as determined by its configuration. In creating the circuit graph, we had to consider the CLB modes, e.g., we created two separate nodes from one CLB if the CLB was in the "FG" mode which allows one CLB to act as two different logic blocks. We analyzed the LCA XC3090PG175 which has 320 CLB's and 144 I/O blocks.

The experimental results determined using partitioning benchmark data (Partitioning93) [6] are shown in Table I. In the table, "Data" are data names, and data starting with character "c" indicate combinational circuits. Data starting with character "s" indicate sequential circuits. "CLB" and "I/O" indicate the numbers of CLB's and I/O blocks used. "Edge" and "Node" are the numbers of edges and nodes of the circuit graph. "Orig. reg." indicates the number of registers in the original circuit. In addition, columns "A" and "B" show the maximum delay (and the minimum clock frequency, if it is a sequential circuit) of the initial circuit and of the circuit obtained after applying the presented algorithm respectively. Furthermore, "Inserted reg." indicates the number of registers newly inserted by our algorithm, and "A/B" indicates the ratio between columns "A" and "B," which is called the *performance improvement ratio*. Finally, "Delay" indicates the *clock response delay* caused by the proposed algorithm, while "CPU" shows the execution time of the algorithm. All registers in the I/O blocks were originally not used, and the algorithm was allowed to use them as well as the unused registers in CLB's.

In the case of the Xilinx LCA, if a register is inserted, it is necessary to route the clock signal from the clock input pin to the register. In the above analysis, we invoked the routing tool again after register insertion, but did not reroute any nets except to add the appropriate clock nets if routing succeeded. The initial routing data was given to the routing tool as a guide. Furthermore, no added clock nets were critical paths. Thus our assumption for the presented algorithm, i.e., the original placement and routing results are unchanged, was satisfied in practice. For two circuits "c3540xc3" and "s953xc3", unfortunately, the algorithm succeeded but rerouting failed, i.e., some unrouted nets remained. This fact is indicated by the "NG" notation in column "B" in Table I.

The average performance improvement ratios were 4.33 for the combination circuits, and 1.25 for the sequential circuits. This fact indicates that the proposed algorithm effectively inserts registers into the rather long critical paths common in combination circuits.

VI. Conclusion

We have presented a new register insertion algorithm for synchronous circuits realized as LUT-based FPGA's. It offers considerable user support by improving circuit performance without changing the initial placement and routing results. According to experimental results gained with benchmark data, the performance of almost all circuits, including both combination and sequential circuits, is improved by the presented algorithm. For a few circuits, our algorithm succeeded, but actual register insertion failed. Failure, however, was due to the constraint of the Xilinx FPGA used, not the algorithm itself. In other words, if the FPGA had a mechanism that controlled the closure of the register located at the output of each LUT without changing the original configuration [7], the proposed algorithm would improve the performance of all circuits. The execution speed of the algorithm is fast so it can be applied to large scale FPGA's. i.e., multichip FPGA systems.

If register insertion fails, it is useful to apply *retiming* before invoking the presented algorithm in order to change the initial register locations. However, the constraints of the location and number of registers in the FPGA should be considered if *retiming* is applied. This is one of our future tasks.

References

[1] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic, *Field-Programmable Gate Arrays.* Norwell, MA: Kluwer, 1992.

[2] C. E. Leiserson, F. M. Rose, and J. B. Saxe, "Optimizing synchronous circuitry by retiming," in *Proc. 3rd Caltech Conf. VLSI*, Mar. 1983, pp. 87–116.

[3] H. Nakada, K. Yamada, A. Tsutsui, and N. Ohta, "A design method for realizing real-time circuits on multiple-FPGA systems," in *More FPGA's*, W. R. Moore and W. Luk, Eds. Oxford, UK: Abingdon, 1993.

[4] N. Park and A. C. Parker, "Sehwa: A software package for synthesis of pipelines from behavioral specifications," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 356–370, Mar. 1988.

[5] "The field programmable gate array data book," Xilinx Inc., 1992.

[6] R. Kužnar, F. Brglez, and K. Kozminski, "Cost minimization of partitions into multiple devices," in *Proc. 30th DAC*, June 1993, pp. 315–320.

[7] N. Ohta *et al.* "PROTEUS: Programmable hardware for telecommunication systems," in *Proc. ICCD'94*, Oct. 1994, pp. 178–183.

