Specification Partitioning for System Design

Frank Vahid and Daniel D. Gajski
Department of Information and Computer Science
University of California, Irvine, CA, 92717

Abstract

Behavioral partitioning can be applied to attain various goals, one of which is the satisfaction of chip-packaging constraints. Such partitioning heavily influences decisions made in subsequent structural design, and may therefore lead to higher performance designs and more efficient use of area and pins than possible when structure is designed before partitioning. Current behavioral partitioning approaches are limited in that they partition at the level of control/dataflow graph operations. We introduce a new approach which partitions entire computations of a behavioral specification, such as processes and procedures, into chip behavioral specifications. We demonstrate the approach's usefulness and highlight results of several examples.

1 Introduction

Given a behavioral specification, our overall aim is to implement it as custom hardware and/or as software. Behavioral partitioning can be used to achieve one or more of the following subgoals in which each partition is implemented:

- using a specific class of hardware technology, such as custom layout, ASICs, or off-the-shelf processor chips. The essential tradeoff is between design/manufacturing costs and system performance, both being heavily influenced by the technologies used. Note that hardware/software partitioning is encompassed by this goal.
- as a chip that satisfies chip-capacity constraints. Here performance is affected by the amount of interchip communication, while cost depends on the chip packages chosen within the technology being used. Note that different chip technologies will have differing size constraints in terms of the silicon-area, gates, or number of instruction-words, as well as differing pin and power constraints.
- on a single sequential processor, where there may be one or more processors per chip. In the context of behavioral partitioning, a processor is either a custom-designed controller and datpath, or an off-the-shelf component such as a microcontroller. Here fewer processors mean less cost and less interprocessor communication time, but more processors can yield more parallel execution.
- by a synthesis tool, separately from other partitions. If the partitions are chosen well, results obtained by the synthesis tool may be superior to those obtained by the tool when applied to the entire unpartitioned behavior. Also, the unpartitioned behavior might exceed the tool's runtime or memory capacity; partitioning can create smaller behaviors that can be handled by the tool.

These goals are closely related and may overlap. Various combinations of these goals result in partitioning problems which differ in the types and weights of constraints, in the techniques and accuracy of estimations made of the constrained parameters, and in the objective functions which evaluate partitionings based on the estimations and constraints. In this paper, we focus on the goal of partitioning a behavior to satisfy chip-capacity constraints while considering system-performance constraints. We assume a hardware (as opposed to software) implementation with a uniform chip technology.

The paper is organized as follows. In Section 2 we discuss the need for a new partitioning approach, outline our specification partitioning approach, and highlight its advantages. In Section 3 we describe details of our approach. In Section 4 we provide the results of partitioning several examples using the specification partitioning tool we are developing. In Section 5 we discuss future work.

2 Specification partitioning overview

Structural partitioning can be used to attempt to satisfy chip-capacity and system-performance constraints, but there are drawbacks when compared to behavioral partitioning. In a structural partitioning approach, a behavior is first converted to structure, and then the structure is partitioned among chips. While yielding accurate size and pin estimates, the approach ignores the fact that the design of the structure can be heavily influenced by the chip-partitioning. For example, two sequential procedures that each perform an addition can share an adder if both procedures are implemented on the same chip. If the procedures are separated among two chips, the same sharing results in poor performance due to interchip communication time; a better design would include two adders, one on each chip.

The effect of partitioning on the structural design supports the need for behavioral partitioning. There are essentially two levels at which such behavioral partitioning can be performed. At the *operation* level, dataflow-level operations such as addition and subtraction 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. Figure 1 shows goals that are achievable at each level. Most goals involve tradeoffs between cost and performance. In general, algorithmic-level goals relate to larger systems than

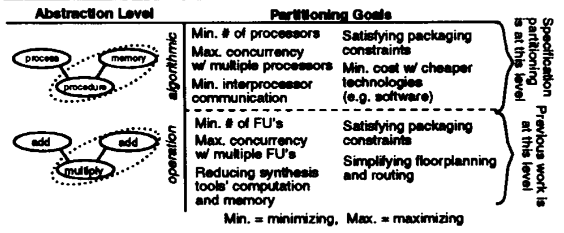

Abstraction Level
Partitioning Goals
process
memory
procedure
algorithmic
add
add
multiply
operation
Min. # of processors
Max. concurrency w/ multiple processors
Min. interprocessor communication
Satisfying packaging constraints
Min. cost w/ cheaper technologies (e.g. software)
Min. # of FU's
Max. concurrency w/ multiple FU's
Reducing synthesis tools' computation and memory
Satisfying packaging constraints
Simplifying floorplanning and routing
Specification partitioning is at this level
Previous work is at this level
Min. = minimizing, Max. = maximizing

