Constant-Time Cost Evaluation for Behavioral Partitioning

Frank Vahid
Daniel D. Gajski

Technical Report #92-29
March 19, 1992
Revised May 9, 1994

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

vahid@ics.uci.edu

Abstract

To aid in the functional partitioning of a system into interacting hardware and software components, fast yet accurate estimations of hardware size are crucial. We introduce a technique for obtaining such estimates in several orders of magnitude less time than previous approaches without sacrificing accuracy, by incrementally updating a design model for a changed partition rather than reestimating entirely. The technique eliminates the major shortcoming of functional partitioning tools in real design environments, namely that the estimators were either far too slow or grossly inaccurate. Our results demonstrate the striking speed with which we can obtain estimates which have the same accuracy of previous, much slower techniques.

Contents

List of Figures

1 Introduction

Combined hardware/software implementations are common in embedded systems. Software running on an existing processor is less expensive, more easily modifiable, and more quickly designable than an equivalent application-specific hardware implementation. However, hardware may provide better performance. A system designer's goal is to implement a system using a minimal amount of application-specific hardware, if any at all, to satisfy required performance. In other words, the designer attempts to partition the functionality such that as much functionality as possible is implemented in software.

The traditional approach to functional partitioning, which starts from an informal natural-language description, is not adequate for several reasons. First, since natural-language descriptions are not machine readable, we cannot automate estimation and partitioning. The quality of a partition is thus heavily influenced by the designer's experience. Second, we cannot perform early verification using the natural language, so functional errors are not detected until late in the design cycle, at which time they are difficult to correct. Third, since natural-language descriptions are usually ambiguous, it is difficult to integrate the resulting hardware and software components. The integration problem arises when a different designer implements each component. Each designer will likely interpret the description differently. Such different interpretations manifest themselves as errors when we integrate and test the components.

Executable-specification partitioning has evolved to address the above limitations. In this specific approach to functional partitioning, we first capture the system's functionality with an executable specification-language, which is machine-readable and simulatable, such as VHDL. We derive functional objects from this specification, such as procedures, statement blocks, statements, or operations, and then we partition them. Such approaches overcome the limitations described above.

The executable-specification partitioning process must be guided by quality metrics that indicate the "goodness" of any particular partition of functional objects among hardware and software components. Such metrics include hardware size (which we wish to minimize), execution-time, and channel bitrates. In this paper, we focus on the hardware-size metric.

There are several different ways to determine the hardware size for a given partition. The most accurate approach would be to synthesize a design for the partition, but such an approach requires too much time if we wish to examine more than a few possible partitions, as is usually the case. To overcome this limitation, several research efforts incorporate a hardware size estimator [1, 2, 3, 4]. In essence, such estimators roughly synthesize a design for the given functional objects, while omitting the time-consuming synthesis tasks such as logic optimization, so they require only a few seconds to obtain a fairly accurate estimate. Such **design-based** estimators are useful when we wish to examine tens of possible partitions, but are still too slow for examining hundreds or thousands of partitions generated by an iterative-improvement partitioning algorithm, such as simulated annealing.

Since we want to use iterative-improvement algorithms in order to obtain good hardware/software partitions, recent approaches which perform hardware/software partitioning have abandoned design-based hardware-size estimators in favor of **abstract-weight-based** estimators. In these estimators, an abstract weight is assigned to each functional object, and then a hardware "cost" for a given partition is obtained simply by adding all the weights of the functional objects mapped to hardware [5, 6]. These approaches gain speed at the expense of estimation accuracy.

In developing our hardware/software partitioning system, we desired to use design-based estimates in order to obtain accuracy, but we also wanted to use iterative-improvement algorithms. Since previous estimation approaches could not accomplish both goals, we needed to develop a new method for rapid design-based estimation. Towards this end, we observed that iterative-improvement algorithms make only a few changes between iterations, so the change between one partition's design and the next one is incremental, as illustrated in Figure 1. Thus, we developed design-based estimators that use a new data structure and algorithm that permit fast, incremental modification of the design, and hence obtain extremely fast estimates that are still as accurate as previous design-based approaches. Our contribution is this new method of estimation that achieves the advantages of *both* classes of previous approaches, namely accuracy and speed.

