, and in particular under latency constraints, can be performed. Second, the functional model captures large hardware systems with fewer objects (than at the logic netlist abstraction level), making the partitioning algorithm more efficient. Third, hardware sharing trade-offs can be considered.

The problem of hardware partitioning is formulated as a hypergraph partitioning problem and constrained optimization algorithms are developed to achieve optimum partitions. *Vulcan* provides an interactive work-bench to perform partitioning at the functional abstraction level, where the digital hardware being designed is represented by a hierarchical sequencing model capturing the operations to be performed and their dependencies. Such a functional model is a common abstraction in high-level synthesis and, for example, it can be obtained by compiling a hardware model in the *HardwareC* language. *Vulcan* is integrated with the Olympus Synthesis System developed here at Stanford University. *Vulcan* supports externally imposed hardware sharing, area and latency constraints. Like the mythical smith it is named after, *Vulcan* provides an interactive work-bench to apply partitioning transformations, create, simulate and synthesize multiple chip systems.

Key Words and Phrases: High-level Synthesis, Partitioning, High-level Partitioning, Multiple Chip Modules (MCMs)

Contents

1 Introduction **3**

1.1 Motivations for High-Level Partitioning 4

2 Previous Work **4**

3 The Hardware model **5**

3.1 Hardware Area and Timing Model 6

4 The Partitioning Problem Definition and Assumptions **8**

5 Partitioning Algorithms **9**

5.1 **Partitioning with Shared Resources** 14

5.2 Partitioning of Hierarchical Hypergraphs 16

6 The Vulcan Partitioning System **18**

6.1 Models and Partition Information Maintained in Vulcan 20

6.2 Partitioning of Model Graphs 21

6.3 Creation Of Hardware Models From Partitioned Model Graph 21

6.4 Assembly of Interacting Hardware Models into A Single Structure 22

7 Examples **22**

7.1 Effect of Resource Sharing: Elliptic Filter 24

8 Summary **27**

9 Acknowledgements **28**

10 Appendix A: Vulcan Commands Listing **29**

List of Figures

1	*A SIF Hardware Model*	7
2	*Tagging and Latency Computation of SIF Graphs*	11
3	*Computation of Communication Cost in Hypergraphs*	14
4	*Partitioning of Complex Condition-less Vertices*	16
5	*Partitioning of Complex Conditional Vertices: Loops*	17
6	*Partitioning of Complex Conditional Vertices: Conditions*	18
7	*Interaction of Vulcan with Other Programs in The Olympus Synthesis System*	19
8	*Data Organization in Vulcan*	21
9	*Partitioned Elliptic Filter Sequencing Graph without Resource Sharing*	26

VULCAN - A System for High-Level Partitioning of Synchronous Digital Circuits

Rajesh K. Gupta and Giovanni De Micheli
Center for Integrated Systems
Stanford University
Stanford, CA 94305.

1 Introduction

Recent advances in high-level synthesis techniques have seen emergence of synthesis systems that can synthesize chips from a given algorithmic description of chip behavior in a hardware description language [1] [2] [3] [4]. The emphasis of all these synthesis systems so far has been in synthesizing single-chip systems. However, as synthesis techniques mature, the need for synthesis of multiple-chip system designs from a single high-level system behavior description is expected to grow for several reasons. One, the limited number of available gate counts on most programmable logic devices makes it impossible to effectively synthesize any realistic chips of reasonable complexity beyond 20K gates. Further, the computational performance of synthesis system itself is severely affected in case of large hardware models. Therefore, a partitioning of hardware functions into a chip-set is crucial in achieving an efficient implementation. While hardware partitioning is dictated by the chip area limitations, it also affects the performance of the overall system. Although considerable amount of work has been done in obtaining partitions based on physical considerations, use of partitioning at behavioral level has been rather limited. This report describes the problem of partitioning of synchronous digital systems at the functional model description level within the framework of *Olympus Synthesis System* developed at Stanford University. First we concentrate on defining the problem of hardware partitioning and various algorithmic approaches to obtain the partitions. We then describe the features of a practical partitioning system, *Vulcan*[5]. *Vulcan* provides a workbench to investigate partitioning techniques of hardware models that allow efficient implementation of hardware as multiple chip systems. Input to *Vulcan* is a model of the hardware to be partitioned along with the area and performance constraints on the final implementation. The hardware model consists of sequencing graph abstraction, referred to as Sequencing Intermediate Form or SIF, which is further described in Section 3. *Vulcan* supports resource sharing constraints related to limitations on hardware modules that can be used by the final implementation. *Vulcan's* interactive interface, modeled after the generic Unix shell interface, provides commands to apply various partitioning techniques as well as routines to assemble partitioned hardware models into a single interconnecting block structure for simulation purposes.

1.1 Motivations for High-Level Partitioning

As mentioned earlier the goal of high-level partitioning is to obtain a set of interacting behavioral models from one behavioral description, that satisfy *both* chip area constraints as well as an overall latency timing constraint. Such a partitioning of hardware models can then be used in synthesizing multi-chip systems or even in speeding up hardware-assisted simulation of large behavioral models. Among the advantages of using partitioning techniques at the functional abstraction level are:

- Scheduling techniques can be applied concurrently to partitioning. Therefore, partitioning under timing constraints can be performed. In particular, overall latency of a partitioned structure can be readily evaluated including the inter-chip communication delays. In this way, area and performance (in terms of hardware latency) trade-offs can be exploited effectively.
- The functional model captures large hardware systems with fewer objects than at the logic netlist abstraction level, making partitioning algorithms more efficient.
- Hardware sharing trade-offs can be considered.

The major disadvantage is the possible inaccuracy of the area and delay models, that are estimated directly from the functional models and that do not include precise interconnect delays.

The following sections in this report are organized as follows. Section 2 provides a brief review of existing literature in high-level partitioning. Section 3 introduces the hardware model abstraction and the hardware area and timing model used in *Vulcan*. Section 4 formally defines the hardware partitioning problem and underlying assumptions. Section 5 discusses various partitioning algorithms available in *Vulcan*. Section 6 introduces the main features of *Vulcan*. Section 7 presents some examples of partitioning.

2 Previous Work

Much of the partitioning related work in the past has been concentrated at providing partitions at the structural level of representation where the process of resource allocation and scheduling have long since been performed [33] [34]. At the structural level there exists a wealth of information regarding physical aspects of the hardware being partitioned. However, it is difficult to assess and, more importantly, control the effects of partitioning on the overall performance of the hardware at this level. High-level partitioning techniques were pioneered by Dirkes and Thomas [6]. They considered a multistage clustering algorithm, that perfected the clustering techniques presented in [7] and [8]. The algorithm operates on the Value Trace [9] functional model, that is similar to the model used in Olympus Synthesis System. However, Dirkes' approach was not formulated as a constrained optimization problem and scheduling and latency computation was done separately in a later step [2]. Kucukcakar and Parker [10] studied the problem of system-level partitioning under area and performance constraints. In their approach the input to the system is potential partitions along with area and performance constraints on resulting implementations. The partitioner heuristically searches through various possible implementations of *given* partitions to

find feasible implementations. However, the partitions themselves are manually created by the designer. Camposano and Brayton [11] studied high-level partitioning techniques by means of clustering, with the goal of improving the efficiency of logic synthesis. The major difference of our method over previous ones is that in our approach hardware partitioning is carried out under area and performance constraints on the partitioned hardware implementation. In contrast to [10], *Vulcan* creates and as well as evaluates partitioned implementations.

3 The Hardware model

The hardware behavior is modeled as a set of operations and a partial order among the operations. This model is a common abstraction used in high-level synthesis [1] [9] [12] [13]. The hardware abstraction model used in the Olympus Synthesis System can be derived by compiling hardware descriptions in the *HardwareC* language. This abstraction model can also be derived, *mutatis mutandis*, from most other procedural Hardware Description Languages, that support structured programming, i.e. nested model calls, branching and iterative constructs. The sequencing abstraction model, also referred to as *Sequencing Intermediate Form* – SIF, can be seen as a *hierarchical hypergraph* [14]. At each level of the hierarchy, the hypergraph is characterized by a set of vertices, a set of directed edges (ordered vertex pairs) and hyperedges (unordered vertex subsets). The vertices represent operations to be performed, the directed edges represent their dependencies and the hyperedges represent resource (hardware) sharing between operations. We assume that an hyperedge can contain only one vertex as a limiting case (dedicated resource), that the hyperedges are disjoint subsets and their union covers the vertex set, i. e., they form a partition of the vertex set. We call sequencing graph the sets of vertices and edges (i.e., excluding the hyperedges). The sequencing graph is then *acyclic* because only structured programming constructs are assumed (e.g., no unrestricted *goto* statements) and the corresponding complex vertices break the graph through the use of hierarchy. The sequencing graph is also a polar graph, with unique source and sink vertices, that are effectively no-op vertices used to denote the first and last operation to be performed. The vertices are classified into *simple* and *complex* vertices. Example of simple vertices are those representing blocks of Boolean expressions, input-output operations and no-ops. Complex vertices allow groups of operations to be performed, and include model calls, conditionals, and loops. These complex vertices induce a hierarchical relationship among the hypergraphs. A call vertex invokes the hypergraph corresponding to the called model. A conditional vertex selects among a number of branches, each of which is modeled by a hypergraph. A loop vertex iterates its body until the exit condition is satisfied; the body of the loop is also a hypergraph.