Figure 1: Abstraction levels vs. goals

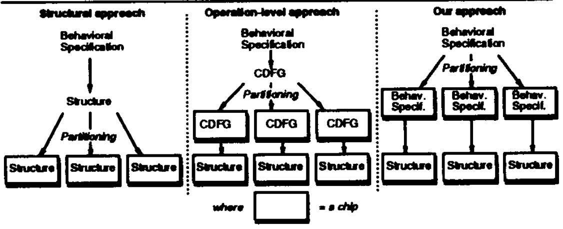


Figure 2: Specification partitioning vs. other methods

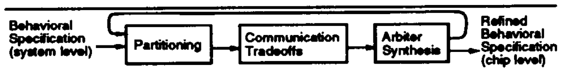

Behavioral Specification (system level)
Partitioning
Communication Tradeoffs
Arbiter Synthesis
Refined Behavioral Specification (chip level)

Figure 3: Iterative specification-refinement methodology

do operation-level goals.

Previous approaches to behavioral partitioning have been at the operation level; see [1] for a detailed survey of these approaches. In discussing these approaches, we shall use the term "CDFG" to denote any graph-type representation of behavior consisting of control and dataflow operations. Such representations are used internally by most synthesis tools.

YSC [2, 3] partitions logic containing behavioral operations, such as addition and subtraction, so that logic-synthesis runtime and memory-use constraints are not exceeded and so that follow-up floorplanning quality is improved. BUD [4] partitions the dataflow-graph operations of a CDFG in a manner that encapsulates scheduling and allocation decisions. Various partitionings are evaluated with respect to estimations of area and time. A similar approach is found in [5]. In Aparty [6, 7], the partitioning algorithm is extended to allow for the use of differing closeness functions and evaluation criteria at each one of multiple partitioning stages. CHOP [8] provides area and time estimates of an acyclic dataflow graph's operations for a given chip partitioning. Vulcan [9] partitions CDFG vertices, each of which represents a finite-state machine or combinational block, among chips.

As chip capacities as well as behavior-description sizes increase, we feel that algorithmic-level rather than operation-level partitioning approaches will be necessary. In this paper, we introduce an algorithmic-level approach (see Figure 2). **The main contribution of this research is the elevation of partitioning to a higher level of abstraction, i.e. grouping entire computations as opposed to merely operations.** In our specification partitioning approach, a behavioral description 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 partitioned such that each partition represents a chip. To find a partitioning that satisfies constraints, we must be able to estimate values for pins, area, and performance for a given partitioning, thus requiring estimation models. The results of partitioning are reflected in a refined specification consisting of a set of interconnected chips, each possessing behaviors and storage elements.

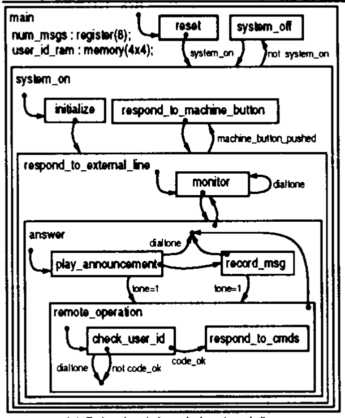
Specification partitioning has several advantages. There are far fewer algorithmic-level objects than operation-level objects. In addition, because partitioning is performed on the specification itself, most of the objects are familiar to the designer. The first fact permits the application of more thorough partitioning algorithms. The two facts combined enable ***designer control*** over the partitioning decisions. Also, ***functional test patterns*** can be written for each chip to test intended behavior. If the most abstract description of a chip consists of thousands of gates or low-level control/data operations, such test patterns are difficult to obtain.

Another advantage is the localization of the effects of *late specification changes*. Each chip specification consists of entire portions of the original specification behaviors (e.g. processes), with little or no modification. If a process needs to be internally changed after structural design of the chips, only the affected chip need be changed, leaving the rest of the multi-chip system intact. In CDFG approaches, a single simple behavior may be spread over multiple chips.