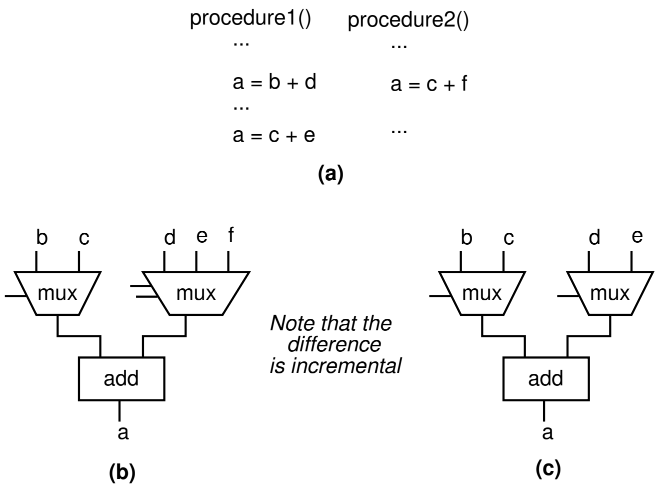


Figure 1: Example of incremental change resulting from object move: (a) behaviors, (b) datapath for both behaviors, (c) datapath after procedure2 is deleted.

This paper is organized as follows. In Section 2, we describe the design-model which we use for hardware-size estimation, a model adopted from previous design-based estimators. In Section 3, we describe a new data structure which captures not only the design, but also the contribution of each functional object to the design. In Section 4, we detail an algorithm for updating, in constant time, this data structure when a functional object is moved, either by an interative-improvement algorithm or manually. In Section 6, we summarize our results that show the speed of our method, and discuss future work.

2 Hardware-size estimation design model

In this section, we describe the design model used to obtain hardware-size estimates for a set of functional objects. We use a control-unit/datapath (CU/DP) design model similar to those presented in [1, 2, 7], illustrated in Figure 2. Note that we are not required to use this model in all cases; if a particular design environment uses a different model, then we can develop a data structure and an update algorithm in a manner similar to that described in the next sections. In other words, we are introducing a general method as it is applied to a particular model; the method can also be used for other models.

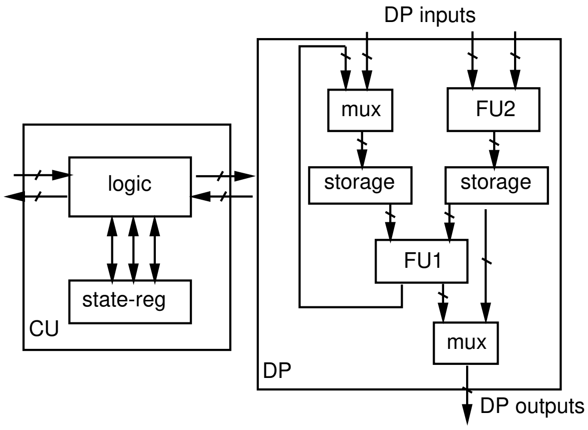


Figure 2: CU/DP area model

The total CU/DP size for the model is computed as the sum of the following basic terms: *Functional-unit and storage-unit size* (including registers, register files and memories), *Multiplexer size*, *State-register size*, *Control-logic size*, and *Wiring size*. Each of the above terms is a function of one or more of the following basic design parameters:

$states$	= the number of possible controller states,
$size\_list$	= a list containing the size of each unit in the datapath,
$srcs\_list$	= a list containing the number of possible inputs (sources) to each unit in the datapath,
$ctrl$	= the number of control lines between the controller and the datapath,
$active\_list$	= a list containing the number of states that each control line is active,
$units$	= the number of units in the datapath,
$wires$	= the number of wires in the datapath.

For example, the *functional-unit and storage-unit size* may be a function of all $size\_list$ values, the *multiplexer size* may be a function of $srcs\_list$, the *state-register size* may be a function of $states$, the *control-logic size* may be a function of $states$, $ctrl$, and each $active\_list$, and the *wiring size* may be a function of $units$, each $size\_list$, and $wires$. The details of these functions are beyond the scope of this paper, and are actually an orthogonal issue to the method described here. In fact, a particular implementation of our method should support a variety of such functions in order to handle estimations for a variety of technologies. Hence, we assume the existence of a hardware size estimation function which takes the above parameters as inputs; specifically, we assume the following function exists:

$$HwSize(states, size\_list, srcs\_list, ctrl, active\_list, units, wires)$$

though there may be more than one such function.