The semantic interpretation of the sequencing graph model is as follows. A vertex is *executed* by performing the task described by the vertex. For example, to execute a *computation* vertex, the task to be performed is the evaluation of the corresponding expression; to execute a *conditional* vertex, the operations within the selected branch are executed. In the case of a *call* vertex, the control flow is temporarily transferred to the called graph. When the called graph completes execution, the control flow returns to the calling vertex. The execution of a sequencing graph is the execution of its vertices according to the dependencies of the graph. A vertex begins execution when all its predecessors have

completed execution. Since a vertex may have multiple fan-ins and fan-outs, the SIF hardware model supports *multiple threads of concurrent execution flow.*

We name *resource* a generic hardware model that can be referred to by a *call* vertex in the sequencing abstraction. In particular, a resource can be a "standard" functional unit, such as an adder or a multiplier. However, the notion of resource is not limited to these standard functional models. A resource can be shared, when the same hardware block implements the operation corresponding to two or more calls to its model. Up to a certain extent resource sharing is beneficial in reducing the hardware needed for implementing a given behavioral model. (However, as the degree of resource sharing increases so does the amount of control circuitry required to facilitate resource sharing.) For the ease of partitioning problem formulation (as a hypergraph partitioning as we shall see later), we assume that resources can be shared only when they correspond to vertices at the same level of hierarchy of the SIF model. We denote resource sharing by grouping the corresponding vertices together by means of hyperedges (i.e., edges incident to more than one vertex). It is important to note that each hyperedge is incident to vertices of the same type and that each vertex can be incident to one hyperedge only. We also assume that operations corresponding to vertices incident to a hyperedge can not execute simultaneously, and therefore that appropriate serialization of the tasks corresponding to shared resources is modeled by directed edges. Figure 1 shows example of a hardware description and its corresponding sequencing graph model.

3.1 Hardware Area and Timing Model

Area and timing costs represent two major indicators of quality of final hardware designed. In general terms, the **area cost** reflects the size of the resulting structure, which consists of the sum of the area costs of various hardware resources (blocks) used. In addition, the total hardware cost also involves the cost of *control circuits* and *registers* needed to implement the control. Since the control circuitry is generated during the structural synthesis at a later stage, therefore at the behavioral or SIF level only estimates of resulting control cost can be made. Hence, in our approach the 'data-path' portion of a chip is accurately modeled in terms of gate count and delay by applying logic synthesis and technology mapping to various hardware resources used. Whereas only estimates can be used for the area and delay of the control portion and for the wiring delays. In the SIF model we assume that each vertex has an area-cost associated to it. The area-cost of the simple vertices can be computed as follows. An estimate of the area taken by Boolean expression blocks, can be derived by counting the total number of literals, as usually done by most logic synthesis techniques [15]. The area cost of registers and I/O structures is assumed to be known and no area-cost is associated to no-ops. The area-cost of a hyperedge is equal to the area cost of any vertex it is incident to. The area cost of the complex vertices is the area-cost of the corresponding called hypergraphs. Therefore, the area-cost of a SIF model can be computed bottom-up. Note that in case of no resource sharing, the total area is the sum of the area of all vertices. This is, of course, an upper bound on the resulting hardware area.[1] *Control cost* is estimated by using the control generation scheme suggested in [16]. Control cost associated with every vertex depends upon the number of anchors and the maximum offset of that vertex from its anchor set.

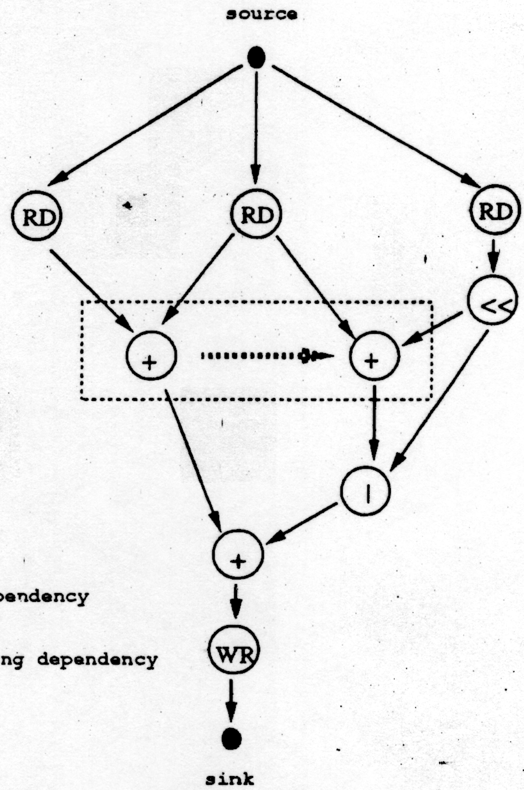
[1]Control cost estimation using anchor sets has not been implemented in the current version of Vulcan.

```
process example(ain, bin, cin, dout, rst)
    in port ain[SIZE], bin[SIZE], cin[SIZE];
    out port dout[SIZE];
    in port rst;
{
    boolean r1[SIZE], r2[SIZE], r3[SIZE];

    while(!rst) {

                r1 = read(ain);
                r2 = read(bin);
                r3 = r1 + r2;
                r1 = read(cin);
                r1 = r1 << 2;
                r2 = r1 + r2;
                r2 = r2 | r1;
                r1 = r2 + r3;
                write dout = r1;

    }
}
```



sequencing dependency

resource sharing dependency

hyper-edge

HARDWARE-C DESCRIPTION

SEQUENCING MODEL

Figure 1: *A SIF Hardware Model*

Similarly each vertex of the SIF model has an associated **delay-cost**, related to the execution delay of the corresponding operation. We assume that the delay-cost is specified in terms of an integer number of cycles, possibly equal to zero. This number may be unknown in the case of data-dependent iterations and synchronization primitives [17]. When at least one vertex has unknown delay-cost, the overall delay-cost is not known and the SIF model is said to have *unbounded* delay-cost. Otherwise, the timing-cost of a hypergraph is defined to be the *critical path* length of the corresponding sequencing graph. The critical path is the longest directed path from source to the sink, where the timing-cost of a complex vertex is the timing-cost of the corresponding hypergraph. Therefore the timing-cost of a hierarchical SIF model can be computed bottom up and it is called *latency*.

A behavioral hardware specification may contain timing constraints. In particular, timing constraints may be imposed on a given hardware model to ensure that the time interval between execution of two given operations is within a certain upper and/or lower bounds. Such constraints can be specified as maximum and minimum time-interval between start of two given operations in the corresponding sequencing graph, and they can be represented as additional directed edges. In particular, an upper bound on the overall latency can be specified as an upper bound timing constraints between the source and sink vertices of the hypergraph at the root of the hierarchy.

4 The Partitioning Problem Definition and Assumptions

Given the hardware model described in the previous section, the problem of partitioning of hardware models is formulated under following assumptions:

- Each block of the partition has an upper bound $\overline{A}$ on the area-cost
- Each block of the partition has an upper bound $\overline{C}$ on the pinout-cost where the pinout-cost is defined as the number of I/O pins excluding power and ground
- The overall latency λ has an upper bound $\overline{\lambda}$
- Synchronous single clock hardware implementation
- Synchronous inter-block communication where an integer delay-cost is associated to each inter-block data transfer. Without loss of generality, we assume it to be one clock cycle.
- Shared hardware resources can not be split among blocks of the partition. In presence of shared resources appropriate serialization of operations has been done prior to partitioning.

We denote the (hierarchical) hypergraph by $\mathcal{H}$ and we state the general partitioning problem as follows:

Problem 1: *Partition a hypergraph $\mathcal{H}$ into a minimal number n of hypergraphs $\mathcal{H}_i$, $i = 1, 2, \ldots, n$ such that the area-cost of each block $A_i \leq \overline{A}$, the pinout-cost of each block $C_i \leq \overline{C}$ and the overall latency $\lambda \leq \overline{\lambda}$.*

While it is possible to obtain optimum number of partitions using flow or clustering algorithms, it is usually difficult to constrain the sizes of the resulting partitions in such approaches. Therefore, we approach the multi-way partitioning problem by performing successive bipartitions. Let C (without any subscripts) be the *communication cost* (i.e. the number of wires) between the two blocks of the partition. That is, C represents the pinouts common between the two block. The bipartitioning problem can then be stated as follows:

Problem 2: *Partition a hypergraph $\mathcal{H}$ into two hypergraphs $\mathcal{H}_i$, $i = 1,2$ such that the area-cost of each block $A_i \leq \overline{A}$, the pinout-cost of each block $C_i \leq \overline{C}$, the overall latency $\lambda \leq \overline{\lambda}$ and the cost function $f = \alpha C + \beta(\lambda - \overline{\lambda})$ is minimal.*

In Problem 2, the parameters α and β represent desired tradeoff between communication cost and latency penalty due to partitioning. A solution to Problem 2 is also a solution to Problem 1 for $n = 2$. If no feasible solution to Problem 2 exists, then a solution to Problem 1 may be found by relaxing the upper bound inequality on the size of the second block. Then, partitioning is applied iteratively to the second block of the partition until the area and pinout capacity constraint is met. The problem can be stated as follows:

Problem 3: *Partition a hypergraph $\mathcal{H}$ into two hypergraphs $\mathcal{H}_i$, $i = 1,2$ such that the area-cost $A_1 \leq \overline{A}$, the pinout-cost $C_1 \leq \overline{C}$, the overall latency $\lambda \leq \overline{\lambda}$ and the cost function $f = \alpha C + \beta(\lambda - \overline{\lambda}) + \gamma(\overline{A} - A_1)$ is minimal.*

Note that the last term in the cost function is a heuristic to achieve full utilization of the first block. In the sequel we describe a set of algorithms to solve Problems 2 and 3. Due to the similarity of the two problems, we will address the solution to Problem 2 in detail, and we will leave as comments the modifications needed to solve Problem 3.