Now let us consider the most important advantage, which relates to the ability to modify the partitioned specification. To discuss this, we must first describe the overall design task of which behavioral partitioning is actually a subtask. We distinguish between two specification levels: the ***system specification***, which denotes desired system behavior, and ***chip specifications***, which denote the division of the system into interconnected chip specifications. Conversion from the first to the second level is part of what we call **System Design**, and often results in modified or added behaviors. We view system design as the refinement of a specification through iterations of partitioning, communication tradeoffs (e.g. bus merging), and arbiter synthesis, among other tasks (see Figure 3).

In specification partitioning, it is ***easy to modify or add to the specification*** when chip information is obtained, since the specification is maintained near its original abstraction level. For example, a 16-bit parallel data transfer between two behaviors on separate chips could easily be converted to two 8-bit transfers or even to a serial transfer in order to reduce pins, simply by changing the internal details of the two behaviors involved. As another example, multiple data buses can be merged into a single bus, which may then require the addition of a bus arbiter process to resolve contending bus-transfers.

In contrast, CDFG approaches usually view a specification as unchangeable, thus limiting synthesis to tasks such as scheduling, binding and logic synthesis. For example, multiple data transfers can use a single bus only if there would not be contention; adding an arbiter is not consid-



(a) Behavioral description (partial)

```
check_user_id
     entered_code : memory(4x4);
     i : integer range 1 to 5;

  i := 1;
  while i <= 4 loop
    wait until button_tone;
    entered_code[i] := button;
    i := i + 1;
  end loop;

  if (entered_code[1]=user_id_ram[1])
   and
     ...
    (entered_code[4]=user_id_ram[4])
  then
    code_ok <= true;
  else
    code_ok <= false;
  end if;
```

(b) a leaf behavior

Figure 4: Answering machine example.

ered. The arbiter would have had to exist in the original specification, even though its necessity may only arise when trying to satisfy pin constraints *after* partitioning.

In the cases when the specification partitioning approach does not decompose a behavior into fine enough granularity to satisfy chip-capacity constraints, it can be followed by an operation-level or structural-level partitioning approach. However, we believe such cases will become less frequent as chip capacities continue to increase.

3 Method

3.1 The behavioral specification

We specify behavior with the SpecCharts language [10], which allows behavior to be decomposed into concurrent sub-behaviors, sequential sub-behaviors sequenced by arcs, or VHDL sequential statements. SpecCharts is intended as a system specification language rather than a hardware description language (HDL); its main advantages over HDLs are the support of behavioral hierarchy and a unified representation of control, function, and structure. These features ease accurate capture and aid comprehension of the refined specification. Since SpecCharts subsumes behavioral VHDL, our approach is also applicable to VHDL.

An example of a behavioral specification is the telephone answering machine shown in Figure 4(a). It consists of a hierarchy of behaviors which respond to a phone line such that the machine can answer the phone, play an announcement, record a message, or be programmed remotely after a user identification number is checked. The code for the *check_user_id* behavior is shown in Figure 4(b).

We now describe the main steps of specification partitioning.

3.2 Object determination

Behavioral objects are all SpecChart behaviors and procedures (for VHDL, they would simply be all processes and procedures). Using VHDL-like declarations, memory storage is any variable or signal of array type where the elements are scalars or bit-vectors. Register storage is any other VHDL signal of register-kind or any global variable.

The example in Figure 4(a) shows 14 behaviors (e.g. *main, reset, system_on, system_off*, etc.) and 3 storage elements (*num_msgs*, *user_id_ram* and *entered_code*).

3.3 Estimation model creation

Estimation models are needed in order to estimate pins, area, and performance for a given partitioning of objects. The estimates are used by the partitioning algorithm to evaluate various partitionings.

Pin model: The model for pins is the simplest of the three. We use a hypergraph model where each vertex represents an object and each hyperedge represents communication between objects. There are several types of hyperedges:

- *storage access*: A hyperedge connects each storage with all behaviors which globally access (read and/or write) it. The weight of the hyperedge equals the register bitwidth or memory address/data bitwidth plus some control lines depending on the nature of the access.
- *behavior control*: A hyperedge connects each behavior with the parent behaviors that control it (e.g. calling procedures). Its weight is the sum of the bitwidths of any parameters, plus two control lines representing activation/completion control.
- *global-wire/external-port access*: For each non-storage signal or external port, a hyperedge connects it to all accessing behaviors. Its weight is the signal/port bitwidth.