3 Incrementally-updatable data structure

We now consider the format of the data structure that will allow us to represent a roughly-synthesized design using the model described in the previous section, while at the same time allowing us to incrementally modify in constant time that design when a functional object is added or deleted. The behavior that is input to the partitioning system is assumed to consist of a single process (such as a VHDL process), consisting of hundreds or thousands of sequential statements that may include loops, branches, and procedure calls. We shall later describe a simple extension to support multiple processes.

3.1 Preprocessed information

After creating an initial partition of functional objects among hardware and software components, but before applying an iterative-improvement algorithm, we can build our initial data structure. We must first obtain a rough design for the given set of functional objects that have been mapped to hardware, since such a design is the basis for the estimates obtained by design-based estimators. To obtain this design, we must allocate a set of functional units (FUs) and storage units (SUs), bind operations and data values to FUs and SUs, and schedule operations into control steps (not necessarily in the given order).

We first perform scheduling, allocation, and binding to the entire set of functional objects. The algorithms that we use should match the algorithms that will be used to synthesize the final hardware, in order to obtain the highest accuracy; if the algorithms are not known, then we can use default algorithms instead. A traditional estimator would stop here, but we must proceed to determine how each functional object contributes to that design, so that we can incrementally modify that design when an object is moved.

Let us first consider how each object contributes to the datapath. For each functional object, we create a list of FUs that the object uses. For example, if a functional object contains an addition operation, then we append an adder unit to that object's FU list (or whatever unit is used to implement addition). If two adder units are used by that object, then we append two adders to the list. This FU list will be used later to inform us what units the functional object requires when it is moved to hardware. We create a similar list for SUs. Having determined how each functional object contributes to the datapath's FUs and SUs, we turn our attention to the multiplexers. The size of a multiplexer in front of an FU is determined by how many possible **sources** (i.e., SU or FU outputs, or datapath inputs) can be input to the FU. Likewise, the size of the multiplexer in front of an SU is determined by how many sources can be input to the unit. Multiplexers may also be needed in front of datapath outputs, as in Figure 2. Based on these factors that determine a multiplexer's size, we see that we must determine which sources each functional object inputs to each FU, SU and datapath output.

At this point, an assumption that we wish to make explicit is that a functional object represents a sequence of statements, meaning each represents either a block of statements, a function or a procedure. The larger the number of statements in each object, the more accurate the estimations will be, since inter-object optimizations would then play a smaller role in the overall design. This same granularity of objects has been assumed by other research efforts as well, such as in [5, 8, 9]. Lower levels of granularity not only would make estimation very difficult, but would also prevent designer comprehension of the objects and thus make interactive design nearly impossible, a limitation that would likely be unacceptable in real use.

Having obtained the above information for the datapath, we turn our attention to the control unit. We record the number of possible states for each functional object, since this information will tell use how many additional states must be represented by the state register when we move an object to hardware. We also record the number of states that each FU, SU and datapath output is active, since this number affects the complexity of the control logic.

Note that the information that we have obtained so far could have been obtained before an initial partition was created, since it deals with all of the functional objects without

fct. objs.	states	destination	sources	active states
Main	5	A	C	1
		B	'1'	1
procedure1	6	A	E	1
		comparator1	A D	1
		adder1	C D	1
procedure2	9	B	'0'	1
		adder1	C E	1
		tmp_reg1	adder1	1

Figure 3: Preprocessed information for hardware size (PP)

regard to the partition. Thus, we call this **preprocessed** information. Figure 3 shows the preprocessed information created for each procedure of the example in Figure 1(a). Note that this example is trivially small, but that it sufficiently demonstrates our technique.

We now define our data structure that maintains the preprocessed information. Based on the above discussion, we choose to organize the information as a list of **destinations**, where a destination is an FU, SU or datapath output, since each component in the datapath corresponds to a destination. The data structure consists of the following:

PP	$= \{O, DPI, DPO, U\}$ = the preprocessed information,	
O	$= \{o_1, o_2, ..., o_n\}$ = the set of functional objects,	
o_i	$= \{states, dsts\}$,	
$states$	$\in Natural,$ = the number of control states for the functional object,	
$dsts$	$= \{dst_1, dst_2, ...\}$ = the destinations written by the object,	
dst_i	$= \{id, srcs, active\}$,	
id	$\in DPO \bigcup U,$ = the particular FU, SU, or DP output that the dst represents,	
$active$	$\in Natural,$ = the number of states the destination is active for this object,	
$srcs$	$= \{src_1, src_2, ...\}$ = the sources that the object assigns to this destination,	
src_i	$\in DPI \bigcup U,$	
DPI	$= \{dpi_1, dpi_2, ...\}$ = the set of datapath inputs,	
DPO	$= \{dpo_1, dpo_2, ...\}$ = the set of datapath outputs,	
U	$= \{u_1, u_2, ...\}$ = the set of available functional and storage units,	
u_i	$= \{size, ctrl\}$,	
$size, ctrl$	$\in Natural,$ = the size and number of control lines for a particular unit.	

3.2 Design information

Given the preprocessed information, we can focus on creating a design for the subset of functional objects that have been mapped to hardware. We need to reassemble the datapath and controller, which is done essentially be reversing the above process. Specifically, the datapath FUs required to implement the hardware objects are determined as the union of the FUs needed by each object. In other words, since FUs can be shared among the functional objects due to their sequentiality, the number of each FU type is simply the maximum needed by any one object. For example, if one object requires units *u1* and *u2*, and another requires units *u1* and *u3*, then the datapath FUs will be *u1*, *u2* and *u3*. The datapath SUs are determined similarly. The multiplexer sizes are determined for each unit and datapath output by taking the union of the sources contributed to that unit/output by each object. The number of states in the controller is simply the sum of the number of states of the functional objects (remember that this is the number of *possible* states, rather than a measure of the start-to-finish performance), and the number of states that each datapath control line is active is the sum of those contributed by each object. We store the information in a table, as in Figure 4, that corresponds to a datapath.

From the above discussion, we see that values for the basic parameters for the hardware size functions have been determined, so the size can now be computed by calling $HwSize$, as shown in Figure 4.

We will now define our data structure that maintains the design information in an incrementally updatable manner. The data structure consists of the following:

	destination	sources	contributing fct. objs.	component required	size	control lines	active states
DP outputs	A	C E	Main Proc1	8–bit 2x1 mux	200	1	2
	B	'1' '0'	Main Proc2	1–bit 2x1 mux	25	1	2
FUs	compar1	A D	Proc1 Proc1	8–bit ">"	300	0	1
	adder1	C D E	Proc1, Proc2 Proc1 Proc2	8–bit 2x1 mux 8–bit "+"	200 400	1 0	2 2
SUs	tmp_reg1	adder1	Proc2	8–bit reg.	175	2	1

(from PP)

hwsize = HwSize(wires, srcs_list, units, size_list, ctrl, active_list, states)

hwsize = HwSize(10, srcs_list, 6, size_list, 5, active_list, 20) = ...

Figure 4: Hardware design information for Main, procedure1 and procedure2 functions

D	$= \{dsts, usize, units, ctrl, wires\}$ = the design information,
$usize$	$\in Natural,$ = the size of all FUs, SUs and multiplexers,
$units$	$\in Natural,$ = the number of FUs, SUs and multiplexers,
$ctrl$	$\in Natural,$ = the number of control lines between the controller and the datapath,
$wires$	$\in Natural,$ = the number of wires in the datapath,
$dsts$	$= \{dst_1, dst_2, ...\}$ = the destinations in the design,
dst_i	$= \{id, src\_cons, active\}$,
id	$\in DPO \bigcup U,$ = the unit or DP output that this destination represents,
$active$	$\in Natural,$ = the total number of states that this destination is active,
src_cons	$= \{src\_con_1, src\_con_2, ...\}$ = a list of sources, and the contributing objects for each,
src_con_i	$= \{src, con\}$,
src	is a source from the preprocessed information PP,
con	$\subset PP.O$ = the objects which contribute this source to the destination.

Computational complexity

Relative to the number n of functional objects, the complexity of determining the contribution of each functional object is $O(n)$. The 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 functional object's dataflow graph; c is usually large compared to n.

4 Constant-time update algorithm

We now turn our attention to the movement of functional objects between the hardware and software components, or more specifically, to the addition or deletion of a functional object to or from hardware. Such additions/deletions are made by iterative-improvement algorithms, by a designer performing manual partitioning, and also when creating an initial partition, since an initial partition can be created by moving each functional object to hardware one at a time.