5 Partitioning Algorithms

Generic network flow algorithms for hypergraph partitioning were investigated by Lawler [18] [35]. Generally max-flow/min-cut algorithms provide an exact solution to the problem of finding min-cut between vertex pairs. However, one common characteristic of most flow algorithms is that they inherently tend to favor unequal partitions which naturally give the lowest cut value. There have been attempts to improve the situation by ratio cut partitioning of circuits [19]. Since in our problem we want to obtain *constrained* partitions, we have focused on algorithms that support constraints on the sizes of the resulting partitions.

The partitioning algorithms currently available in *Vulcan* are based on two iterative improvement schemes: *Kernighan-Lin* and *Simulated-Annealing* algorithm. Iterative improvement is achieved by two kinds of moves: single-vertex displacement or swap of two vertices. Both Kernighan-Lin (KL) and Simulated-Annealing (SA) algorithms fall in the general category of *local neighbourhood search*

algorithms. Neighbourhood search algorithms try to achieve an optimal solution (also referred to as a *configuration*) by searching for favorable solution (that is lower cost solution) in the neighbourhood of existing solution in a space of feasible solutions. The final quality of the obtained solution very much depends upon the ability of the algorithm to *reach* to globally optimal solutions. The neighbourhood search is generally carried out either by *first improvement*, i.e., select the first favorable configuration in the neighbourhood, or by *steepest descent* which exhaustively searches the neighbourhood for the best configuration. First improvement is faster but the optimality of the solution can not be guaranteed. Exhaustive neighbourhood search, on the other hand, increases the run time considerably. The KL takes an intermediate search approach by performing a variable-depth local search that tries to achieve optimum by allowing non-optimal (*hill climbing moves*) in a *group* of otherwise favorable moves. This evaluation of cost function over a group of moves (rather than a single move) enhances the neighbourhood search considerably. Partitioning results obtained using Kernighan-Lin are at least locally optimal with respect to exchange of single pair of vertices since at every iteration it will always find the most favorable single swap if such a swap exists.

On the other hand, simulated annealing is a probabilistic search algorithm that tries to achieve global optimum by accepting individual unfavorable moves at a rate which is a function of the 'annealing temperature' parameter of the algorithm. In the case of the simulated annealing algorithm, we transform Problem 2 into an unconstrained optimization problem with penalty functions [20]. Then the cost function is a linear temperature-varying combination of the inter-block communication cost C', excess latency $(\lambda - \overline{\lambda})$, excess area $(A_i - \overline{A}; i = 1,2)$ and excess pin-out $(C_i - \overline{C}; i = 1,2)$. For problem 3, the excess area and pin-out are computed only for block $i = 1$ and the term $(\overline{A} - A_1)$ is added to the linear combination. Only the variation of the cost function is computed for each move as explained in the following paragraphs. Under conditions of equilibrium and a certain cooling schedule, theoretically the algorithm can reach global optimum with probability 1. We refer the reader to [20] and [21] for the details of the algorithms. In the following paragraphs we concentrate on the moves and computation of the cost function.

For ease of explanation, we describe first partitioning of hardware models with no shared resources (i.e. no hyperedges) and no hierarchy. Therefore the hardware model is a directed graph. A partition is described by flagging each vertex with a tag that can take either one of two values. An example is shown in Figure 2.

We call *communication cost* the size of the data transferred between two blocks and its associated control. The pin-out C_i for the ith block is the sum of the cost of I/O ports and communication cost. The variation in communication cost is computed by following the procedure outlined in [21]. The variation in area can be computed by adding (subtracting) the vertex area-cost for the block to (from) which the vertex has been displaced. In the case of swaps, the variations of the cost function can be computed as a sequence of two displacements.

As mentioned earlier the Kernighan-Lin algorithm provides iterative improvement by applying a sequence of group-exchanges of vertices across a constructively created initial partition of vertices. In a pass we consider possible exchange vertices which would lead to largest gain (or reduction in the cost function), keeping in view the area and pin-out constraints on the individual partitions. We then set

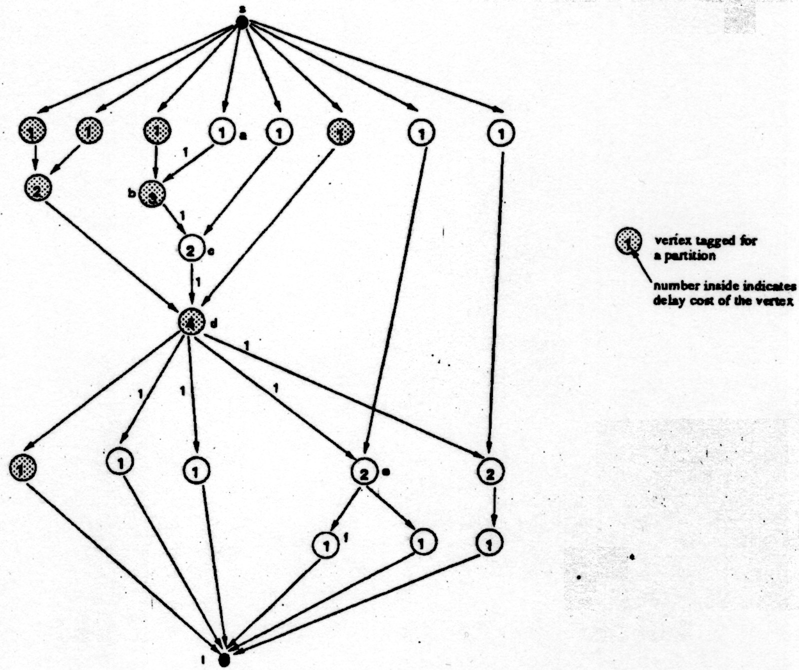


Figure 2: *Tagging and Latency Computation of SIF Graphs*

aside this pair and search of the next pair and so. We continue this process until all vertices have been exhausted and a sequence of possible exchanges and corresponding gains are identified. Now the set of vertices to be exchanged corresponds to the maximum partial sum of the gains. Note that the sum of all gains would be zero. This process is repeated until there is no further reduction in cost function. This algorithm is outline below. A partition obtained this way is optimal with respect to pairwise exchanges and corresponds to a locally optimum partition with respect to all possible moves.

Algorithm 1:

construct initial partition, A = $\{ q \}$, B = $\{ b_j \}$
<u>repeat</u> {
    X = Y = (null)
    <u>repeat</u> {
        pick vertex pair (a_i, b_i) for largest reduction in F
        A — A - $\{ a_i \}$; B — B - $\{ b_i \}$
        X — X $\cup$ $\{ a_i \}$; Y — Y $\cup$ $\{ b_i \}$
        compute reduction in cost function, g
    } <u>until</u> no more vertices available for exchange
    choose k to maximize $\Delta\mathcal{F} = \sum_{i=1}^{k} g_i$
    A — A $\cup$ $\{ b_1, \ldots, b_k \} \cup \{ a_{k+1}, \ldots a_n \}$
    B — B $\cup$ $\{ a_1, \ldots, a_k \} \cup \{ b_{k+1}, \ldots b_n \}$
} <u>until</u> $\Delta\mathcal{F} = 0$

In algorithm 1, the identification of vertex-pair that leads to maximum reduction in the cost function is complicated by the fact that any potential exchange will affect not only communication cost,C, but also the overall latency of the SIF graph. While it is relatively straightforward to incrementally update the changes in communication cost using Kernighan-Lin scheme, the evaluation of changes in latency is more complex. The *latency* of a partitioned graph is defined as the length of a modified critical path that includes edge weights, being 1 the weight of the edges that join vertices in different blocks and 0 otherwise. As an illustration, consider the example in Figure 2. In this case, the latency of the partitioned graph is determined by the longest path (s, a, b, c, d, e, f, t) to be 17 cycles of which 4 cycles are due to data transfer in the partitioned structure. Unlike area and communication costs, it is not possible to compute and update the latency *incrementally* for a generic move without traversing the subgraph(s) induced by the vertices that are successors of the moved vertex (vertices). This requires a rescheduling of this subgraph which can be done linear-time, whereas area and communication cost updates can be done constant time.

The original algorithm [21] applies to unconstrained optimization problems with $\beta = 0$. The computation of the full cost function for the selection of each move is computationally expensive and would not make the performance measure of the algorithm competitive with the simulated annealing approach. The restriction of the cost function to ΔC only corresponds to applying the original algorithm [21]. However, the satisfaction of the constraints of Problems 2 and 3 would require checking the feasibility of a solution before each exchange of vertices. This requires choosing the sequence of moves that minimizes the cost function and that leads to a feasible solution as well. For this reason, we have considered following two intermediate strategies.

