Title
Constant-time cost evaluation for behavioral partitioning

Authors
Vahid, Frank
Narayan, Sanjiv
Gajski, Daniel D.

Publication Date
1992-03-19

Peer reviewed

Constant-Time Cost Evaluation for Behavioral Partitioning

Frank Vahid
Sanjiv Narayan
Daniel D. Gajski

Technical Report #92-29
March 19, 1992

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

narayan@ics.uci.edu
vahid@ics.uci.edu

Abstract

Given a system behavioral specification, partitioning can be used to distribute among chips the processes, procedures, and storage elements that comprise the specification. We introduce a technique for constant-time recomputation of pin, area, and execution-time estimates for a behavioral partitioning move. The technique permits fast, accurate estimations of a large number of partitionings, thus enabling better results than approaches which attain tractable computation time by using gross estimates or less thorough partitioning algorithms. The key to our technique is the isolation and extraction before partitioning of the basic design attributes needed for estimation, and the updating of this information in constant-time for each move. The estimation models are almost as detailed as those presented in previous estimation approaches not intended for constant-time update. The results we provide indicate the speed and practicality of our estimation approach in conjunction with sophisticated partitioning algorithms.

Contents

List of Figures

1 Introduction

A behavioral description can be partitioned into sub-descriptions such that each partition is to be implemented: (a) in a particular technology (custom layout, ASIC, software), (b) as a chip such that chip-capacity constraints are met, or (c) using a single controller. These goals, as well as the advantages of behavioral vs. structural partitioning, are discussed in [1].

Any partitioning approach uses an algorithm to select possible partitionings, and an objective function to evaluate each possible partitioning. Sophisticated partitioning algorithms which obtain good results, such as simulated annealing, tend to select many possible partitionings for evaluation. Objective functions which yield accurate evaluations for behavioral partitioning use estimation models that approximate high-level synthesis. Unfortunately, such estimation usually requires much computation time, so using both a sophisticated partitioning algorithm and an accurate evaluation method is usually computationally prohibitive. For example, assume a behavioral description is decomposed into n behavioral objects, and the partitioning algorithm generates n^2 possible partitionings of these objects, while the evaluation time for each possible partitioning is proportional to n^3. The overall computation time is proportional to the product of the number of possible partitionings and the evaluation time for each partitioning, i.e. $n^2 \times n^3 = n^5$.

In an attempt to obtain good results while staying computationally feasible, an approach may either: (1) use sophisticated partitioning algorithms, thereby forcing use of gross estimates which can be computed quickly, or (2) use accurate estimation methods, thereby forcing the use of simple partitioning algorithms (see Figure 1). Both approaches may lead to unsatisfactory results. In (1), gross estimates may result in chip constraint violations or in underutilized chips when the structure is eventually synthesized. In (2), only a small number of partitionings and hence a small portion of the complete solution space is examined, making it likely that a poor overall solution is selected.

However, we have developed a technique to obtain both fast *and* accurate estimations for behavioral partitioning for goal (b) stated above. The key observation is that many partitioning algorithms make a small constant number of object moves to generate a new partitioning from an existing one. Our approach is thus to perform a (computationally expensive) detailed estimation only once, and then to incrementally update the estimation in constant-time for an object move. The details of the technique can be summarized as follows. We roughly synthesize a structural implementation for an initial partitioning, while maintaining design attribute information for each behavior and storage object; these attributes indicate the contribution of each object to the structure. Pin, area, and execution-time estimates are then computed from the structure. When an object is moved, we use that object's associated design attributes to incrementally modify the existing structure. The estimates are then recomputed.

Incremental update techniques are well-known for structural partitioning ([2, 3]), but are far more difficult for behavioral partitioning. For example, the fact that multiple procedures can be implemented with a single controller and datapath implies that a "sum of object areas" model of a partition's area, which is used in structural partitioning, is not sufficient at the behavioral level. The same fact implies that an object interconnect model used to determine a partition's pins, which is trivially obtained from structural objects, is not obvious from behavioral objects.

The report is organized as follows. In Section 2, we formulate the behavioral partitioning problem that we will address. In Sections 3, 4 and 5 we provide behavioral-level estimation models for pins, area, and execution-time, respectively. For each model, we define the design attributes that form the basis of estimates, we demonstrate how to obtain an estimate from an initial partitioning, and we then show how to incrementally update that estimate in constant-time for a single move. In Section 6, we extend our approach for more general behaviors than assumed in the previous sections. In Section 7 we provide results which demonstrate the speed of our estimation approach.

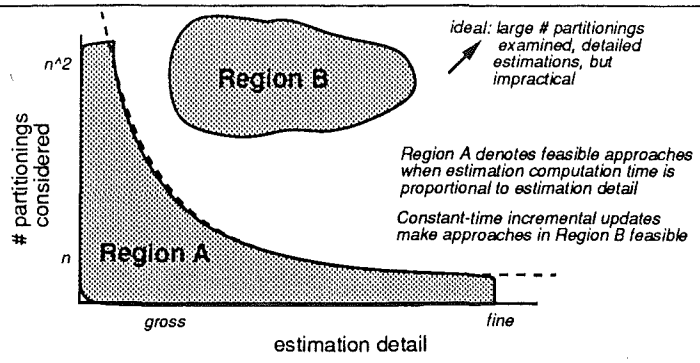


Figure 1: Constant-time estimation permits better results

2 Problem Formulation

2.1 Input Behavior

The input behavior is described using VHDL [4] or SpecCharts [5]. Currently we shall assume the behavior is a set of VHDL concurrent processes; we describe later in the report extensions necessary for the hierarchical behaviors found in SpecCharts. A process is essentially a main procedure whose sequential statements (e.g. assignments, branches, and procedure calls) are enclosed in an infinite loop. We shall thus commonly refer to a process as a procedure. Execution of the input behavior simply consists of calling all main procedures concurrently. A main procedure may contain declarations of data items and sub-procedures.

VHDL signals are declared global to processes. Unlike variables, signals possess a value defined over time, so signals can be used to model concurrent assignments, and can be accessed by multiple concurrent processes.

An example input behavior is shown in Figure 2(a), which describes an interface between a CPU's 8-bit bus and a peripheral's 16-bit bus, with checksum used for error checking. This simple example is *far smaller* than the behaviors for which our partitioning approach is intended, and serves *for illustrative purposes only.* We shall refer to this example throughout the report.

2.2 Behavioral Partitioning Level

One must decide the granularity level at which the input behavior can be decomposed. We distinguish between two levels: (see Figure 3):

- Operation-level: the objects grouped into partitions are dataflow operations such as addition, comparison, and multiplication. Most previous approaches are operation-level [6, 7, 8, 9].
- Algorithmic-level: the objects are program-grained computations such as procedures, and storage. As chip capacities increase, we feel that the likelihood that a behavior's operations need be partitioned among chips will decrease, which may necessitate algorithmic-level approaches.

In this report, behavioral partitioning refers to the algorithmic-level. We view the input behavior as a set of procedures and global storage, as shown in Figure 2(b). (This view is in contrast with operation-level partitioning in which behavior is viewed as a control/dataflow graph). Global storage is defined as any VHDL signal of register-kind, any global variable, or any variable requiring a two-dimensional memory implementation.