We must define an algorithm that modifies the design information in D to reflect the given object move. We shall start by describing the algorithm informally, and later give a formal algorithm. When a functional object o is added to the hardware, the following actions must be taken:

1. Any destination in o but not already in hardware is added. If an added destination is an FU or SU, the hardware size is incremented by the unit's size, the number of datapath units is incremented, and the number of control lines between the controller and datapath is also increased accordingly.

2. For each o destination, the sources are unioned with the corresponding hardware destination's sources. This union may add sources to a destination and hence may affect the multiplexer size and multiplexer control lines used for the destination. The hardware's size, number of control lines, and number of units are increased accordingly. Also, the number of active states for the control lines of a hardware destination is incremented by the number of active states for o destination.

3. The hardware's number of possible states is incremented by o's number of possible states.

4. The hardware size function is reevaluated with the updated parameters.

Likewise, when a functional object o is deleted from hardware, the following actions must be taken:

1. For each destination, all sources existing solely due to o are deleted. Such deletion may affect the multiplexer size and multiplexer control lines used for the destination. The hardware size and number of control lines are decreased accordingly. The number

of active states for remaining control lines are decremented by the number of active states for the deleted sources. If a multiplexer is no longer needed, the number of units is decremented accordingly.

2. After deleting sources existing solely due to o, a destination may have no remaining sources. The destination is thus deleted. If it was an FU or SU, the hardware size is decremented by the FU or SU size, and the number of control lines is decreased accordingly.

3. The number of possible states is decremented by o's number of possible states.

4. The hardware size function is reevaluated with the updated parameters.

An example of how the design information is updated when a functional object is deleted is shown in Figure 5, in which *procedure2* has been deleted from hardware.

We now define an algorithm to update the design information D for an addition of a functional object o to hardware. The algorithm uses a procedure *SeekDesignDst* which returns the design destination that refers to the same unit as the given object destination. A procedure *NewDesignDst* creates a new design destination for the given object destination. Procedures *Size* and *Ctrl* return the size and number of control lines for the given object destination's unit, respectively, returning 0 if the destination corresponds to a datapath output. A procedure *SeekSrc_con* returns the design's source/contributors item that corresponds to the given source. A procedure *NewSrc_con* creates a new source/contributors item for the given source. A procedure *GetMuxSize* determines the size of the multiplexor(s) needed in front of a particular destination for the given sources. The size is dependent on the number of sources and on whether there is one or two inputs on the destination (e.g., an adder has two inputs so no multiplexer is needed for two sources, whereas an incrementer with two sources does need a multiplexer since it has only one input).

Algorithm 4.1 UpdateDesignInfoForObjectAdd(D, o)

> **for each** $dst_o \in o.dsts$ **loop**
> dst_d =SeekDesignDst($D, dst_o.id$)
>   **if** $dst_d = NULL$ **then**
> dst_d =NewDesignDst(dst_o)
> $D.dsts = D.dsts \bigcup dst_d$
> $D.usize = D.usize+$Size(dst_o)
> $D.ctrl = D.ctrl+$Ctrl(dst_o)
>   **if** $dst_o.id \in FU \bigcup SU$ **then**
> $D.units = D.units + 1$
>   **end if**
>   **end if**
> $muxsize\_bef$ =GetMuxSize($src\_cons, dst_o.id$)
>   **for each** $src \in dst_o.srcs$ **loop**
> src_con =SeekSrc_con($dst_d.src\_cons, src$)
>   **if** $src\_con = NULL$ **then**
> src_con =NewSrc_con(src)
> $dst_d.src\_cons = dst_d.src\_cons \bigcup src\_con$
> $D.wires = D.wires + 1$
>   **end if**
> $src\_con.contribs = src\_con.contribs \bigcup o$
>   **end loop**
> $muxsize\_aft$ =GetMuxSize($src\_cons, dst_o.id$)
> $D.usize = D.usize - muxsize\_bef + muxsize\_aft$
>   **if** $muxsize\_bef = 0$ and $muxsize\_aft > 0$ **then**
> $D.units = D.units + 1$
>   **end if**
> $dst_d.active = dst_d.active + dst_o.active$
> **end loop**
> $D.states = D.states + o.states$
> **return**

The algorithm for deleting a functional object is similar to that shown for adding an object, so we have omitted it for the sake of brevity.