Given a hypergraph, the number of pins for a partition is calculated as the sum of the weights of all hyperedges which cross the partition's boundary (i.e. the partition's cutsize).

Area model: Given a partition containing behavior and storage objects, we must estimate an area for the partition. This task is far more difficult than that for pins. An accurate area-estimation approach would perform synthesis for each partition, mapping sequential behaviors to a single control-unit and datapath (CU/DP). Unfortunately, such synthesis is computationally expensive, so performing it for each of the thousands of possible partitionings explored by a partitioning algorithm is infeasible.

An alternative is to assign a fixed area to each behavior and storage object, and then estimate a partition's area as the sum of its objects' areas. For each storage, area is determined by querying a component library. For each behavior, we synthesize a CU/DP and then use the resulting area. The advantage of this approach is that it only performs the computationally-expensive synthesis once per behavior, before partitioning begins. However, estimated partition area may be inaccurate since eventual implementation may not actually use a separate CU/DP for each behavior. The inaccuracy of this estimation approach is small if most behaviors are concurrent (i.e. processes) and thus will actually be implemented as separate CU/DPs, or if functional-unit sharing between sequential behaviors (which will occur if the behaviors use the same CU/DP) does not substantially affect overall area for a partition.

To minimize the error related to functional-unit sharing as well as reducing partitioning computation-time, only a subset of the specification's behaviors can be selected for

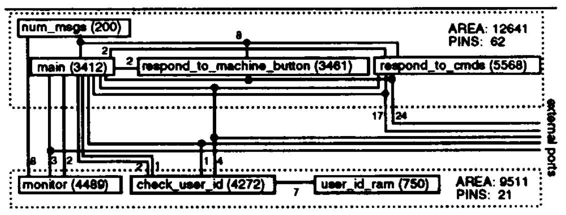


Figure 5: Partitioned hypergraph for answering machine

partitioning. By default, each remaining behavior is then encompassed by its parent behavior in the hierarchy.

We shall now discuss the synthesis technique used to create a CU/DP and estimate its area, although the technique is somewhat independent of our partitioning approach. We use the tool described in [11] to perform scheduling, allocation, and binding. We provide a basic set of functional units (e.g. an adder, a subtractor, and so on) and restrict synthesis to use only one functional unit of each type when building a datapath. This restriction could easily be changed to allow multiple small functional units but only one of each large functional-unit type (e.g. a multiplier) if such a change would more closely match the structural design strategy that will eventually be used. Once the structure is synthesized, the datapath area is determined by creating a bit-slice style floorplan of the muxes, registers, and functional units comprising the datapath, estimating wire-routing area, and then computing a bounding box. The control-unit area is determined from the state-register size and the number of gates.

Once each object has been assigned an area value, we map these values to the vertices of the hypergraph model already created for the pin model, so that standard hypergraph partitioning algorithms can be used. Figure 5 shows the hypergraph created for the answering machine example, along with a sample partitioning showing pin and area estimates. Areas are shown with each object, and interconnect-widths with each hyperedge. Note that many objects in the specification are not in the hypergraph; those objects are encompassed through hierarchy by objects in the hypergraph.

Performance model: Any behavior can have its average start-to-finish execution-time constrained. Execution time for a behavior can be modeled as the sum of two parts: (1) time spent communicating with other behaviors or storage over communication channels, and (2) time spent performing computations. The computation time is estimated by simulating the scheduled behavior with a representative data set, with all communication times over channels set to zero. This simulation also provides the average number of times that the behavior communicates over each channel during a single pass through the behavior; this number is multiplied by the latency of a channel's communication protocol to obtain total communication time over each channel. A communication is either a storage access or a sub-behavior access (activation/completion) plus the sub-behavior's execution-time. Default access protocols with given latencies are used for each type of access. Each channel actually has two protocol-latency values, one for accessing the object when it is on-chip, and one for the case when it is off-chip.

A given partitioning determines whether the on-chip or off-chip latency is to be used for each channel. The expected execution-time for a behavior is then the computation time plus the total communication time of its channels, which in turn are calculated using the appropriate latencies. We extend the hypergraph such that each vertex has a computation time. We add edges to represent channels between vertices, along with associated on-chip/off-chip communication times.