Each procedure represents a computation; the computation is complete upon return. Note that a VHDL procedure does not necessarily obey this model, because it may schedule a signal update

```
ports (CMD : in bit_vector(0 to 2); AD_BUS : in byte;
       ERROR : out bit; OUTL, OUTH, CHSUM : out byte);
signal lsb,msb : byte register;

Main : process
  variable checksum : byte;
  variable addrlatch : bv(0 to 1);
       <procedure declarations>
begin
  if CMD="001" then
    Address(AD_BUS);
  elsif CMD="010" then
    Read();
  elsif CMD="011" then
    Write(addrlatch,AD_BUS);
  end if;
  wait on CMD;
end process;

procedure Address(addr : in byte) is
begin
  addrlatch := addr;
  if addrlatch > "01" then
    ERROR <= '0';
  end if;
end;

procedure Read() is
begin
  OUTL <= lsb;
  OUTH <= msb;
  checksum := lsb + msb;
  CHSUM <= checksum;
end;

procedure Write(addr: in bv(0 to 1);
                data: in byte) is
begin
  case addr is
    when "00" => lsb <= data;
    when "01" => msb <= data;
    when "10" =>
      checksum := data;
      if (lsb+msb /= checksum) then
        ERROR <= '1';
      end if;
    when "11" => ...
  end case;
end;
```

(a) Behavioral description

(b) View of the behavior for partitioning

Figure 2: An example VHDL behavior to be partitioned

that occurs sometime *after* returning from the procedure. For example, consider the following VHDL description:

```
signal ir,pc : integer;
main : process

  procedure Fetch(M : Mem) is
  begin
    ir <= M(pc) after 10 ns;
  end;

  procedure IncPC is
  begin
    pc <= pc + 1 after 10 ns;
  end;

begin  --main
  wait until clock='1';
  IncPC;
  Fetch(M1);
end process;
```

On each rising *clock*, the main process calls *Fetch* and *IncPC*. The procedures schedule new values for *pc* and *ir*, but neither update occurs until 10 ns have passed. Hence, the updates take place *concurrently* 10 ns *after* the rising clock edge. (Note that reversing the calls to *IncPC* and *Fetch* would still yield identical results).

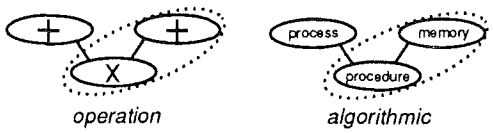


Figure 3: Behavioral partitioning abstraction levels

In order to match our view of a procedure performing a computation which is complete upon return, scheduling a transaction to occur after return from a procedure, as in the example above, is *not permitted* in the input behavior. Procedures which do not match our completion requirement must be expanded inline.

Also, we do not permit recursive procedure calls since mapping such procedures to hardware is difficult.

2.3 Target Architecture

When a procedure p is separated off-chip from a caller, we convert its formal parameters into global signals and create two signals for the procedure, *p.start* and *p.done*. An off-chip calling procedure first assigns values to global signals representing *in* direction parameters, asserts the *p.start* signal, waits until *p.done* is asserted, and then reads any global signals representing *out* direction parameters.

We assume that any subset of procedures from the same process is always implemented on a single control-unit and datapath (**CU/DP**). A datapath contains functional-units (e.g. adders, comparators), registers, memories, multiplexors and buses, whereas the control-unit contains a state-register and combinational logic for generating the next-state and for controlling the datapath components.

If a process' main procedure is not part of the subset, then a *virtual-main* procedure is implicit. *Virtual-main* handles all requests made by an off-chip procedure for starting an on-chip procedure, i.e. it waits for the assertion of any *p.start* signal, calls p, and then asserts the *p.done* signal.

Figure 4 illustrates the above concepts. A single *main* process is shown consisting of five procedures, $p1$ through $p5$. A call-graph is shown, where a solid directed arc from node a to node b represents the fact that procedure a calls procedure b. A sample chip-partitioning is shown, along with the details of the *virtual-main* procedure needed for $p3, p4, p5$. The parameters for $p3$ and $p4$ are converted to global signals. For example, if $p4$ has one formal parameter A, and $p1$ calls $p4(X)$, then A is converted to a signal $p4.A$, and the call of $p4(X)$ is converted to: $p4.A <= X; p4()$; All calls to $p4$ are then modified to:

$p4.start <= true;$
$waituntilp4.done;$
$p4.start <= false;$

Each *global* storage element is treated as its own sequential behavior, and thus is implemented with a CU/DP where the CU is empty and the DP contains one register or memory. The CU/DP's that implement procedures have registers and memories to implement the *local* storage needs of the procedures.

Figure 5 summarizes the mapping of procedures to CU/DP's. Note that each chip may have more than one CU/DP, and that a single process may be implemented with multiple CU/DP's if divided among chips.

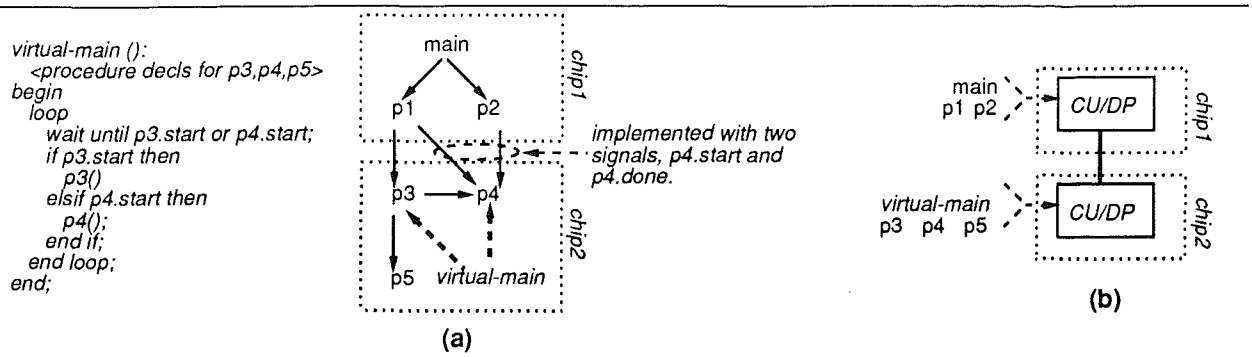


Figure 4: Separating procedures from the main procedure

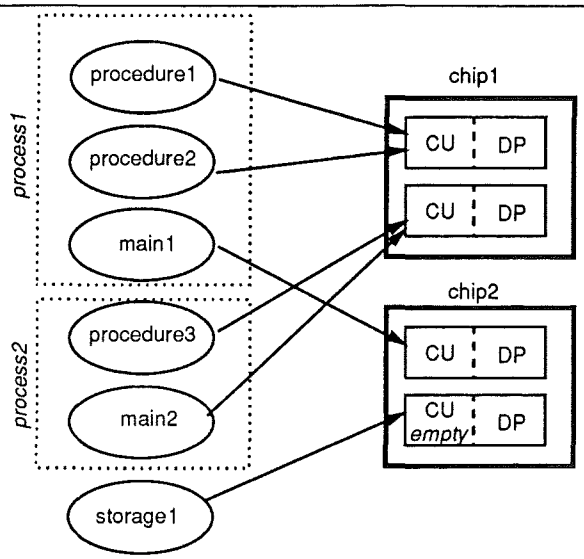


Figure 5: Each sequential behavior on each chip uses one CU/DP

2.4 Computational Complexity

Constant-time complexity is relative to the number of procedures and storage elements which must be partitioned. It is *not* relative to the number of operations (e.g. addition, multiplication) in a procedure; the number of operations is bound by a constant since there are a finite number of language operators available. In addition, only the incremental update is constant-time, not the initial preprocessing required before partitioning. Finally, the data-structure implementation is such that searching for an item in a set of items is essentially constant-time. Such a search is possible using a hash table.