One strategy consists in computing an *approximation* of the latency at each move and incorporating it in the cost function. Moves such that the latency approximiation violates its bound are discarded, as well as those that violate area and pinout constraints. Latency is affected by vertex displacement/swaps on the critical path, that may change after each move. Latency is approximated by assuming that the critical path does not change for the set of moves considered in the inner loop of the Kernighan-Lin algorithm. The local variation of the latency estimate can be easily computed for individual moves by checking the tags of the predecessor and successor vertices of the vertex being moved. The critical path is updated only when an exchange of vertices takes place, i.e. at the exit from the inner loop of the Kernighan-Lin algorithm. Clearly, as the number of swaps in an iteration grows, this approximation gets worse. Note that an exact latency computation would require the computation of the changes to *all* paths in the graph. The approximation provides a lower bound on actual latency because it cannot be inferior than the delay of *a* path in the graph. However, the actual latency may be larger than the bound because of a change of the critical path. Due to a vertex displacement, there may exists another critical path which is adversely affected. Also by definition of latency, the approximate latency will never exceed actual latency since it is always equal to the delay of *a* path in the graph. For this reason, feasibility of a configuration must be checked by computing the exact latency before exchanging groups of vertices. Algorithm 2 below outlines this approach:

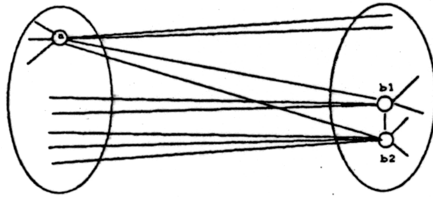
Algorithm 2:

```
construct initial partition, A = { a_i }, B = { b_j }
repeat {
    compute longest path, L
    X = Y = (null)
    repeat {
        pick vertex pair (a_i, b_i) for largest estimated reduction in F
        A — A - { a_i }; B — B - { b_i }
        X — X ∪ { a_i }; Y — Y ∪ { b_i }
        estimate reduction in cost function, g based on latency gain
    } until no more vertices available for exchange
    choose k to maximize ΔF = Σ_{i=1}^{k} g_i
    A — A ∪ { b_1 , .., b_k } ∪ { a_{k+1}, .. a_n }
    B — B ∪ { a_1 , .., a_k } ∪ { b_{k+1}, .. b_n }
} until ΔF = 0
```

latency gain in the above algorithm is computed by the following:

```
consider a vertex b to be moved:
  if vertex b ∈ longest path and a—b—c ∈ longest path then
     if a, b, c belong to same partition then
        latency gain = -2
     else if only b, c belong to the same partition then
        latency gain = 0
     else if a, c belong to same partition then
        latency gain = 2
  else
        latency gain = 0
```

Further gain in speed is possible at the cost of reduction in power of the algorithm by using a simpler cost function, $\mathcal{G}$, of communication cost, excess area and excess pin-out (i.e., excluding latency) in selecting the set of vertices to be exchanged in the inner-loop of the above algorithm. Once such a group of vertices is identified we perform latency computation and the new overall cost function, $\mathcal{F}$. A reduction in $\mathcal{F}$ leads to acceptance of the vertex set exchange. Clearly, in such an approach we are looking for the optimum $\mathcal{G}$ function which also leads to a lower $\mathcal{F}$ value. While a minimum in $\mathcal{F}$ would not be a minimum in $\mathcal{G}$, from our experimental results we obtain a good correlation between optimal values of the two functions.



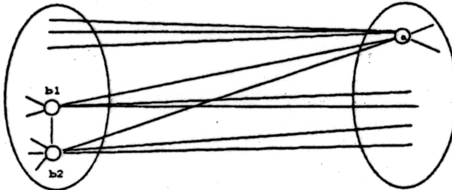


Figure 3: *Computation of Communication Cost in Hypergraphs*

5.1 Partitioning with Shared Resources

Let us comment now on the case in which hardware resource are shared. We assume that appropriate serialization of the shared tasks is represented by the graph edges [4]. Therefore the latency computation can still be done by graph traversal. Since shared resources cannot be split across partitions, the moves are now limited to the hyperedges. In other words, objects of the moves are hardware resources that are either dedicated or shared. The algorithms described before be still be used with minor modifications, related to the move selection and cost function evaluation. In particular, the evaluation of communication cost function gain presented in [21] is extended to include movement of a group of vertices represented by the corresponding hyperedges.

Using terminology presented in [21], suppose we wish to partition a given set of vertices into a bipartition A, B and let C = $\{c_{ij}\}$ represent the associated inter-connect cost matrix. We assume that

$$c_{ij} = \begin{cases} 1 & \text{if there is an edge between } i \text{ and } j \\ 0 & \text{otherwise} \\ 0 & \text{if } i = j \end{cases}$$

Let us define for each vertex, $a \in$ A, an *external inter-connect cost* as $E_a = \sum_{y \in B} c_{ay}$ and an *internal inter-connect cost* by $I_a = \sum_{x \in A} c_{ax}$. The total inter-connect cost due to partitions A and B is given by $T = \sum_{A \times B} c_{ab}$.

The reduction in inter-connect cost (i.e., the communication cost function) due to exchange of an hyperedge, $h_1 = \{a\}$ for an hyperedge $h_2 = \{b_1 . b_2\}$ can be determined as follows: let z be the

total inter-connect cost due to all connections between A, B that do not involve h_1 or h_2, that is, $z = \sum_{A-h_1 \times B-h_2} c_{ab}$. Then total inter-connect cost before exchange is given by:

$$T = z + E_a + E_{b_1} + E_{b_2} - c_{ab_1} - c_{ab_2} \quad (1)$$

After exchange of h_1 and h_2, the new total inter-connect cost can be expressed as (Figure 3):

$$T_{new} = z + I_a + c_{ab_1} + c_{ab_2} + (I_{b_1} + I_{b_2} - 2 \cdot c_{b_1 b_2}) \quad (2)$$

Thus the reduction in inter-connect cost due to exchange of $h_1 = \{a\}$ and $h_2 = \{b_1, b_2\}$ is given by

$$T - T_{new} = D_a + D_{b_1} + D_{b_2} - 2(c_{ab_1} + c_{ab_2}) + 2(c_{b_1 b_2}) \quad (3)$$

where D values represent the difference in number of external and internal connections, i.e., $D_a = \sum_{y \in B} c_{ay} - \sum_{x \in A} c_{ax}$.

Similarly, the reduction in communication cost function due to exchange of vertex-subset $\{a_1, a_2\}$ with vertex-subset $\{b_1, b_2\}$ can be expressed as:

$$D_{a_1} + D_{a_2} + D_{b_1} + D_{b_2} - 2(c_{a_1 b_1} + c_{a_1 b_2} + c_{a_2 b_1} + c_{a_2 b_2}) + 2(c_{a_1 a_2} + c_{b_1 b_2}) \quad (4)$$

Having computed D values once, these can be incrementally updated after each exchange using the following expressions:

$$D'_x = D_x + 2(c_{xa_1} + c_{xa_2}) - 2(c_{xb_1} + c_{xb_2}) \quad \forall x \in A - \{a_1, a_2\} \quad (5)$$

$$D'_y = D_y + 2(c_{yb_1} + c_{yb_2}) - 2(c_{ya_1} + c_{ya_2}) \quad \forall y \in B - \{b_1, b_2\} \quad (6)$$

In general, when exchanging vertices in two hyperedges, $h(a) = \{a_i\}$ and $h(b) = \{b_j\}$, the reduction in communication cost function is given by:

$$\sum_{a_i \in h(a)} D_{a_i} + \sum_{b_j \in h(b)} D_{b_j} - 2\sum c_{a_i b_j} + 2\sum (c_{a_i a_j} + c_{b_i b_j}) \quad (7)$$

and the D values can be incrementally updated using the following expressions:

$$D'_x = D_x + 2\sum c_{xa_i} - 2\sum c_{xb_j} \quad \forall x \in A - \{a_1, a_2, \ldots\} \quad (8)$$

$$D'_y = D_y + 2\sum c_{yb_j} - 2\sum c_{ya_i} \quad \forall y \in B - \{b_1, b_2, \ldots\} \quad (9)$$

For a given set, S, of n vertices, the D values are computed for all vertices at the beginning of an iteration at $O(n^2)$ cost since for each vertex in S all other vertices must be considered. After a pair of hyperedges (vertex sub-sets) are selected for exchange, and before the next pair is selected, the D values can be incrementally updated in $O(n)$ time using relations (8), (9) above. Thus the total updating time in a pass would grow as $O(n^2)$. The main effect of extending Kernighan-Lin algorithm to hypergraphs is seen in the running time required to select the best of group of vertices for exchange at each pass. This search time would grow as n^3 rather than $n^2 \log n$ due to the fact that in original KL, the strategy used in finding the best swap-pair in each pass takes $O(n \log n)$ time while in the extended KL for hypergraphs, this search takes $O(n^2)$ time.

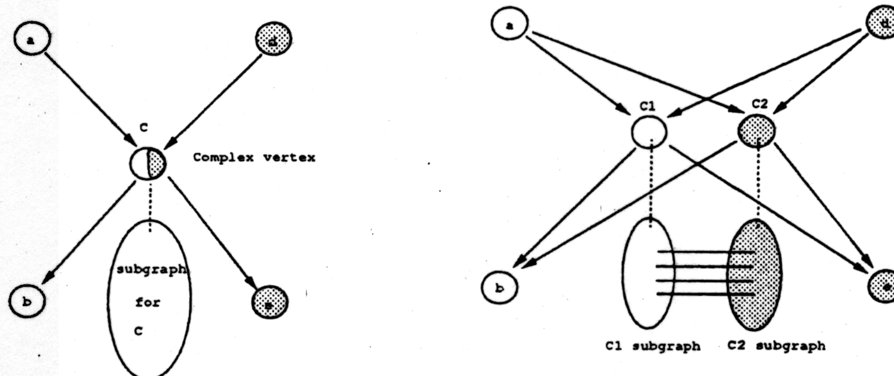


Figure 4: *Partitioning of Complex Condition-less Vertices*

5.2 Partitioning of Hierarchical Hypergraphs

Let us now consider hierarchical hypergraphs. According to our functional model [4], model call, looping and conditional constructs are represented by *complex* vertices and by using hierarchy. These vertices are not partitioned if a solution is found in terms of partitioning the hypergraph at the root of the hierarchy where complex vertices are considered indivisible. Otherwise, at descending levels of hierarchy the partitioning algorithm identifies the complex vertex with the largest associated area-cost. If the complex vertex corresponds to a model call then the called model can only be partitioned if it is considered dedicated to that instance only. In case of shared procedure calls no further partitioning of the shared model is possible. In order to preserve the original sequencing dependencies for correct behavior, when the called model is partitioned into two sets of operations it is essential to duplicate the complex calling vertex corresponding to the two model calls as shown in Figure 4. The calling vertex, C, is duplicated into two calling vertices, $C1$ and $C2$ each of which calls the respective partitioned model. Note that dependency edges $(a,c).(c,b).(d,c).(c,e)$ are modified accordingly in order to preserve the original sequencing dependencies.