3.4 Partitioning

Let each hypergraph vertex be denoted as v_j, each channel edge by $c_{j,k}$ and each partition as V_i. We currently use the following straightforward objective function, which attempts to minimize constraint violations, to evaluate a given partitioning:

$$OBJFCT = k_1 \sum_i \left(100 \times \frac{excessarea(V_i)}{maxarea(V_i)}\right)^2$$

$$+k_2 \sum_i \left(100 \times \frac{excesspins(V_i)}{maxpins(V_i)}\right)^2 + k_3 \left(100 \times \frac{excesschips}{maxchips}\right)^2$$

An excess value (e.g. $excessarea(V_i)$) is equal to the estimated value (e.g. $area(V_i)$) minus the imposed constraint (e.g. $maxarea(V_i)$), with negative values being treated as 0. Thus, any partitionings that meet all constraints are considered equal since $OBJFCT$ will evaluate to 0. The k's are user-defined constants which indicate the relative importance of each metric. To favor balanced over unbalanced excesses, we multiply each term by 100 and then square. Other objective functions can also be used.

To extend $OBJFCT$ to consider performance, we add a fourth term:

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

$excessexectime(v_j)$	$= exectime(v_j) - maxexectime(v_j)$, or 0 if negative,
$exectime(v_j)$	$= commtime(v_j) + comptime(v_j)$,
$comptime(v_j)$	the expected time to execute a single pass of the behavior associated with v_j assuming communication times are 0,
$commtime(v_j)$	$= \sum_k commdelay(c_{j,k})$ for all v_k with which v_j communicates
$commdelay(c_{j,k})$	$= offchipdelay(c_{j,k})$ if v_j and v_k are not in the same partition V_i, $onchipdelay(c_{j,k})$ otherwise (recall that these delays incorporate expected frequency of channel use). If v_k is a behavior, it's execution-time equation is added.

Note that this term involves v_j, not V_i. Specifically, excess execution time is determined per vertex, not per partition. In other words, although execution time is affected by the chip partitioning, time-constraints are specified for behaviors (v_j), not partitions (V_i).

Well-known partitioning algorithms such as clustering and group migration are then applied. We currently use a clustering algorithm similar to that in [4], modified to use a closeness function defined for behaviors rather than operations. Evaluation of clusterings is done with the above objective function, rather than an area/time function. Our group migration algorithm is similar to the Kernighan/Lin algorithm discussed in [12], except that each move involves one object, not two. Also, an object can be moved to any one of multiple partitions, as opposed to just the opposite partition in bipartitioning. The objective function is that given above.

3.5 Refining the specification

The partitioning results are used to refine the original specification with chip information. Each chip will contain the behaviors and storage partitioned into it. If a storage is accessed by a behavior on a separate chip, that storage is converted to a process and all accesses are made via ports. Likewise, if a behavior is controlled by another behavior on a separate chip, control must be added between these behaviors. To aid designer comprehension, refinement should make minimal changes from the original specification. Each chip specification will serve as the behavioral input to a subsequent structural design phase, which may be done by a high-level synthesis tool.

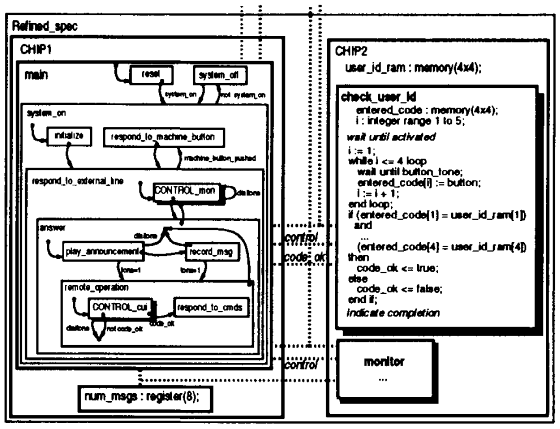
Assume a function $anc\_obj(v_i)$ returns the vertex whose behavior is the closest ancestor to the behavior of v_i in the specification hierarchy. The function $accessing\_objs(v_i)$ returns the set of vertices which access the storage associated with v_i. We use the following algorithm to refine the original specification with the partitioning information:

Algorithm 3.1 : Specification refinement

```
for each vertex $v_i$ representing a behavior
    if $anc\_obj(v_i)$ is on a different chip than $v_i$
        convert $v_i$'s behavior to a concurrent behavior $b_i$
for each vertex $v_i$ representing storage
    if any of $accessing\_objs(v_i)$ is on a different chip
        convert $v_i$'s storage to a concurrent behavior $b_i$
for each chip-partition $V_j$
    create a concurrent behavior $B_j$ representing a chip
for each conc. beh. $b_i$ (created from a $v_i$ above)
    move $b_i$ to a conc. subbehavior of $B_j$ such that $v_i \in V_j$
```

As an example, let us assume that the best partitioning in terms of area and pins has *check_user_id, monitor*, and *user_id_ram* on one chip, and all other objects on another chip, The refined specification resulting from this partitioning is shown in Figure 6.