In the next three sections, we shall discuss the pin, area, and execution-time models used in our approach. For simplicity, we assume the behavior is a *single* process; we describe the extensions for more general behaviors in Section 6. Given the set of procedures and storage that make up the partitioning objects, an initial partitioning can be achieved by any constructive algorithm, such as by clustering or by random allocation. Given the initial partitioning of objects among chips, we must first estimate values for pins per chip, area per chip, and execution-time per sequential behavior. We must then update these values when an object is moved by an iterative-improvement partitioning algorithm. Obtaining and updating such values requires models for interconnect, area, and execution-time. The next three sections discuss each of these models individually. In each section, we first discuss the various levels of detail at which estimation can be performed, and then indicate which

level our approach is at. Second, we discuss the model of interconnect, area, or execution-time that we use. Third, we discuss the information which we preprocess given the initial partitioning, and show how this information is used to compute values for initial estimates. Lastly, we demonstrate how the information changes when an object is moved during iterative partitioning, and how this change affects the value of the estimate. We also show that these changes require only constant-time computation.

3 Pins

3.1 Estimation Detail

An extremely accurate method of estimating pins requires that storage elements be bound to real storage so that the number of ports and control lines are known, considers the number of pins needed for power and ground, and determines precisely what control lines are needed for a CU/DP to activate a procedure implemented in an off-chip CU/DP. Currently we ignore power and ground, assume one type of register and one type of memory are available with known control lines, and assume procedure activation is implemented as a simple handshake.

3.2 Interconnect Model

A common model used for estimating pins is a hypergraph with ports. Our task is to obtain a hypergraph model for the procedures and storage that make up the behavior. Once the model is obtained, well-known hypergraph techniques are used to estimate the number of pins per partition and to incrementally update those estimates for a vertex move.

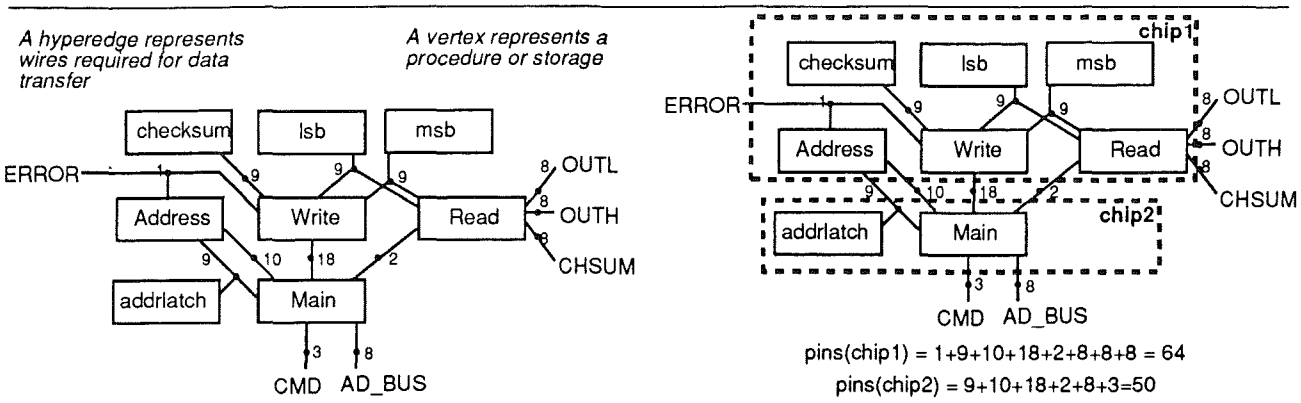
3.3 Preprocessed Information

We first convert the set of procedures and storage to a hypergraph model, as in Figure 6(a). We map each storage and procedure to its own vertex. There are several types of hyperedges:

- *storage access*: A hyperedge connects each storage with all procedures which access (read and/or write) the storage. Such accesses are often called global accesses. The weight of such a hyperedge equals the register bitwidth or memory address/data bitwidth, plus 1 or 2 control lines, respectively.
- *procedure call*: A hyperedge connects each procedure with all procedures that call it. Its weight is the sum of the bitwidths of the procedure's parameters, plus 2 control lines for activation/completion.
- *global-wire/external-port access*: For each non-storage signal, a hyperedge connects all procedures that access this signal. Its weight is the signal bitwidth.

Given a hypergraph, a partition's pins is determined as the sum of the weights of all hyperedges which cross the partition's boundary (i.e. the partition's cutsize), as shown in Figure 6(b).

Note that we assume two control lines for off-chip procedure activation: one to initiate the procedure and one to detect completion. Actual synthesis may result in a different number of lines. For example, if N procedures on a chip are called from off-chip, only $log(N+1)$ lines are needed to activate a procedure, and only one line may be needed to indicate completion. Such details are difficult to estimate.



(a) Interconnect information for pins

(b) Pin counts for initial partitioning

(c) Incrementally updating pin counts

Figure 6: Hypergraph for determining pins

3.4 Constant-time Incremental Updates

Given a vertex move, we must incrementally update the cutsizes of the partitions involved, which is done by determining if each neighboring hyperedge contributes to the cutsize before and after the move. The update technique is well known, so we won't cover it here. See [3] for details. A simple example is shown in Figure 6(c).

4 Area

4.1 Estimation Detail

Area can be estimated at varying levels of detail. At one extreme, gross estimations can be made by providing an area for each procedure and storage (perhaps by assuming a single CU/DP implementation of each procedure), and then computing a partition's area as a sum of areas. This approach has the advantage of enabling constant-time incremental updates, but may be inaccurate since it does not reflect the single CU/DP model.

At the other extreme, detailed estimations can be made by synthesizing the layout for a given partition. This includes performing tasks for each partition such as scheduling, module-selection and allocation from a full library of components with consideration of area and time constraints, controller optimization, bus optimization, register-sharing through life-time analysis, floorplanning, routing, and bounding box calculation. Although this approach is accurate, it is computationally expensive and hence infeasible for estimation purposes.

We take a middle-ground approach which obtains estimations using tasks such as scheduling, allocation from a basic library, unoptimized controller synthesis, approximations to multiplexor requirements, and rough wiring estimates. Specifically, we quickly synthesize a structure with the goal of approximating the eventual high-quality synthesized structure *while maintaining an incremental update ability.* The level of detail of our estimation approach is approximately the same as approaches discussed in [6, 7, 10] which do not consider incremental updates.