The hypergraph called by the complex vertex is then considered for partitioning. An initial partition, satisfying resource sharing constraints, is then constructed and iteratively improved. This process is applied top-down in the hierarchy until an optimal overall partition is found. It is important to note that this formulation of hierarchical hypergraph partitioning preserves the ability of incrementally updating the communication cost as originally presented for flat graphs in [21]. [2]

As mentioned earlier, in the sequencing graph model the conditional execution paths are specified through the use of hierarchy, a complex conditional vertex may, therefore, call more than one hypergraph

[2] The effect of complex vertex duplication on communication cost, without modifying the actual graph topology, is achieved as follows: the complex vertex to be partitioned becomes *gray* instead of *black* or *white*, and all the edges connected to such a (gray) vertex are considered going to the 'external' group (i.e., new D_value = #edges connected to a gray vertex); therefore, the gain in communication cost is = external - (external + internal) = - internal cost.

It can now be easily shown that cut cost of a gray vertex is equivalent to cost of two vertices connected as shown in Figure 4. That is, gain in the cost function = old cost - new cost = - internal (*when turning gray from black or white*); + internal (*when turning black or white from gray*)

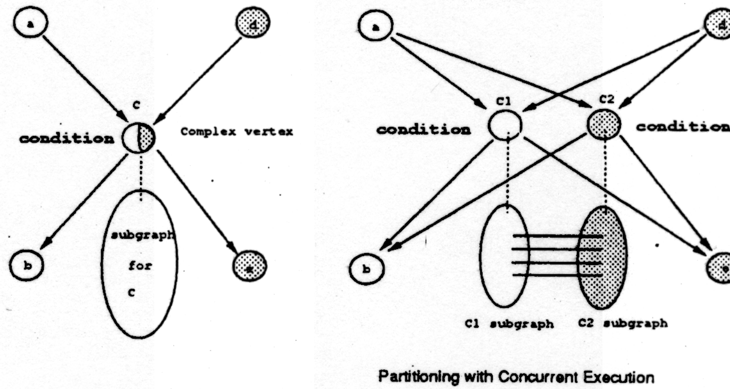


Figure 5: *Partitioning of Complex Conditional Vertices: Loops*

(only one of which is invoked for execution due to the condition semantics). While partitioning such vertices it is important to maintain the semantics of the conditional vertex by ensuring that the conditional logic is properly replicated while partitioning. In case of *conditional* complex vertices such as related to looping constructs (repeated model call) and to those representing conditional constructs (selective model call), the complex calling vertex also contains logical conditions to either terminate the call (as in case of a loop) or to select one of the many conditionally called models. In these cases also the complex vertex replication with additional sequencing edges is necessary in order to preserve the original sequencing behavior. In addition, however, the conditional logic at the complex vertex can be treated differently depending upon the overall goal of partitioning. If the goal of partitioning is to create concurrently executing hardware models, then the conditional logic at the complex vertex is duplicated in the two complex vertices as shown in Figure 5. This duplication is necessary to preserve the concurrency of two hardware models created after partitioning. It is also possible to partition such that one partition acts as a dedicated set of resources to the rest of the hardware. In this case any group of operations can be enclosed in a procedure model which can be 'called' by the other partitions. Such a partitioning scheme avoids overheads due to conditional logic replication. The difference between these two partitioning approaches is explained further in Section 6.3.

Partitioning of conditional vertices is achieved by duplicating first the complex vertex (which represents the operation that computes the conditional clause) and the source and sink vertices of the hypergraphs corresponding to the body of the conditionals (i.e. corresponding to the TRUE and FALSE clause), as shown in Figure 6. Then these hypergraphs are assigned to different blocks as an initial partition. This partition is then iteratively improved, and, therefore, one (or both) bodies of the conditional can be split among the two blocks. In case of resource sharing across conditions, the vertices sharing resources are moved together to the same partition. In any subsequent moves any group of vertices sharing a resource are always kept together.

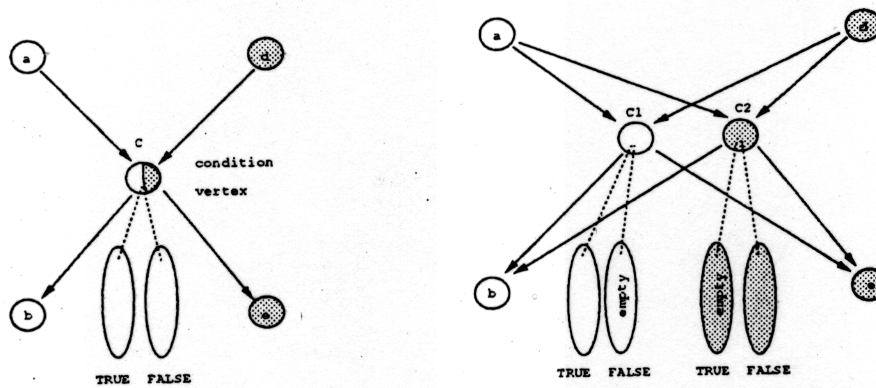


Figure 6: *Partitioning of Complex Conditional Vertices: Conditions*

6 The Vulcan Partitioning System

The partitioning algorithms described in previous section have been implemented in program *Vulcan*. *Vulcan* is written in the C programming language and is integrated with the Olympus Synthesis System. Figure 7 shows important components of Olympus related to *Vulcan*. We assume that the hardware being designed is synthesized from a high-level description in HardwareC (or any other HDL) under a maximum area and timing constraint on the overall hardware latency. This situation is fairly typical of hardware prototyping using programmable gate arrays, that have a limited capacity in terms of gate count. By using the same functional model in a HDL, both a multi-chip prototype and the final implementation can be synthesized automatically. Bounds on the latency of the prototype are important to ensure that accurate performance measures can be derived from it.

The hardware description is first compiled it into a SIF model using *Hercules* that also applies a series of compiler-like behavioral optimizations to the SIF model. Such a description can be simulated at the functional level by program *Ariadne*. Program *Hebe* is used to perform structural synthesis of the hardware model which is then passed on to program *Ceres* for logic synthesis and technology mapping. If a design configuration is achieved that meets the required area and latency constraints, then clearly no partitioning is required. However, when such a structure can not be found, then partitioning and/or resource sharing techniques [22] can be used to overcome the area limitations. In case of resource sharing, however, sharing and subsequent serialization of parts of the hardware may lead to a slower implementation that does not meet the latency requirements. Therefore, the designer may need to split the hardware over two or more chips to satisfy both the area and latency requirements. As described earlier *Vulcan* supports use of resource sharing constraints by means of hyperedges in the input hardware graph model. Resource sharing constraints can be specified either in *HardwareC* descriptions by use of attributes or these can be manually applied by *Hebe* during structural synthesis [23]. Given a set of resource sharing constraints, resource allocation and binding can be performed using *Hebe*. In presence of resource constraints it is assumed that appropriate serialization of operations on shared resources has been performed after the resource binding phase in order to avoid any potential resource conflicts.

Since partitioning may introduce timing penalties due to the inter-chip communication delays. There-

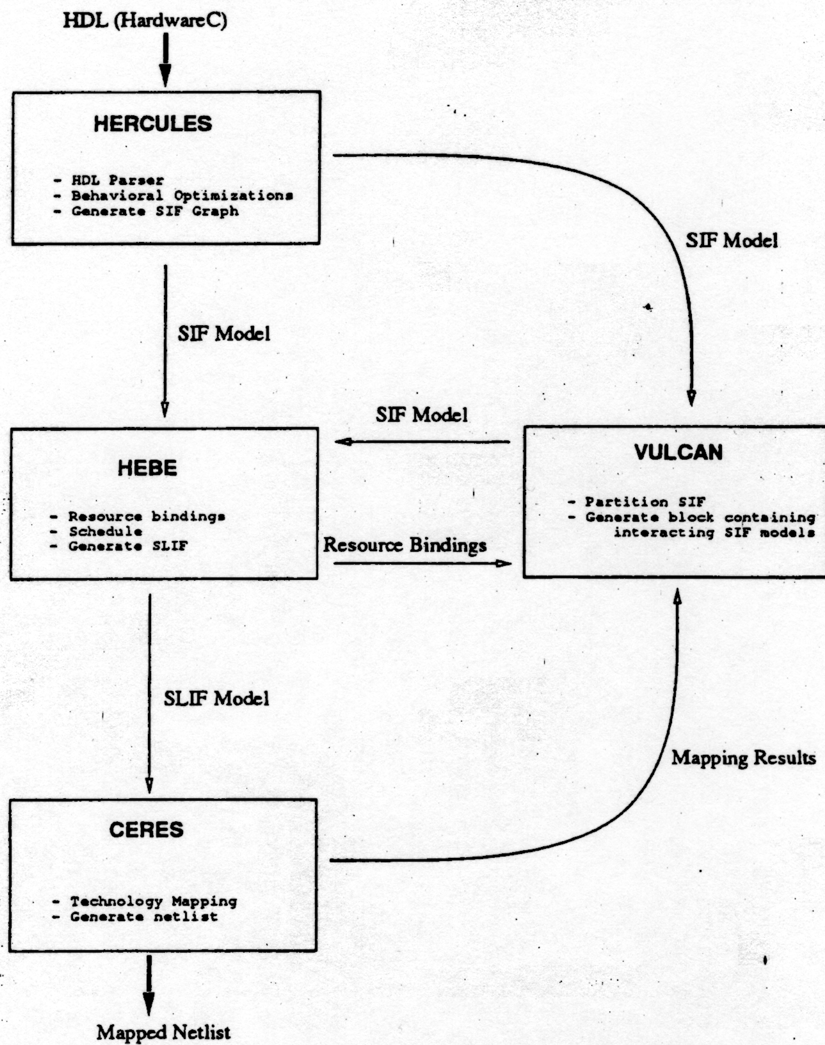


Figure 7: *Interaction of Vulcan with Other Programs in The Olympus Synthesis System*

fore, we choose a design configuration that satisfies the latency constraint as a starting point for partitioning. Thus the search for a binding (that defines the hardware sharing) is done prior to partitioning (by program *Hebe*), and it benefits the partitioning method in providing a starting point with an estimated area smaller than an unbound configuration. A SIF model of the hardware along with resource binding performed by *Hebe* is passed to *Vulcan*. *Vulcan* returns a set of (partitioned) SIF models with the appropriate interblock communication hardware. *Vulcan* also provides routines to assemble the partitioned blocks into an inter-connecting block structure which can be simulated by *Ariadne* for correct functionality. Subsequently, *Hebe* can be invoked to generate the control circuit and the corresponding logic-level hardware description of the partitioned system model.