Using ports to model storage access or subbehavior control unnecessarily models communication at a wire level. Communication should be modeled at a higher level in order to maintain information regarding the port groupings and the protocols over those ports. Therefore, whenever possible, we use the constructs of *protocols and channels* (see [10]) rather than ports. A protocol is defined as a set of ports (e.g. address and data) and a behavior over those ports. A channel is simply an instantiation of an associated protocol's ports. A channel is connected to other channels just as ports are, and the associated behavior is activated similar to a procedure call. These constructs separate communication from computation, making specifications readable and modifiable. Communication tradeoffs can be made later by merging channels, merging ports of a single channel, or associating a different protocol with a channel. All storage access and sub-behavior control in the refined specification uses channels with default protocols.



```
check_user_id
    entered_code : memory(4x4);
    i : integer range 1 to 5;
  wait until activated
  i := 1;
  while i <= 4 loop
    wait until button_tone;
    entered_code(i) := button;
    i := i + 1;
  end loop;
  if (entered_code(1) = user_id_ram(1))
   and
     ...
    (entered_code(4) = user_id_ram(4))
  then
    code_ok <= true;
  else
    code_ok <= false;
  end if;
  indicate completion
```

Figure 6: Refined specification

4 Results

SpecPart is a tool we are developing for specification partitioning. Its input is a specification in the SpecCharts language. Object selection, manual and automated partitioning, and partitioning evaluation are all aided by an X-based graphical interface. SpecPart currently consists of approximately 16,000 lines of C code.

Figure 7 summarizes partitioning results on several examples. All area units are generic; a particular technology would result in values in square mils or equivalent gates/transistors. The second column provides information on the size and characteristics of the specification, i.e. the number of specification lines, the number of behavioral hierarchical levels, and the number of behaviors and storage elements that could be selected for partitioning. Refined specifications were created for all partitionings.

We briefly discuss each example; for more details, see [13]. Rockwell's DRACO interfaces 16 I/O ports to a microprocessor's 8-bit multiplexed address/data bus, with extensive error and security checking. Two partitionings are shown with differing pin constraints. In the latter case, communication tradeoffs multiplex communication with 11 registers over a single bus, reducing pin counts from (134,124) to (58,48). A different merging resulted in (79,69) pins but had less effect on performance. The refined specification for one partitioning had 460 lines.

The telephone answering machine is the complete model from which the example in this paper was derived. By selecting several approximately equal-sized behaviors from the hierarchy, balanced two-way and three-way partitions are obtainable, as shown. The refined specification for the two-way partitioning had 1200 lines.

The Intel 8237 is a DMA controller. We tried three different selections of behaviors to obtain differing granularities for partitioning, as shown in the table. For the third case, pins can be reduced from (163,153) to (81,71) by merging the buses used by a particular behavior to access several registers on the opposite chip. Since the particular behavior (the program mode of the 8237) is only performed occasionally, the decrease in speed caused by these merges is of little consequence. For the fourth case, we imposed

Example	# spec. lines, hier. levels, poss. objects	#chips	area const.	pin const.	Area	Pins	comments
DRACO	302,4,19	1	--	--	15200	55	
		2	8000	100	5287	107	
					9933	108	
		2	8000	150	6677	134	comm. tradeoffs reduce pins to 58 and 48.
					8523	124	
Answering machine	726,10,59	1	--	--	29002	39	
		2	15000	50	14649	52	
					14254	46	
		2	15000	30	11076	25	
					17296	43	
		3	10000	40	10313	26	
					10459	45	
					8131	46	
I8237	697,5,54	1	--	--	38980	51	
		2	15000	80	9425	67	coarse objects were chosen
					31804	114	
		2	15000	80	26781	94	area balance sacrificed for performance
					10907	47	
		2	15000	150	19105	163	comm. tradeoffs reduce pins to 81 and 71.
					18029	153	
		2	15000	50	28535	76	
					8509	29	
AST-like bus controller	464,3,10	1	--	--	2455	130	
		2	5000	100	1620	88	model was respecified into control and data
					803	58	

Figure 7: Results of partitioning several examples

a time-constraint on a particular behavior (the transfer mode). In this case, several register files which were previously placed on a separate chip, were grouped with the constrained behavior to reduce communication time.