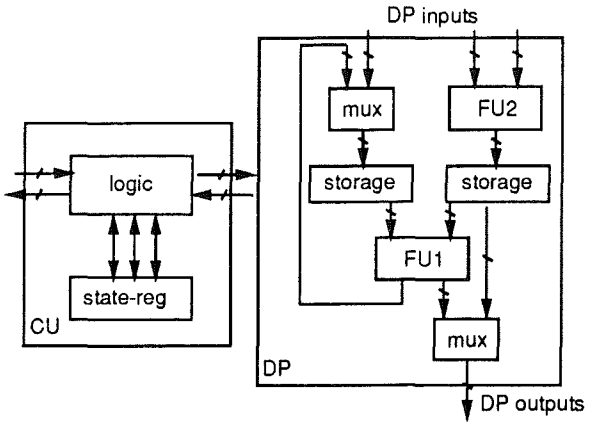


Figure 7: CU/DP area model

area factor			is a function of
CU/DP	CU	state_reg	# states
		+	
		logic	# states, # ctrl_lines, # states each ctrl_line is active
	+		
	DP	storage	# bits and # words of each storage
		+	
		func_units	# bits and type of each FU
		+	
		muxes	# sources of each storage or FU input, or DP output port
		+	
		wires	# DP connections, # DP components

Figure 8: Equation and terms for computing CU/DP area

4.2 CU/DP Area Model

We use a CU/DP area model similar to that presented in [10] (see Figure 7). The total CU/DP area is computed as shown in Figure 8 as the sum of several basic terms:

- *Functional-unit (FU) area*: determined by the operations performed in the behavior. Allocation determines a set of FU's which can implement all operations of the behavior. We shall discuss allocation later. An FU's area is a function of the FU's type and bitwidth.
- *Storage-unit area*: determined by the storage necessary for holding values for more than one state, possibly necessitated by allocation and scheduling. A storage-unit's area is a function of the unit's bitwidth and number of words (for memories).
- *Mux area*: If a storage-unit input, FU input, or DP output (which we shall refer to as *destinations*) is written by more than one source, a multiplexor is required. A *source* is a storage-unit output, FU output, or DP input. The mux size is a function of the number of sources, and the mux area is a function of the mux size.
- *State-register area*: a function (i.e. log) of the number of possible states of the scheduled behavior.
- *Control-logic area*: a function of the number of states, the number of DP control lines, and the number of states each DP control line is active. The logic is assumed to consist of a set of AND gates which detect a state and possibly DP conditions, and a set of OR gates, one for each DP control line. Further details concerning the CU logic can be found in [10].

The schedule indicates the states that each destination is actively written. If the destination is a storage-unit, these states correspond to the active-states of the storage-unit control-line. If the destination requires a mux, the states correspond to the active states for the mux control lines.

- *Wiring-area*: We differ slightly from wiring as determined in [10] in that we don't do placement of DP components. Instead, we approximate that each DP connection will require a wire who's length is some fraction of the number of DP components. This approximation will enable constant-time update, as we shall see.

Note that all of the above terms are a function of some basic attributes: number of possible states, number of control lines, number of states each control line is active, number and size of each storage unit, number of size of each type of FU, number of sources for each storage-unit/FU/DP-port, and the number of DP connections.

4.3 Preprocessed Information

Recall that the preprocessed pin information included a hypergraph along with partition cutsizes. The area information is far more complex.

After creating a symbol table providing bitwidths of all data symbols, the following is done on *each procedure individually*:

1. *CDFG creation and transformations*: converts the behavior to a control/dataflow graph and transforms the graph to a maximally parallel control/dataflow behavior. CDFG representations are discussed in [11], and transformations are discussed in [12]. The main transformations used are combination of selects and unrolling of loops. Ideally the algorithm for transformation application will match the algorithm that will eventually be used to create structure.

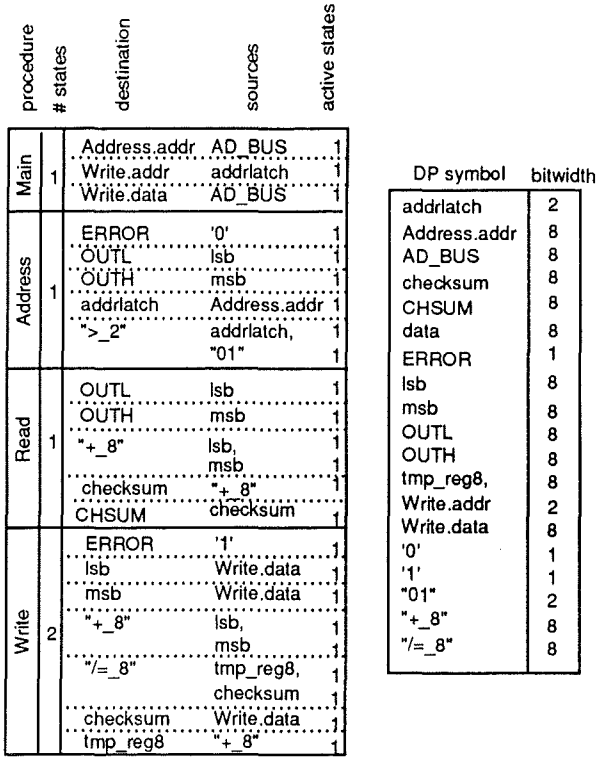
2. *Allocation and binding*: determines the types, number, and bitwidths of FU's needed by the procedure. Allocation and binding will bind each operation in the procedure to a functional-unit that can implement that operation. For example, all additions and subtractions may be bound to a single ALU. On the other hand, each addition may be bound to its own adder, and each subtraction to its own subtractor. There may be many types of adders used, some fast and large, others slow but small.

Ideally the allocation and binding algorithms used at this stage would be similar to the algorithms that will eventually be used to create structure. If the specific algorithms are not known, we can use simple heuristics.

One heuristic assumes a simple library that permits a many-to-one mapping of behavioral operations to FU's. In other words, a given behavioral operation such as addition can be implemented by exactly one type of FU. An example library might include a single FU for each of the following operations: $=, <, >, <=, >=, /=, -, +, /, *, **, rem, mod, abs$ (using VHDL notation).

Given the simple library, one allocation heuristic allocates no more than one FU of each required type, and binds each operation to the single FU which can implement that operation. Conversely, we can allocate as many FU's as there are operations, and bind each operation to its own FU. A third possibility is a mix of the previous two heuristics in which we allocate as many FU's as there are operations if the FU is smaller than some particular area, otherwise we allocate no more than one FU of that type. For example, a behavior could use many adders but only one multiplier, since typically the former are small while the latter are large.