For some resource binding configurations it is possible that no suitable partition can be found satisfying the latency constraint. In such cases, partitioning should be reapplied after changing the resource binding configuration in *Hebe*. For a given set of resource constraints, *Hebe* generates possible binding of resources to operations in the hardware model. Further these bindings can be prioritized according to user-specified cost criterion. This binding information is utilized by the partitioning algorithm in generation of SIF graphs which are then partitioned under area, pinout and latency constraints.

Hebe creates the Structure/Layout Intermediate Form (SLIF) model after structural synthesis of the hardware models presented by *Vulcan*. Note that in *Vulcan* only the data-path portion of a chip is accurately modeled in terms of gate count and delay by applying logic synthesis and technology mapping to the individual hardware resources, whereas only estimates are used for the area and delay of the control portion and for the wiring delays. However, after structural synthesis, program *Ceres* can be used for logic synthesis and technology mapping of the complete hardware model. Area and performance results after mapping can then be passed to *Vulcan* in order to reiterate on partitioning algorithms with appropriately adjusted partition parameters. This way *Vulcan* provides ability to perform partitioning at the functional abstraction level while at the same time using the results from technology mapping of the resulting hardware to fine tune the partitioning metrices to meet desired area and performance objectives.

Vulcan's main activities can be broadly divided into two categories:

PARTITIONING of a hardware model into interacting models

- Partitioning accomplished by 'tagging' a hardware model graph as shown in Figure 2.
- Creation of individual hardware models through *behavior-preserving transformations* necessary to carry out hardware partitioning.

ASSEMBLY of various interacting hardware models into a single 'block' structure.

6.1 Models and Partition Information Maintained in Vulcan

Vulcan maintains a list of hierarchically connected hardware models. A model can be of structural (interconnection block) or sequencing type. The sequencing models are further classified as process or procedural models. Figure 8 shows data organization used in *Vulcan*.

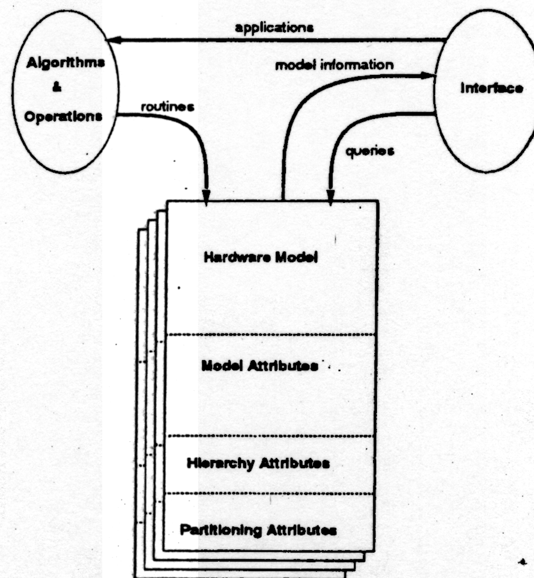


Figure 8: *Data Organization in Vulcan*

6.2 Partitioning of Model Graphs

As described earlier, a partition of the model graph is represented by tagging of its operation vertices by different indices belonging to different partitions. When performing bipartitions, vertices are tagged *black* or *white* according to partitions they belong to. In addition, complex vertex replication and partitioning is achieved by turning them *gray* as described in Section 5.

6.3 Creation Of Hardware Models From Partitioned Model Graph

From a partition of vertices in the input sequencing graph, a set of behavior-preserving graph transformations are applied in order to generate resulting interacting sequencing graphs. These transformations implement the abstract cut of the graph into corresponding hardware partitions. In order to understand these transformations, it is important to note the process/procedure paradigm of hardware representation in the SIF models. A process in SIF model refers to a piece of hardware that restarts itself upon completion of execution. On the other hand, procedure refers to a hardware model which is executed only when it is called by another hardware model [31]. Without any loss of generality, let us assume that the goal of partitioning is to create a bipartition of a given hardware model. It is then possible to implement the final hardware as either two *concurrently* executing chips which interact with each other only through ports of communication, or as an assembly where one chip essentially serves as a dedicated resource to the other. In the first case, in order to generate interacting hardware pieces which execute *concurrently* both the corresponding hardware models should be created as process models. In presence of conditional execution paths to different hardware resources in the two models, the concurrency requirement requires that appropriate conditional execution paths be replicated in the two chips so that their execution can proceed in parallel. On the other hand, if one chip acts as a dedicated resource there would be no need

to replicate its control activation logic which can reside on the 'master' chip. The master chip is then implemented as a process while the dedicated resource chip is implemented as a procedure model in *Vulcan*.

The tradeoff between the two approaches depends upon the area increase due to replication of control paths versus performance penality due to loss of concurrency in the two hardware pieces. *Vulcan* supports both forms of partitioning under user control.

The following transformations are applied:

Replication - If portions of a conditional execution path are partitioned into separate blocks, the vertex representing the condition (loop condition or a case condition) is replicated with the same successor and predecessor dependencies as the original. The original and the replicated conditional vertices belongs to separate partitions in order to facilitate concurrent evaluation of the conditional in the two partitions.

Insertion of IBC vertices - From the definition of the SIF graph model, no hyperedges representing only the resource dependencies are ever cut by the partitioning algorithms presented before. The only edges that may cross a partition are either (a) containing a data dependency or (b) dependency related to control flow. In case of data dependency, it clear that some sort of interface buffer is needed to hold the data to be transferred across the partition. On the other hand, the control dependency refers to synchronization of operations which may or may not require an explicit data transfer depending upon the final outcome of the schedule of operations in the two partitions. In case, an explicit data transfer is required, *Vulcan* inserts appropriate Inter-Block Communication vertices into the partitioned models. These vertices represent interface hardware necessary to support communication between hardware partitions.

6.4 Assembly of Interacting Hardware Models into A Single Structure

Having obtained separate interacting hardware models corresponding to a partition of the input graph model, the resulting multi-chip system can also be represented by an interconnection structure of partitioned model graphs. *Vulcan* provides the ability to obtain such a structural model by defining the inter-process communication channels as dedicated interconnection wires between separate pieces of hardware.

7 Examples

Program *Vulcan* was tested on a number of examples. First we present the partitioning results on sequencing graphs obtained from *HardwareC* descriptions of some hardware designs. We then analyze the results of partitioning for the fifth order digital elliptic wave filter [24] under different resource sharing situations.

Table 1 presents some results on partitioning of hardware models for implementation on Actel PGA chips each of which can accommodate at the most 546 cells. It is seen that *Elliptic Filter*, *Parker1986*,

Example	*Unpartitioned*		*Partitioned*	
	size	latency	size	latency
Elliptic Filter	689	20	333, 332	21
Parker1986	614	12	259, 240	14
Tseng	755	16	257, 264	21
Diffeq	1632	15	541, 308, 342	20

Table 1: *Partitioning for ACTEL PGA Implementations*

Example	*Size*	λ	$\overline{\lambda}$	*Size1*	*Size2*	λ'	$\lrcorner$	*CPU*
6502	211094	34	38	97155	113939	38	0.06	1546
Proc46	104092	14	15	44124	60028	15	0.03	1948
Proc34	95405	22	25	49882	46283	25	0.40	667
Proc8	22604	20	25	12986	9698	24	0.18	26
FRISC	16018	24	27	6148	9990	26	0.37	778
Elliptic	12542	20	21	6070	6532	21	0.24	11

Table 2: *Partitioning Under Latency Constraints*

and *Tseng* can be implemented in two Actel PGA (ACT1020) chips, whereas implementation of *Diffeq* would require use of three such chips.

Table 2 shows results of partitioning under latency constraints. 6502 refers to a commercial microcontroller containing 88 opcode. FRISC is a simpler 16-bit microprocessor[36] containing about 20 opcodes. Proc*nn* refer to subsets of microprocessor descriptions containing *nn* opcodes. The size and latency (λ) entries in columns 2 and 4 respectively refer to the original un-partitioned designs. Column 3 refers to the latency constraint, $\overline{\lambda}$ on the partitioned design. Size of the resulting partitions and latency (λ') of the partitioned model are reported in the subsequent columns. These partitioning results are reported for Kernighan-Lin algorithm using complete cost function. CPU times are reported in seconds while running on DecStation 3100 with 16 MBytes of memory.