We also modeled a subset of protocols handled by an AST bus controller which interfaces 32-bit address and data buses with two 8-bit buses. To obtain a partitioning which met pin constraints, the original model had to be rewritten as two concurrent behaviors, one implementing the controlling state machine, the other the bus routing.

The above examples represent only the lower end of the domain for which SpecPart is intended. We are currently working on a much larger industrial example, consisting of a RISC signal processor, timers, several memories, and multiple I/O behaviors. We believe that the fact that such extensive functionality will be implemented on a single chip (with approximately one million transistors) lends support to the need for algorithmic-level partitioning approaches. We plan to examine not only multichip implementations, but also to extract partitions which lead to better single chip designs by using multiple custom processors, as discussed in the introduction.

5 Future work

We are currently developing a method for estimating area far more accurately than discussed in this paper, but in a manner which still permits constant-time updates of the estimate when an object is moved among partitions.

Currently, our refinement algorithm converts sequential behaviors which are separated from their parent into concurrent processes. However, if the behaviors on a chip are sequential, they can be merged into a single process in order to encourage more efficient implementation as a single CU/DP.

As mentioned in the introduction, attacking other problems such as hardware/software partitioning, or partitioning among multiple processors to extract concurrency (as discussed for the RISC example above) involves changes in the constraints, estimations, and objective functions. We plan to make these changes to broaden the scope of the specification partitioning approach.

If a satisfactory partitioning is not found, respecifying the behavior using more procedures can provide finer granularity. Automated techniques to isolate and convert candidate code portions should be developed.

We also plan to develop tools for generating and matching communication protocols, for performing communication tradeoffs, for binding memories, and for synthesizing arbiters to resolve bus contention.

6 Conclusion

We have introduced a partitioning approach that is the first to address behavioral partitioning at a process/procedural level as opposed to a dataflow-operation level. We have shown how the approach can be used to solve the specific problem of obtaining multichip partitions. As behavioral description sizes continue to increase, there are many other behavioral partitioning problems that must be addressed. We feel that building on the concepts outlined in this paper should lead to promising solutions to these problems.

7 Acknowledgements

We would like thank Sanjiv Narayan for his work on estimation which enabled implementation of our partitioner, and for his numerous other contributions. This research was supported by the National Science Foundation (grant #MIP-8922851) and the Semiconductor Research Corporation (grant #90-DJ-146); we are grateful for their support.

References

[1] F. Vahid, "A Survey of Behavioral-Level Partitioning Systems." UC Irvine, TR ICS 91-71, 1991.

[2] R. Camposano and R. Brayton, "Partitioning Before Logic Synthesis," in *Proc. ICCAD*, 1987.

[3] R. Camposano and J. van Eijndhoven, "Partitioning a Design in Structural Synthesis," in *Proc. ICCD*, 1987.

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

[5] J. Rajan and D. Thomas, "Synthesis by Delayed Binding of Decisions," in *Proc. 22nd DAC*, 1985.

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

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

[8] K. Kucukcakar and A. Parker, "CHOP: A Constraint-Driven System-Level Partitioner," in *Proc. 28th DAC*, 1991.

[9] R. Gupta and G. Micheli, "Partitioning of Functional Models of Synchronous Digital Systems," in *Proc. ICCAD*, 1990.

[10] S. Narayan, F. Vahid, and D. Gajski, "System Specification and Synthesis with the SpecCharts Language," in *Proc. ICCAD*, 1991.

[11] S. Narayan and D. Gajski, "Area and Performance Estimation from System-Level Specifications." UC Irvine, TR ICS 92-16, 1992.

[12] B. Preas and M. Lorenzetti, *Physical Design Automation of VLSI Systems*. California: Benjamin/Cummings, 1988.

[13] S. Narayan, F. Vahid, and D. Gajski, "Modeling with SpecCharts." UC Irvine, TR ICS 90-20, 1990.