Even if the eventual allocation and binding algorithms are known and used during preprocessing, some error may still occur. Recall that CDFG creation and transformation were done for each procedure individually. After partitioning, when all the procedures on a chip are known, a single CDFG for all those procedures can be formed. Transformations applied to this CDFG may move operations



procedure	# states	destination	sources	active states
Main	1	Address.addr	AD_BUS	1
		Write.addr	addrlatch	1
		Write.data	AD_BUS	1
Address	1	ERROR	'0'	1
		OUTL	lsb	1
		OUTH	msb	1
		addrlatch	Address.addr	1
		">_2"	addrlatch,	1
			"01"	1
Read	1	OUTL	lsb	1
		OUTH	msb	1
		"+_8"	lsb,	1
			msb	1
		checksum	"+_8"	1
		CHSUM	checksum	1
Write	2	ERROR	'1'	1
		lsb	Write.data	1
		msb	Write.data	1
		"+_8"	lsb,	1
			msb	1
		"/=_8"	tmp_reg8,	1
			checksum	1
		checksum	Write.data	1
		tmp_reg8	"+_8"	1

DP symbol	bitwidth
addrlatch	2
Address.addr	8
AD_BUS	8
checksum	8
CHSUM	8
data	8
ERROR	1
lsb	8
msb	8
OUTL	8
OUTH	8
tmp_reg8,	8
Write.addr	2
Write.data	8
'0'	1
'1'	1
"01"	2
"+_8"	8
"/=_8"	8

Figure 9: Basic information for area

across former procedure boundaries, and may therefore lead to a different scheduling. Second, a good allocation algorithm will consider area constraints. Since we are allocating before partitioning, such constraints are not known and hence cannot be incorporated.

3. *Scheduling*: creates a finite-state machine controller for the datapath components such that the desired behavior is executed. At this point, we ignore procedure calls by assuming they require zero time (the reason for this will be clear later). We refer to the number of possible states resulting from such scheduling as the number of *internal* states of the procedure. Scheduling also determines the temporary storage-units needed to hold values across states. Registers of equal bitwidths can be shared by multiple states. Ideally the scheduling algorithm will match that eventually used to create structure. Otherwise, we can use a default algorithm such as list scheduling.

4. *Determination of sources*: is done for each destination (FU/storage-unit input or DP output) written to in the procedure. We also record the number of states in the schedule that each source is actively assigned to its destination.

Figure 9 shows the information created for each procedure in the example in Figure 2.

Given an initial partitioning, we use the above area information to determine the CU/DP area terms (as discussed in Section 4.2) for each set of procedures on a chip as follows:

Functional-units (*fct(#,types,bitwidths)*): Since FU's can be shared among procedures due to their sequentiality, the number for each FU type of a specific bitwidth needed by the chip is simply the maximum needed by any one procedure.

Storage-units (*fct(#,bitwidths)*): Storage units of equal bitwidth are shared among procedures. Hence the number of each storage-unit is the maximum needed by any one procedure.

Muxes (*fct(# sources/destination)*): Multiple procedures may contribute the same or different sources to the same destination (FU/storage-unit input or DP output). The sources of each of a chip's

	destination	sources	contributing procedures	component required	area	control lines	active states
DP output ports	ERROR	'0'	Address	2x1 mux	20	1	2
		'1'	Write				
	OUTL	lsb	Address, Read				
	OUTH	msb	Address, Read				
	CHSUM	checksum	Read				
	addrlatch	Address.addr	Address				
	lsb	Write.data	Write				
	msb	Write.data	Write				
FU's	">_2"	Address.addr	Address	2-bit ">"	30		
		"01"	Address				
	"+_8"	lsb	Read, Write	8-bit "+"	200		
		msb	Read, Write				
	"/=_8"	tmp_reg	Write	8-bit "/="	150		
		checksum	Write				
Storage-units	checksum	Write.data	Write	8-bit 2x1 mux	160	1	2
		"+_8"	Read	8-bit reg.	240	1	2
	tmp_reg8	"+_8"	Write	8-bit reg.	240	1	1

Computing Chip1's area:

FU area = 30 + 200 + 150 = 380
storage area = 240 + 240 = 480
mux area = 160 + 20 = 180
states = 1 + 1 + 2 = 4
statereg area = fct(log(3)) = 60
ctrl logic area = fct(#states, control lines, active states) = 400
wires = # sources in table = 17
DP comps = # entries in "component required" column = 7
wiring area = fct(#wires,#DP comps) = 400
CU/DP area = 380 + 480 + ...
chip area = area(CU/DP) + area(checksum) + area(lsb) + area(msb)

Figure 10: Additional area information for initial partitioning

destinations are determined as the *union* of those sources contributed by each procedure. Given the number of sources, we determine the mux size and control lines.

State-register (fct(# possible states)): We compute the number of possible states as the sum of the number of possible states of all procedures in the partition. Such computation may not be quite accurate. The reason is that a procedure may be called from more than one location. If the procedure is inline expanded during eventual implementation then the number of states it contributes to the partition should be multiplied by the number of locations where it will be inlined. We have yet to determine a constant-time method to update the number of states computed in such a manner.

Control-logic (fct(# states, # ctrllines, # activestates/ctrlline)): The number of control lines is the sum of the number of control lines needed by each storage-unit and mux. The number of states which each chip's source is active is determined as the sum of the number of states for which the source is active in each procedure. This in turn determines the number of states each control line is active.

Wiring-area (fct(# DP connections, # DP components): The number of DP connections is the total number of sources on the chip. The number of DP components is the number of FU's, storage-units, and muxes on the chip.

From the above, we see that values for the basic terms for the area equation in Figure 8 have been determined, so the chip area can now be computed. A partial example is shown in Figure 10.

Computational complexity of preprocessing

Relative to the number n of procedures and storage, the complexity of preprocessing for area information is $O(n)$. The preprocessing complexity is usually dominated by the scheduling algorithm, whose complexity may range from $O(c^2 log(c))$ to $O(c^3)$, where there are c nodes in the behavior's dataflow graph; c is usually large compared to n.

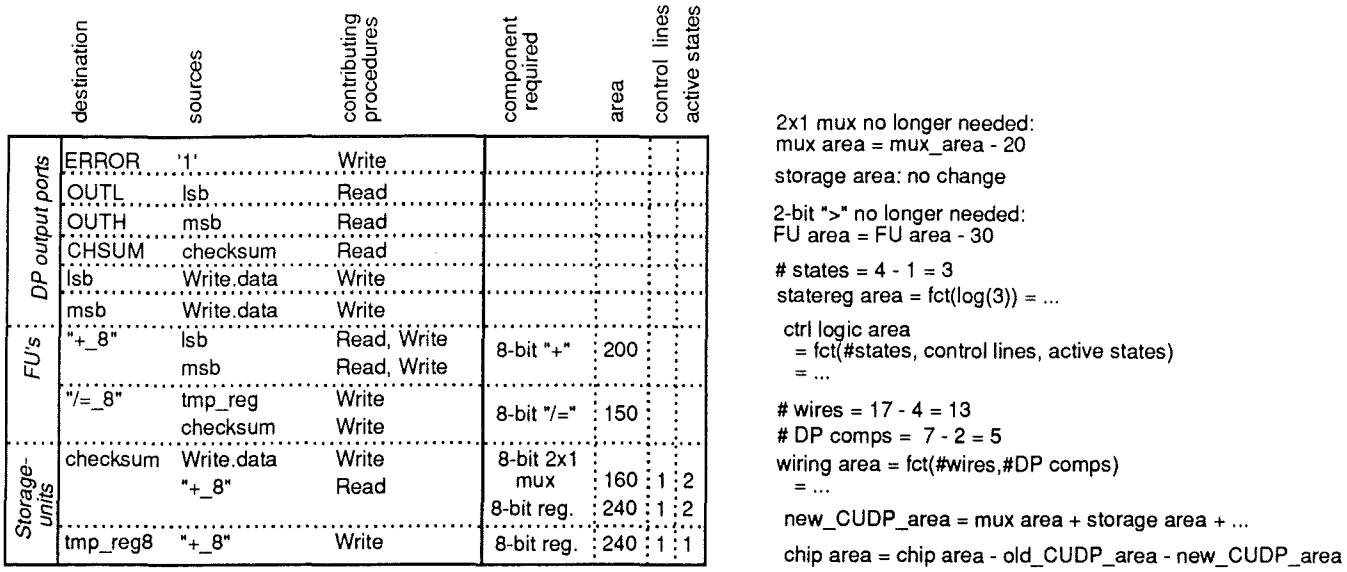


Figure 11: Chip1's CU/DP area update when Address moved to Chip2

Given the preprocessed area information, the complexity of computing the initial area estimate for a given partitioning is $O(n)$.