Table 3 compares partitioning results obtained from different partitioning strategies on various examples. Examples EX1, EX2, EX3 refer to three different SIF models compiled from *HardwareC* descriptions. The size and latency entries in columns 2 and 3 respectively refer to the original un-partitioned design. Size of the resulting partitions and latency of the partitioned model are reported in subsequent columns. We have compared four heuristics: simulated annealing (SA), Kernighan-Lin with objective

function consisting of only area and pin-out costs (KL0), Kernighan-Lin with the complete objective function (KL1), and finally Kernighan-Lin where the latency is approximated as described in section 5 (KL2).

7.1 Effect of Resource Sharing: Elliptic Filter

We now show the effect of resource sharing on hardware partitioning using an example of fifth order digital wave filter[24]. The original filter description was translated into *HardwareC* from an ISP model [25]. The filter description consists of 26 add operations and 8 multiply (by 2) operations. In addition, the design also contains 15 I/O operations. Figure 9 shows the sequencing hardware model of the filter. As shown in the Table 3, the total size of the description, without any resource sharing, is 6458. This size is estimated using the literal count of various blocks. Latency of the unpartitioned filter is 16 cycles. The shaded region on Figure 9 shows the partition created by KL algorithm. This partition consists of two approximately equal sized blocks with 3 interblock communication ports. The overall size and latency in this case were 6488 and 17 respectively.

Now consider the case where only one adder is available to do all the add operations. In this case the total area cost is 508 and latency of the unpartitioned graph is determined by the longest execution path in the *serialized* hypergraph obtained by program *Hebe*. Grouping together all 'add' vertices on a single hyperedges results in a partition in which all 'add' operations are performed in a single partition. Such a cut, however, increases the overall latency significantly. This demonstrates the effect of resource sharing on optimality of resulting partitions. Greater resource sharing leads to larger communication costs which increases the size of individual partitions. For the elliptic filter we see that the effect of partitioning without any resource sharing is to increase area cost by about 0.5% and latency by 6%. However, the effect of partitioning the same filter with resource sharing increases the area cost by 9.0% and latency by 59% in case of four adders. Resource sharing with two adders increases area cost by 17.4% and latency by 55%. Finally, resource sharing with one adder increases area cost by 82% and latency by 23%.

Table 4 illustrates the effect of latency, area and pin-out constraints on partitioning results for the elliptic filter containing 26 add and 8 multiply resources. In contrast to the filter considered in Table 3, the multiply operations are now not restricted to be by 2 only. The multiply operations are instead modeled by calls to hardware blocks requiring two cycles per operation. Therefore, total size of the unpartitioned filter is 12542, considerably bigger than the filter description used in Table 2. The latency of the unpartitioned filter in this case is 20 cycles. In order to compare these results to results reported in [6], we have to impose the additional constraint of using 2 adders and 2 multipliers. In this case our algorithm yields a latency of 20 cycles, which is the same as in [6] when the I/O and inter-block communication delay is taken into account. No area comparison of the two approaches is meaningful due to different assumptions. It is observed that relaxing the latency limit from 21 to 22 reduced the impact on overall area from 1.93% to 0.60%. Relaxing the area limit from 7500 to 8000 reduced the increase on overall area from +0.60% to +0.48% for the same overall latency. Overall pin-out of the filter is 15. A pin-out constraint of 10 per block leads to a partitioned design which is 0.48% bigger.

Example	*Size*	*Latency*	*Partitioning Heuristics*	*Size1*	*Size2*	*Latency*
EX1	644	6	SA	438	256	10
	644	6	KL0	402	302	10
	644	6	KL1	402	302	11
	644	6	KL2	383	301	10
EX2	744	10	SA	513	311	15
	744	10	KL0	456	418	14
	744	10	KL1	456	418	16
	744	10	KL2	456	418	14
EX3	117	5	SA	73	74	9
	117	5	KL0	74	73	9
	117	5	KL1	74	73	9
	117	5	KL2	74	73	9
Elliptic Filter2	6458	16	SA	3244	3254	18
with no	6458	16	KL0	3001	3487	17
resource sharing	6458	16	KL1	3001	3487	17
	6458	16	KL2	3001	3487	17
Elliptic Filter2	1222	17	SA	738	706	26
with four	1222	17	KL0	711	621	29
adders	1222	17	KL1	688	656	27
	1222	17	KL2	688	656	27
Elliptic Filter2	746	20	SA	438	608	28
with two	746	20	KL0	453	423	31
adders	746	20	KL1	458	368	35
	746	20	KL2	403	453	31
Elliptic Filter2	508	30	SA	368	340	35
with one	508	30	KL0	448	480	37
adder	508	30	KL1	343	263	35
	508	30	KL2	343	263	35

Table 3: *Partitioning Under Resource Sharing Constraints: Elliptic Filter*

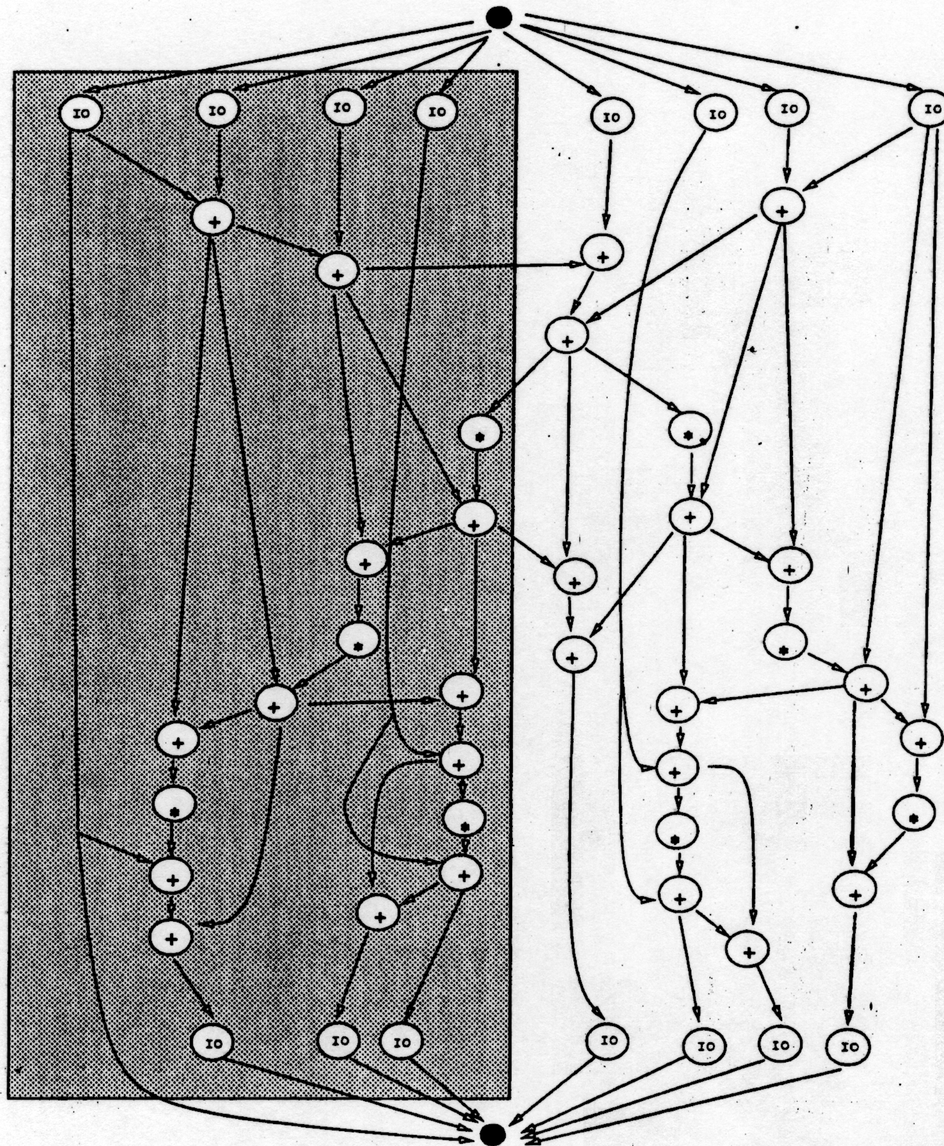


Figure 9: *Partitioned Elliptic Filter Sequencing Graph without Resource Sharing*

$\overline{\lambda}$	$\overline{A}$	$\overline{C}$	*Size1*	*Size2*	λ'	$\Delta Size$
21	none	none	6516	6506	21	+1.93%
22	none	none	6356	6346	22	+0.60%
23	none	none	5629	7073	22	+0.60%
22	7000	none	6356	6346	22	+0.60%
22	8000	none	6100	6562	22	+0.48%
22	9000	none	6070	6532	22	+0.24%
22	none	10	6100	6562	22	+0.48%
22	none	11	5363	7279	22	+0.39%
22	none	12	5825	6797	22	+0.31%

Table 4: *Elliptic Filter Partitioning Results Under Constraints*

However, relaxing this constraint improves the overall size of the partitioned design as shown.

8 Summary

The problem of hardware partitioning at the functional abstraction level under latency, area and pin-out constraints has been modeled as a constrained hierarchical hypergraph partitioning problem. We have explored algorithms for generating partitions based on the Simulated Annealing and on the Kernighan-Lin algorithms. The latter algorithm is faster than the former and yields almost comparable results. It exploits two different heuristics to deal with the constrained partitioning problem. An approximation algorithm to incrementally update the overall latency of the hypergraph is suggested. Such an approximation speeds up the algorithm significantly by avoiding the necessity to recompute graph latency during selection stage of exchange vertices.