Computational complexity

All changes outlined above can be done in constant-time, as long as the data-structure used ensures that the hardware destinations to which o contributes sources are directly accessible without search. A key assumption is that the number of destinations in a given functional object 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.

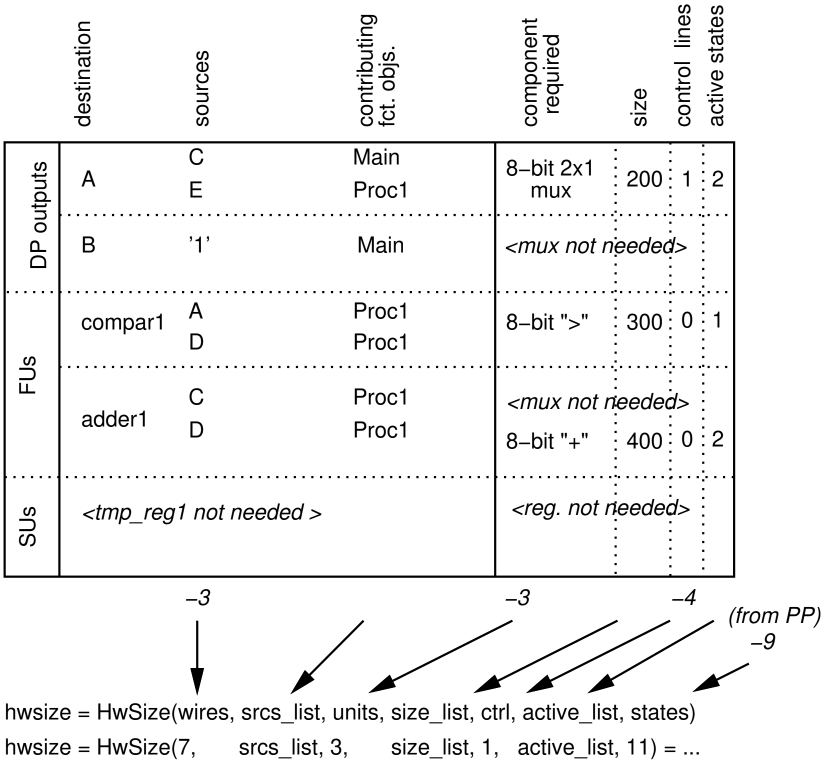


Figure 5: Hardware design information after procedure2 is deleted

5 Incremental estimation of execution time

We now describe our technique for estimating the average execution time of a behavior. 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.

5.1 Execution-time model

For each behavior, we first perform the following steps: allocation, binding and scheduling, as was done for hardware size. 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 can be obtained by performing simulation with a representative set of test vectors. Presently, we output a file containing a list of all branches with default probabilites, which the user can then modify manually. 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 [7], 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.

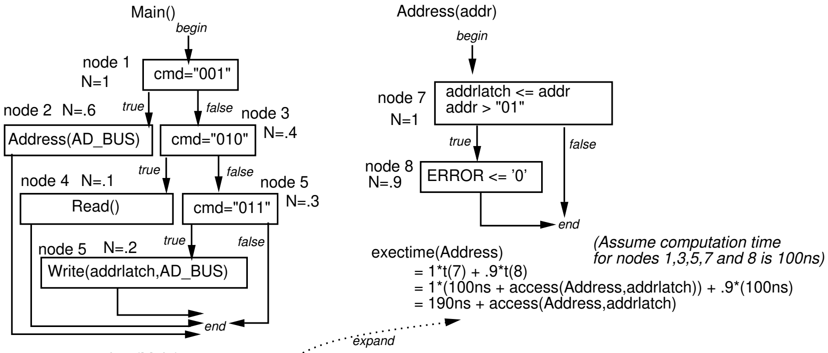
5.2 Preprocessed information

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 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 6(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 6(b), execution-time is computed.

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.

5.3 Constant-time update algorithm

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



exectime(Main)
= 1*100ns + .6*(exectime(Address) + access(Main,Address)) + .4*100ns
+ .1*(exectime(Read) + access(Main,Read)) + .3*100ns
+ .2*(exectime(Write) + access(Main,Write) + access(Main,addrlatch)
= 284ns + .6*access(Address,addrlatch) + .6*access(Main,Address) + .1*access(Main,Read)
+ .2*access(Main,Write) + access(Main,addrlatch) + ... <exectime for Read and Write>

(a) Flowgraphs and basic time equation

Assume: access(procedure1, procedure2) *100ns on-chip* *400ns off-chip* *access(procedure, storage)* *5ns on-chip* *200ns off-chip*

exectime(Main) for initial partitioning: =284 + .6*200 + .6*400 + .1*400 + .2*400 + 5 + ...
= 769 + ...