4.4 Constant-time Incremental Updates

A move of an object i consists of deleting i from one partition and adding i to another; Figure 11 provides an example. When deleting an object i from partition P, the following steps are taken:

1. For each destination in P, all sources existing solely due to i are deleted. Such deletion may affect the mux size and mux control lines used for the destination. P's mux-area and control lines are decreased accordingly. The active states for remaining control lines are decremented by the number of active states for the deleted sources. If a mux is no longer needed, the number of P's DP components is decremented accordingly.

2. After deleting sources existing solely due to i, a destination may have no remaining sources. The destination is thus deleted from P. If it was an FU, P's area is decremented by the FU area. If it was a storage-unit, P's storage-unit area is decremented by the unit's area, and P's control lines are decreased accordingly.

3. P's number of possible states is decremented by p's number of possible states.

4. P's area equation is recomputed with the updated values.

When adding an object i to partition P:

1. Any destination in i but not in P is added to P. If an added destination is an FU or storage-unit, P's area is incremented by the unit's area. If it was a storage-unit, P control lines are increased accordingly.

2. For each i destination, the sources are *unioned* with the corresponding P destination's sources. This union may add sources to a destination and hence may affect the mux size and mux control lines used for the destination. P's mux-area and control lines are increased accordingly. The active states for the control lines are incremented by the number of active states for the added sources.

3. P's number of possible states is incremented by p's number of possible states.

4. P's area equation is recomputed with the updated values.

All changes outlined above can be done in constant-time, as long as the data-structure used ensures that the on-chip destinations to which p contributes sources are directly accessible without search. A key assumption is that the number of destinations in a given procedure is independent of the number of procedures/storage in the overall behavioral description. This assumption fails when each procedure calls every other procedure and accesses every global storage. Since procedures serve to modularize a description, such a situation is highly unlikely. Instead, each procedure will likely access a small (constant) number of other procedures and global storage, in addition to its parameters.

5 Execution-Time

5.1 Estimation Detail

We must determine the average execution time for a sequential behavior to execute from beginning to end. At one extreme, estimation may ignore branching and simply sum operator delays along the longest acyclic path through the scheduled behavior, adding a constant whenever the path crosses a chip boundary. While being simple, this approach is highly inaccurate.

At the other extreme, estimation may consider average execution frequencies of each branch in the behavior, and may synthesize the layout for each partition. The expected execution time is then computed using the execution frequencies, register-to-register delays which include operator, mux, wire, setup and hold times, and off-chip delays. Again, full synthesis is infeasible for estimation.

Our approach uses average execution frequencies along with approximated models of register-to-register delays and on-chip/off-chip access times.

5.2 Sequential Behavior Execution-time Model

For each procedure, we first perform the following steps: CDFG creation and transformations, allocation and binding, and scheduling, as was done during area preprocessing. Each node of the scheduled CFG will contain a set of basic operations (e.g. $a := M(pc) + 1$, pc:= pc+1;), or a procedure call.

The expected execution frequencies of each node of the scheduled CDFG are obtained by performing simulation with a representative set of test vectors. Given the expected number of executions N_i of each node i during execution of the behavior from beginning to end, the expected execution time of the behavior is: $\sum_i N_i t_i$, where t_i is the execution time of node i. We view t_i as being the sum of two terms. The first term is computation time for the node's operations, being either register-to-register delay as determined in [10], or called-procedure execution time as determined by recursively repeating this algorithm on the called procedure. The second term is communication time for accessing external storage or for accessing (activating) called procedures. The access time of external storage differs if the storage is on-chip or off-chip. For procedures, an off-chip procedure call requires a handshake. Values for on-chip and off-chip access times are looked up for storage and procedures.

5.3 Preprocessed Information

We form a single equation for the expected execution-time of a sequential behavior by expanding the above summation, providing both computation and communication terms for each t_i. The computation times are constants so are summed to a single constant. The remaining terms are access times which

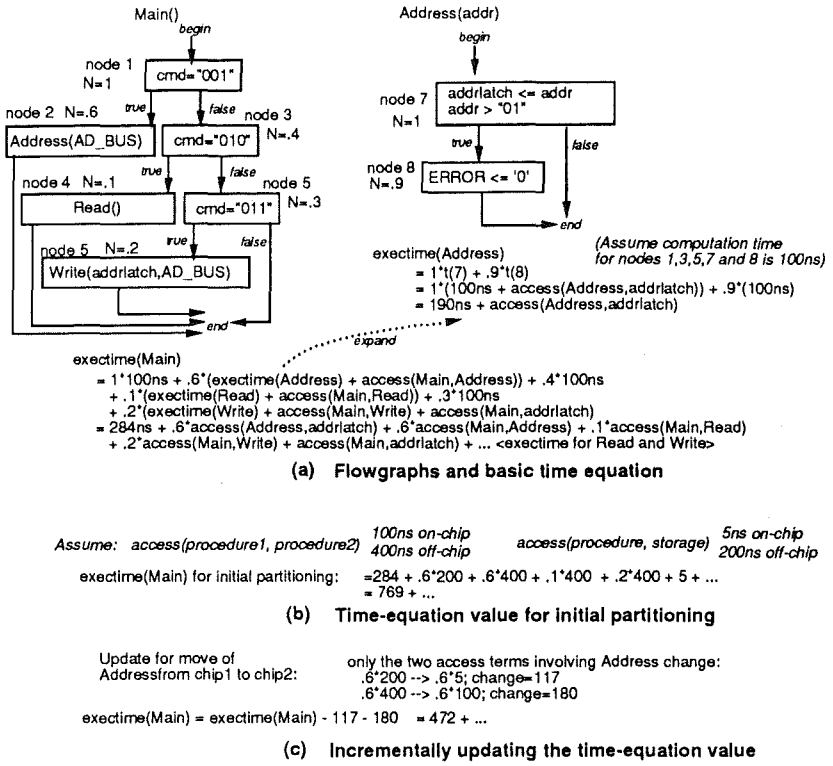


Figure 12: Execution-time equation creation, evaluation, and update

may be one of two values depending on whether or not two specific objects are in the same partition. The two objects of each access are the accessor (a procedure) and the accessee (a procedure or storage). See Figure 12(a) for an example. For a given partitioning, we select the on-chip or off-chip value for each access term, and then sum all terms to obtain the behavior's execution-time. In Figure 12(b), execution-time is computed assuming the initial partition shown in Figure 6(b).