The algorithms have been implemented in program *Vulcan* and tested successfully on benchmark examples. *Vulcan* provides an interactive work-bench to partition and assemble hardware models in accordance with overall area and performance objectives. *Vulcan* can be used in conjunction with other high-level synthesis tools [4] to explore multi-chip implementations of a given functional model. Interesting trade-offs can be achieved by considering partitioning concurrently to resource sharing. The latter technique reduces area requirement at the cost of higher graph latency due to extra serialization introduced. Similarly partitioning reduces area requirement per partitioned block but adversely affects overall latency due to inter-block delays. Clearly, when design constraints can be satisfied by resource sharing alone, then partitioning is not required. However, partitioning techniques are required in the remaining cases, i.e. when chip area limitations can not be satisfied by resource sharing without violating latency constraints.

9 Acknowledgements

David Ku implemented the programs *Hercules* and *Hebe* used for behavioral and structural synthesis respectively in the Olympus Synthesis System. Frédéric Mailhot implemented programs *Ceres* and *Mercury* used for logic synthesis and technology mapping. Thomas Truong implemented programs used for behavioral simulation of the hardware description. This research was sponsored by NSF-ARPA, under grant No. MIP 8719546 and, by AT&T and DEC jointly with NSF, under a PYI Award program. We acknowledge also support from ARPA, under contract No. J-FBI-89-101.

10 Appendix A: Vulcan Commands Listing

SIF Related

- `readsif` - Reads a SIF model description as specified by further arguments
- `writesif` - Writes a SIF model description from memory to a file as described by further arguments
- `printsif` - Prints a SIF model description
- `sif_augment` - Augments a SIF model with appropriate data structures needed by *Vulcan*
- `buildrp` - Builds a table of resources used by a SIF model
- `extract_dataflow` - Modifies SIF model graph to contain only data flow dependencies
- `flatten` - Flattens a SIF model graph
- `cut` - Performs a cut of the input model graph based on tagging information provided by partitioning algorithms
- `evaluate logic, evl` - Based on a technology library, this commands performs evaluation of all combinational logic blocks in a SIF model by first simplifying the combinational block and then performing a technology mapping
- `make_block` - Constructs the interconnection block structure containing partitioned models

Partition Related

- `kl` - Runs Kernighan-Lin partitioning algorithm on a given hardware model graph
- `sa` - Runs Simulated-Annealing
- `untag` - Untags a given hardware model of all existing partitioning information
- `print parameters` - Prints currently set partitioning parameters in case no argument is given, else it prints partitioning parameters used to create a particular partitioned block
- `set parameters` - Sets following partitioning parameters:
  - latency model
  - cost model
  - alpha
  - beta
  - gamma
  - area limit

- pinout limit
- latency limit
- goal
- library

Synthesis Related

- `synthesize` - Based on synthesis goal (min area, min delay) this command invokes *Hebe* to synthesize a given SIF model graph into a structural (SLIF) model

Miscellaneous Utilities

- `show SIF graph` - Graphically displays a SIF model graph
- `show model hierarchy` - Graphically displays the hierarchy of SIF models used in a given hardware model
- `show model cut` - For a given hardware model this command graphically displays the partitions created from it
- `list models` - Lists names of all the models maintained by *Vulcan*
- `delete model` - Deletes a given model from the memory
- `rename model` - Renames a given model

References

[1] R. K. Brayton, R.Camposano G. DeMicheli, C.T. McMullen and R.H.J.M. Otten, "The Yorktown Silicon Compiler System", in D. Gajski editor *Silicon Compilation*, Addison Weseley, 1988, pp. 204-310.

[2] D. E. Thomas, E. D. Lagnese, R. A. Walker, J. A. Nestor, J. V. Rajan, R. L. Blackburn, "The Systems Architect Workbench", Kluwer Academic Press, 1989.

[3] M. C. McFarland,"Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions", *Proceedings of the 23rd Design Automation Conference*, pp. 474-480, June 1986.

[4] G. De Micheli, D. Ku, F. Mailhot, T. Truong, "The Olympus Synthesis System", *IEEE Design and Test of Computers*, Oct 1990.

[5] R. Gupta, G. De Micheli, "Partitioning of Functional Models of Synchronous Digital Systems", *ICCAD-90*, pp. 216-219, Nov 1990.

[6] E. Dirkes Lagnese, D. E. Thomas, "Architectural Partitioning for System Level Design", *Proc of the 26th DAC*, pp. 62-67, June 1989.

[7] M. C. McFarland, S.J., "Computer-Aided Partitioning of Behavioral Hardware Descriptions", *20th Design Automation Conference*, pp. 472-478, 1983.

[8] J. V. Rajan, D. E. Thomas, "Synthesis by Delayed Binding of Decisions", *Proc. of the 22nd DAC*, pp. 367-373. ACM/IEEE, June 1985.

[9] M. C. McFarland, "The Value Trace: A Data Base for Automated Digital Design", *Design Research Center*, Carnegie-Mellon University, Report **DRC-01-4-80**, December 1978.

[10] K. Kucukcakar, A. Parker, "CHOP: A Constraint-Driven System-Level Partitioner", University of Southern California *Technical Report CEng 90-26*, November 1990.

[11] R. Camposano, R. K. Brayton, "Partitioning Before Logic Synthesis", *Proc. 22nd DAC*, pp. 324-326, November 1987.

[12] R. Camposano, A. Kunzmann, W. Rosenstiel, "Automatic Data Path Synthesis from DSL Specifications", *Proc ICCAD'84*, pp. 630-635, Port Chester, New York, October 1984.

[13] A. C. Parker, J. Pizarro, M. Mlinar, "MAHA: A Program for Datapath Synthesis", *Proc of the 23rd DAC*, pp. 461-466, Las Vegas, June 1986.

[14] C. Berge, "Graphs and Hypergraphs", North-Holland, 1973.

[15] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, A. Wang "MIS: A Multiple-Level Logic Optimization System", *IEEE Transactions on CAD/ICAS*, Vol. CAD-6, No. 6, November 1987, pp. 1062-1081.

[16] D. Ku, D. Filo, G. De Micheli, "Optimizing Control by Delayed Execution of Operations", *Fifth International Workshop on High-Level Synthesis*, pp. 118-125 1991.

[17] D. Ku, G. De Micheli, "Relative Scheduling Under Timing Constraints", Stanford Technical Report, **CSL-TR-89-401**, Nov 1989, and *Proceedings of the* 27th *ACM/IEEE Design Automation Conference*, Orlando, Floria, June 1990.

[18] E. L. Lawler, "Çutsets and Partitions of Hypergraphs", *Networks*, **no. 3**, pp. 275-285.

[19] Y-C. Wei, C-K. Cheng, "Towards Efficient Hierarchical Designs by Ratio Cut Partitioning",*ICCAD-89*, pp. 298-301.

[20] P. J. M. van Laarhoven, E. H. L. Aarts, "Simulated Annealing: Theory and Applications", D. Reidel Publishing Company, 1987.

[21] B. W. Kernighan, S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs", *The Bell System Technical Journal*, **49**(2) Feb 1970.

[22] M. C. McFarland, A. C. Parker, R. Camposano, "Tutorial on High-Level Synthesis", *25th Design Automation Conference*, 1988, pp. 330-336.

[23] D. Ku, G. De Micheli, "Synthesis of ASIC's With Hercules and Hebe", Stanford Technical Report, **CSL-TR-91-461**, Feb 1991.

[24] P. DeWilde, E. Deprettere, R. Nouta, "Parallel and Pipelined VLSI Implementation of Signal Processing Algorithms", In S. Y. Kung, H. J. Whitehouse, and T. Kailath (ed), *VLSI and Modern Signal Processing*, pp. 257-264. Prentice-Hall, 1985.

[25] E. Dirkes Lagnese, *Private Communication.*

[26] M. Beardslee, C. Kring, R. Murgai, H. Savoj, R. K. Brayton, A. R. Newton, "SLIP: A Software Environment for System Level Interactive Partitioning",*ICCAD-89 Digest of Technical Papers*, Santa Clara, November, 1989.

[27] W. Donath, "*Physical design Automation of VLSI Systems*", chapter on logic partitioning, Benjamin-Cummings Publishing Company, 1988.

[28] J. Rabaey, H. De Man, J. Vanhoof, G. Goossens, F. Catthoor, "CATHEDRAL-II: A Synthesis System for Multiprocessor DSP Systems", in D. Gajski editor *Silicon Compilation*, Addison Weseley, 1988, pp. 311-360.

[29] S. Kirkpatrick, D. Gelatt and M. Vecchi, "Optimization by Simulated Annealing" *Science*, 220, N.4598, pp. 45-54, May 1983.

[30] G. De Micheli and D. Ku, "High-level Synthesis and Optimization Strategies in Hercules and Hebe" *EURASIC, Proceedings of the European Conference on ASIC design*, Paris, May 1990.

[31] D. Ku, G. DeMicheli, "HardwareC - A Language for Hardware Design (version 2.0) ", Stanford Technical Report **CSL-TR-90-419**, April 1990 (version 2.0).

[32] B. Borriello, E. Detjens, "High Level Synthesis: Current Status and Future Directions", *Proc of the 25th DAC*, pp. 477-482, ACM/IEEE, June 1988.

[33] H. R. Charney, D. L. Plato, "Efficient Partitioning of Components", *IEEE Design Automation*, July 1968, pp. 16.0-16.21.

[34] D. M. Schuler, E. G. Ulrich, "Clustering and Linear Placement", *Proc. 9th Design Automation Workshop*, 1972, pp. 50-56.

[35] L. R. Ford, D. R. Fulkerson, "Flows in Networks", *Princeton University Press*, 1962.

[36] J. R. Southard, "MacPitts: An Approach to Silicon Compilation", *IEEE Computer*, 1983 (pp 74-82).