(b) Time-equation value for initial partitioning

Update for move of Addressfrom chip1 to chip2: only the two access terms involving Address change:
.6*200 --> .6*5; change=117
.6*400 --> .6*100; change=180

exectime(Main) = exectime(Main) - 117 - 180 = 472 + ...

(c) Incrementally updating the time-equation value

Figure 6: Execution-time equation creation, evaluation, and update

simply incremented or decremented by the change in term value. Figure 6(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.

6 Results

We have implemented a design-based incrementally-updatable hardware-size estimator using the previously described data structure and algorithm, and have incorporated it into a functional partitioning tool. The input is a VHDL behavioral description, and the output a refined description containing partition detail. The implementation consists of approximately 16,000 lines of C code. The functional partitioning tool has been released to over 20 companies as part of the SpecSyn system-design environment, and has been used in an industry design (a fuzzy-logic controller) involving five ASICs, and tested on numerous other industry examples including an interactive TV processor and a missile-detection system. The tool is presently being applied to several industry examples in various companies.

example	# fct.. objects	# lines code	time for preproc.	# moves examined	constr.	initial partition	after improvement	time for moves	avg. time per move
mwt	28	603	33.2	639	10000/40	9560/66, 9427/61	9231/37, 9768/39	4.5	**.007**
ans_mach	61	726	63.5	19564	15000/50	14576/133, 14993/125	14918/50, 12663/45	117.4	**.006**
draco	15	302	12.6	1855	8000/100	6241/105, 10003/102	7430/103, 8528/100	11.1	**.006**
ether	64	967	26.0	24251	40000/120	31737/325, 24481/292	42095/117, 17719/127	96.3	**.004**

Figure 7: Results show the our method's speed and constant-time computation

Having shown that constant-time estimation updates are possible, we now demonstrate that the constant is acceptably small for practical use. We have applied the partitioner to many examples; some results are shown in the table of Figure 7. Examples include a microwave-transmitter controller (mwt) designed by an engineering consulting firm, a telephone answering machine (ans_machine) currently available on the market, the DRACO peripheral interface, and the Ethernet coprocessor (ether) introduced in [10]. Details of these examples can be found in [11]. 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 created a random 2-way partition and measured the time to obtain the preprocessed information. We then applied the group migration algorithm [12] extended for multi-way partitions to improve the partitioning, using the cost function specified in [9]. Shown in the table are the number of moves which the algorithm examined, the size/pin constraints on the partitioning (performance constraints were also imposed), the size/pins for each chip in the initial partitioning, the size/pins after applying the improvement algorithm, and the CPU time (in seconds on a Sparc1) required to update the estimation information for these moves. Note that the time-per-move is roughly the same across all four examples, demonstrating that computation is indeed done in constant time. More importantly, 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 as opposed to the traditional estimation approach of reestimating for each partition, consider the results in Figure 8. The table compares the estimation time for our method with the estimation time of a typical design-based estimator ([7]). The typical estimator requires an average of 3 seconds to obtain an estimate for a given partition (which is the same magnitude of time required by other design-based estimators in [1, 2, 4]. While this time is very fast compared to the several minutes required by a high-level sythesis tool to create a real design, it is unacceptably slow for use in the examples. For instance, with the Ethernet example, such an estimator would

Example	Predicted estimation time with previous method	Estimation time with our method	Speedup (with nearly no loss in accuracy)
mwt	1917	4.5	426
ans_mach	58692	117.4	500
draco	5565	11.1	501
ether	72753	96.3	755

Figure 8: The method obtains huge speedups without loss in accuracy

have required over 72,000 seconds (this number is predicted by multiplying 3 seconds with the number of partitions examined of 24251); in other words, over 20 hours. Compare this with the mere 96 seconds used by method, which is just 1.5 minutes, yielding a *speedup* of 755. The results show that we obtain a speedup of between 100 and 1000 times, while using the same design model and hence obtaining estimates of nearly the same accuracy.