A separate equation is formed for each procedure which is constrained. Note that a constrained procedure may itself be called by another constrained procedure, allowing multiple levels of time constraints.

5.4 Constant-time Incremental Updates

Given a move of a procedure or storage o from partition $C1$ to partition $C2$, we simply need update terms which involve o for the new partitioning. For each term involving o which switches from the on-chip to the off-chip value or vice-versa, the overall execution time is simply incremented or decremented by the change in term value. Figure 12(c) provides an example.

To ensure constant-time updates, the data structure must allow those terms related to a single procedure or storage to be accessed directly, without search. An assumption is made, as for area, that each procedure does not access every other procedure and global storage, but instead a small number in addition to its parameters.

example	# partitioning objects	# lines code	time for pre-processing information	# moves made by partitioning algorithm	time for moves	avg. time per move
intel8237	36	692	22.77	13324	107.31	**.008**
ans_mach	49	726	27.16	9983	63.23	**.006**
draco	15	302	12.59	816	4.94	**.006**

Figure 13: Results show the speed of our technique

6 Extensions for Hierarchy and Concurrency

We have introduced our technique using a behavior consisting of only of set of a single process' procedures and storage. To extend the technique for a hierarchy of behaviors sequenced by arcs, as in [5], we simply treat each behavior at each hierarchical level as a procedure without parameters. A behavior activating a subbehavior acts like a procedure calling a subprocedure.

We also extend the technique for a behavior consisting of a set of concurrent process, each possibly containing a set of procedures, as allowed in VHDL. Pin and time information are developed in same manner as before. For each chip, we keep separate area information for each process. When a procedure is moved, on each chip we only update the area information for the relevant process; hence the addition of processes has no effect on the constant-time characteristics of cost-evaluation. Note that there may be more than one CU/DP on a single chip if procedures from different processes exist on the same chip; the areas for these are simply summed.

Finally, we extend the approach to permit concurrent behaviors at any level of the behavior hierarchy as in [5], not just at the top level as with processes, by simply converting such concurrent behaviors to processes activated using handshake control mechanism.

7 Results and Future Work

We have implemented the constant-time updatable estimator, and have incorporated it with a behavioral partitioning tool. The input is a SpecChart behavioral description, and the output a refined description containing chip-partitioning detail. Implementation consists of approximately 6,000 lines of C code. CDFG transformations have not been implemented.

Having shown that constant-time estimation updates are possible, we now demonstrate that the constant is acceptably small for practical use. We applied our partitioning tool to many examples, including the Intel 8237, a telephone answering machine, and the DRACO peripheral interface (see [13] for details of these examples), the results of which are shown in Figure 13. To provide a notion for the size of each example, we indicate the number of behaviors and storage to be partitioned along with the number of lines of code in the behavioral description. For each example, we first applied random constructive partitioning to obtain an initial 2-way or 3-way partitioning, and measured the time to obtain the preprocessed information. We then applied the group-migration algorithm [14, 2, 3] extended for multi-way partitions to improve the partitioning, using the cost function specified in [1]. Shown in the figure is the number of moves which the algorithm performed, along with the time (in seconds) required to update the estimation information for these moves. Note the extremely fast time-per-move shown in the rightmost column. These times are clearly fast enough for practical use.

To see the necessity for fast constant-time updates, consider estimating area for each possible partitioning using the traditional approach. Applying an estimator ([10]) to the Intel8237 example requires approximately 2.75 seconds for a given partition, *excluding* the initial scheduling. If used with the partitioning algorithm discussed above, total computation time would have exceeded 35,000

seconds, rather than the 107 seconds required using the constant-time update technique.

True constant-time updates require a hash-table for set lookups. Our lookups are currently linear; hence times should be *even faster*. Execution times were measured using Unix's 'gprof' execution profiling tool [15].

Future work includes further improving the estimation accuracy. It may be possible to incorporate floorplanning information, which can then be used for more accurate wire length and bounding box calculations.

8 Conclusions

We have demonstrated how in constant-time we can update fairly detailed estimations of pin, area, and time for a move of a behavioral object during behavioral partitioning. The foundation of our technique consists of the maintenance of estimation models which can be incrementally updated when an object is moved. Our technique permits use both of good partitioning algorithms and of detailed estimations, rather than just one or the other, and therefore provides for greatly improved behavioral partitioning results.

9 Acknowledgements

10 References

[1] F. Vahid and D. Gajski, "Specification Partitioning for System Design," in *Proc. 29th DAC*, 1992.

[2] B. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell System Technical Journal*, February 1970.

[3] C. Fiduccia and R. Mattheyses, "A Linear-Time Heuristic for Improving Network Partitions," in *Proc. 19th DAC*, 1982.

[4] *IEEE Standard VHDL Language Reference Manual*, 1988.

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

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

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

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

[11] A. Orailoglu and D. Gajski, "Flow Graph Representation," in *Proc. of the 23rd Design Automation Conference*, 1986.

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

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

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

[15] S.L. Graham, et. al., "gprof: A Call Graph Execution Profiler," in *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, 1982.

A Appendix

A.1 Hypergraph cutsize computation

In this section we provide the algorithms for determining a hypergraph partition's cutsize, and for incrementally updating that cutsize in constant-time for a behavioral object move.

We define a hypergraph as a set V of vertices v_i and a set E of hyperedges $e_{j,k,...}$ connecting two or more vertices v_j, v_k,..... Each hyperedge has a width. Any hyperedge may be an external port. A partition P_l is a subset of vertices.

To determine the cutsize of a partition, we simply sum the widths of all hyperedges which connect a vertex in the partition with something outside of the partition. Assume a function $Conn(e_{j,k,...})$ returns the set of vertices $\{v_j, v_k, ...\}$, and a function $ExtPort(e)$ returns true if e is an external port. The cutsize of partition P is determined as follows:

Algorithm A.1 : ComputePartitionCutsize(V, E, P)

$cutsize = 0$
for each e in E
  if ($ExtPort(e)$) or ($\exists v_i, v_j \in Conn(e), v_i \in P, v_j \notin P$)
    $cutsize = cutsize + width(e)$
  end if
end for
return ($cutsize$)

To update a partition's cutsize when a vertex is moved to or from a partition, for each hyperedge e incident to this vertex we must determine if it either:

- (1)
  - (a) Did not contribute to the cutsize before the move
  - (b) Will contribute after the move
- (2)
  - (a) Contributed to the cutsize before the move
  - (b) Will not contribute after the move

For the source partition, case 1(a) occurs when all connected vertices are in the partition and the hyperedge is not an external port. Case 1(b) then occurs if the hyperedge connects more than one vertex. Case 2(a) occurs when the hyperedge is an external port or it connects a vertex in another partition. Case 2(b) then occurs when the vertex being moved is the only vertex in the source partition that is incident to e. A similar discussion applies to the destination partition.

We define the following functions: $IncidentHyperedges(v)$ which returns the set of hyperedges which involve v. $NIP(e)$ (NumIncidentPartitions) returns the number of partitions containing vertices that e connects. $NCVIP(e, P)$ (NumConnectedVerticesInPartition) returns the number of vertices in partition P which e connects. Assuming a move is made of vertex v from partition P_l to partition P_k, cutsizes are updated as follows:

Algorithm A.2 : UpdatePartitionCutsizeForMove(V, E, v, P_k, P_l)

$I = IncidentHyperedges(v)$
for each e in I
    /* first do the source partition P_k */
    if ($(NIP(e) = 1)$ and (e is not a port)) and $(NCVIP(e, P_k) > 1)$
        /* e did not contribute before, but will after the move */
        $cutsize(P_k) = cutsize(P_k) + width(e)$
    else if ($(NIP(e) > 1)$ or (e is a port)) and $(NCVIP(e, P_k) = 1)$
        /* e contributed before, but will not after the move */
        $cutsize(P_k) = cutsize(P_k) - width(e)$
    end if

    /* now do the destination partition P_l */
    if $(NCVIP(e, P_l) = 0)$ and ($(NIP(e) > 1)$ or $(NCVIP(e, P_k) > 1)$ or (e is a port))
        /* e did not contribute before, but will after the move */
        $cutsize(P_l) = cutsize(P_l) + width(e)$
    else if $(NCVIP(e, P_l) > 0)$ and ($(NIP(e) = 2)$ and $(NCVIP(e, P_k) = 1)$ and (e is not a port))
        /* e contributed before, but will not after the move */
        $cutsize(P_l) = cutsize(P_l) - width(e)$
    end if

end for

A.2 Controller number of states

In order to maintain constant-time update ability, we made the simplifying assumption that the number of possible states of a controller for a set of procedures is equal to the sum of the number of possible states of each procedure. Such an assumption does not consider the fact that a procedure may be called from many places, and should therefore contribute in each place it is called. In this section, we show how a more detailed number of states computation can be made, at the expense of requiring linear time computation of this number for a given move of a behavioral object.

Before partitioning, we create an *call-graph*. It is a directed acyclic graph where each node n_i represents a procedure i and each directed edge $e_{i,j}$ represents the fact that procedure i calls procedure j. The weight of n_i is the number of internal states for procedure i, and the weight of $e_{i,j}$ is the static number of places in the code where procedure i calls procedure j. Figure 14(a) shows the call-graph for the example in the report; note that each edge and node has an associated weight.

For the initial partitioning, we wish to determine the number of states for a given chip's CU/DP. We first determine the number of possible states for each procedure which does not have a calling procedure on the chip (i.e. root procedures); procedures whose calling procedures are on-chip are treated as inlined into those calling procedures. The number of states for the controller that implements the set of procedures on a chip is computed as the sum of the number of states of all root procedures. We provide an algorithm below for determining the number of controller states for a partition P containing a set of procedures. Assume a function $NumCalls(i, j)$ returns the number of static occurences of calls to sub-procedure j from procedure i, and a function $InternalStates(i)$ returns the number of possible states of procedure i assuming sub-procedure calls require no time.

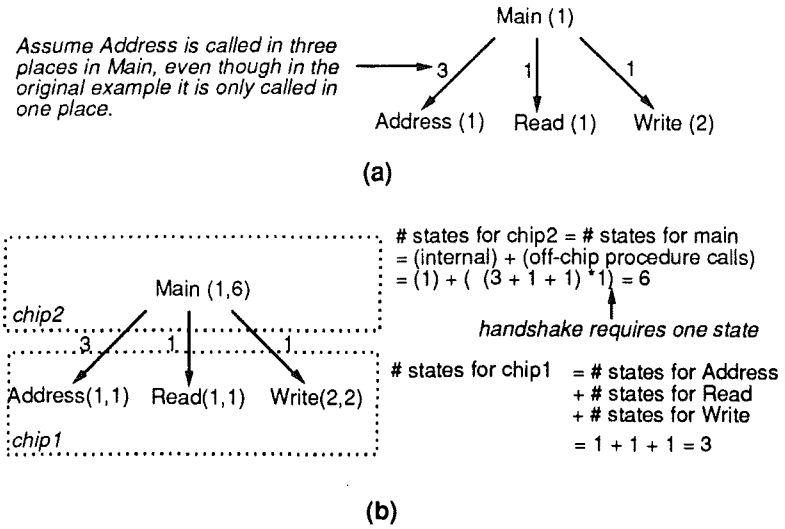


Figure 14: Call-graph for earlier example

Algorithm A.3 : ComputePossibleStates(P)

```
states = 0
for each n_i in P
    if no e_{h,i} exists such that n_h is in P
        states = states + ComputeProcedureStates(i)
    end if
end for
return (states)

ComputeProcedureStates(i, P)
    for each e_{i,j}
        if j is in P
            s = ComputeProcedureStates(j, P)
        else
            s = 1 (for handshake)
        end if
        states = states + s × weight(e_{i,j})
    end for
    states = states + InternalStates(i)
    return (states)
```

The first portion of the algorithm finds all root procedures in a partition, calls a function to compute the number of states for each such procedure, and sums these for the partition total. The function $ComputeProcedureStates$ is passed a procedure i and a partition. The number of states for each sub-procedure j called by i is determined, equal to the number of occurence of calls to j ($weight(e_{i,j})$ times the number of states per call, being 1 if j is off-chip for the handshake, and being the number of states of j, recursively determined, to account for the inlining of j.

As the number of possible states of a procedure k is determined, we append this number to the corresponding n_k in the call-graph, as shown in Figure 14(b).

We now must update the number of states when an object is moved. For each call-graph edge that crossed chips before the move, but is entirely within one chip after the move, we must update the

number of states of the calling procedure and all of its on-chip parents to reflect the inlining of the called procedure rather than handshaking. The opposite task is performed for each edge that is made to cross between chips due to the move. The algorithm below operates on the call-graph to update the number of possible states of partitions $P1$ and $P2$ for a move of procedure i from $P1$ to $P2$. P_x denotes the partition containing node n_x.

Algorithm A.4 : UpdateStates($i, P1, P2$)

```
for each e_{j,k} where j = i or k = i
    1: if P_j = P1 and P_k = P2, orvice − versa
        d = weight(e_{j,k}) × (1 − weight(n_j))
    2: elsif n_j and n_k are in P1
        d = weight(e_{j,k}) × (weight(n_j) − 1)
    end if
    weight(n_j) = weight(n_j) + d
    states(P_j) = states(P_j) + d
    3: if (P_j = P2)
        UpdateParentStates(j, d, P_j)
    end if
end for

UpdateParentStates(j, d, P)
    for each e_{h,j} where n_h is in P
        weight(n_h) = weight(n_h) + d × weight(e_{h,j})
        states(P) = states(P) + d × weight(e_{h,j})
    end for
```

Line 1 detects all edges that cross before the move but don't cross after. Line 2 detects edges which don't cross partitions before the move but will cross after. Line 3 determines if the calling procedure j does not move. In this case, edges may exist which point to j from a procedure on the same partition; these procedures inline j so must be updated by the change in j's number of states. $UpdateParentStates$ handles this updating.

The above algorithm is of linear complexity with respect to the number of procedures. The reason is that when a procedure is moved, in the worst case we will have to update the number of states for all ancestor procedures in the call-graph.