As mentioned earlier, estimation accuracy is orthogonal to our method; the method can be applied to various estimation models. (For the interested reader, the estimation accuracy for the design model which we use is discussed in [7, 1, 2]). However, for the sake of completeness, we note that for the answering machine example, the design model we have described yielded an accuracy of 9% compared to the manual design of the same example. This level of accuracy is certainly within reason for making the coarse-grained system-level decisions necessary during functional partitioning. Also note that alternatives to design-based estimation have severe drawbacks. The alternative of using manual estimates requires much designer experience and is likely to yield low accuracy. The second alternative of using abstract-weight-based estimation methods yields grossly inaccurate values. A third alternative, in which structure is designed before partitioning in order to obtain more accurate estimates, does not consider hardware/software partitions, nor does it support architectural changes after partitioning to improve performance and save area.

7 Future work

Turning now to future work, it would be useful to allow the designer to modify the roughly-synthesized design in order to match his own estimates or decisions. For example, he could, for a given functional object, change the estimated number of states, or change the functional units that are required. Another area of work is in developing estimation models for different technologies; one company has already modified our technique for FPGAs. Other models may be developed for custom layouts or MCMs. Finally, it may in the long run be interesting to combine partitioning with RT-level design, so that real designs are used rather than estimates; however, due to the amount of time that RT design presently requires, even with the aid of high-level synthesis tools, such an approach does not seem feasible in the near future, except possibly for the non-interactive partitioning of arithmetic-level dataflow graphs, as described in [13].

The method we have described can be easily extended to handle concurrent processes (we have until now assumed the existence of only a single process). We simply keep separate design information for each process, since each process will use its own CU/DP, and we then add the sizes of all CU/DP's in hardware. It is clear that additional processes do not affect the constant-time characteristics of hardware-size estimation. Another extension of the technique makes it applicable not only to hardware/software partitioning, but also to hardware/hardware partitioning, such as partitioning functionality among multiple ASICs. We simply maintain design information for each ASIC. A move of a functional object will then consist of a deletion from one ASIC's hardware information and addition to another ASIC's information. In fact, the results summarized above were obtained by partitioning functional objects among two ASICs.

8 Conclusions

We have introduced a new method for estimating hardware size during functional partitioning. Previous approaches to size estimation achieved either speed or accuracy, but not both. Our method, based on a data structure and algorithm that incrementally modify a design model, achieves both speed and accuracy. Because of this achievement, functional partitioning systems become much more useful. Previously, their usefulness was severely limited due to grossly inaccurate estimates or due to excessively long estimation times. The method thus makes functional partitioning systems practical for use in real design environments, as our results have demonstrated.

References

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

[2] E. Lagnese and D. Thomas, "Architectural partitioning for system level synthesis of integrated circuits," *IEEE Transactions on Computer-Aided Design*, July 1991.

[3] J. Rajan and D. Thomas, "Synthesis by delayed binding of decisions," in *Proceedings of the Design Automation Conference*, 1985.

[4] K. Kucukcakar and A. Parker, "CHOP: A constraint-driven system-level partitioner," in *Proceedings of the Design Automation Conference*, 1991.

[5] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," in *IEEE Design & Test of Computers*, pp. 64–75, December 1994.

[6] R. Gupta and G. DeMicheli, "Hardware-software cosynthesis for digital systems," in *IEEE Design & Test of Computers*, pp. 29–41, October 1993.

[7] S. Narayan and D. Gajski, "Area and performance estimation from system-level specifications." UC Irvine, Dept. of ICS, Technical Report 92-16,1992.

[8] D. Thomas, J. Adams, and H. Schmit, "A model and methodology for hardware/software codesign," in *IEEE Design & Test of Computers*, pp. 6–15, 1993.

[9] F. Vahid and D. Gajski, "Specification partitioning for system design," in *Proceedings of the Design Automation Conference*, 1992.

[10] R. Gupta and G. DeMicheli, "System-level synthesis using re-programmable components," in *Proceedings of the European Conference on Design Automation (EDAC)*, pp. 2–7, 1992.

[11] S. Narayan, F. Vahid, and D. Gajski, "Modeling with SpecCharts." UC Irvine, Dept. of ICS, Technical Report 90-20,1990.

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

[13] C. Gebotys, "Optimal synthesis of multichip architectures," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 238–241, 1992.