Incremental Hardware Estimation During Hardware/Software Functional Partitioning

Frank Vahid and Daniel D. Gajski

Abstract—**To aid in the functional partitioning of a system into interacting hardware and software components, fast yet accurate estimations of hardware size are necessary. We introduce a technique for obtaining such estimates in two orders of magnitude less time than previous approaches without sacrificing substantial accuracy, by incrementally updating a design model for a changed partition rather than re-estimating entirely.**

Index Terms— **Constant-time complexity, estimation, hardware size, hardware-software co-design, incremental design, interactive design, system design, system partitioning.**

I. Introduction

The designer of an embedded system is often faced with the challenge of partitioning the system functionality for implementation among hardware and software components, such as among ASIC's and processors. New approaches for such partitioning start with a simulatable specification of system functionality, and then explore numerous possible partitions of that specification's functions among the hardware and software components [1]. We therefore need a method to determine, among other things, the hardware size of a set of functions, to see if that set will meet constraints.

Manuscript received April 19, 1994; revised September 19, 1994. This work was supported by the National Science Foundation under Grant MIP-8922851 and by the Semiconductor Research Corporation under Grant 92-DJ-146.

F. Vahid is with the Department of Computer Science, University of California, Riverside, CA 92521 USA.

D. D. Gajski is with the Department of Information and Computer Science, University of California, Irvine, CA 92717 USA.

IEEE Log Number 9413466.

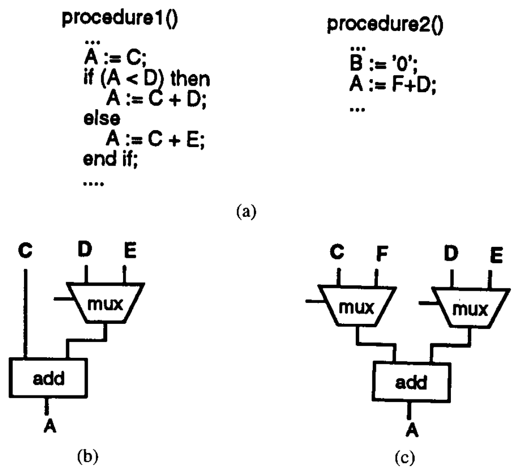


Fig. 1. Example of incremental change resulting from object move: (a) functional objects, (b) partial datapath for Procedure1, and (c) partial datapath for both procedures.

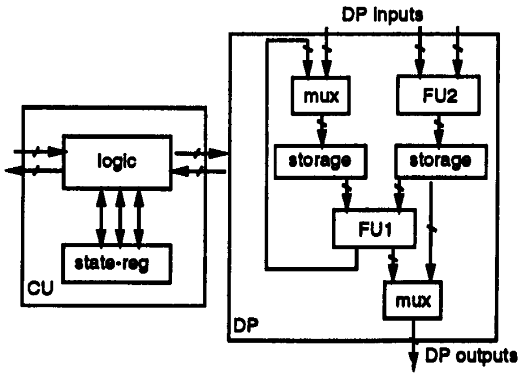


Fig. 2. CU/DP area model.

There are several possible methods. The most accurate would be to synthesize a design for the set of functions, 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 [2]–[5]. In essence, those estimators roughly synthesize a design for the given functions, 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 estimators based on a *design model* have the advantage of obtaining accurate estimates in just a few seconds. At times, though, we wish to examine hundreds or thousands of partitions using an iterative-improvement partitioning heuristic such as simulated annealing, thus requiring faster estimators. Approaches that use iterative-improvement heuristics have until now used abstract-weight-based estimators, in which an abstract weight is assigned to each function, and then a hardware "cost" for a given partition is obtained quickly just by adding all the weights of functions in hardware [6]–[8]. Alternatively, they assume an already scheduled input and estimate hardware size as the size of required functional units [9]. These approaches have the advantage of obtaining very rapid estimations.

In developing our system that partitions an unscheduled specification among hardware and software components, we desired to use an estimator based on a design-model in order to obtain accuracy, but we also wanted to use iterative-improvement algorithms to explore many possibilities. Since previous estimation methods had not addressed both goals, we needed to develop a new method. Toward 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. For example, Fig. 1(a) shows two functions and Fig. 1(b) shows a partial datapath for one of those functions. When we add the other function, the datapath only requires one additional multiplexer, as shown in Fig. 1(c).

We took advantage of this incremental change by developing a data structure (representing an incrementally modifiable design model) and an algorithm that can quickly provide the basic design parameters needed by a hardware-size estimator. As we shall see, we were able to do this by assuming that the granularity at which we partition the specification is at the procedural level (sometimes called the process or task level), as is the case in many new functional partitioning techniques [7], [8], [10]–[15]. Our contribution is the development of this incremental hardware-size estimation method, consisting of a new data structure and algorithm, that achieves the advantages of both classes of previous approaches, namely accuracy and speed.

This paper is organized as follows. In Section II, we describe the design-model that we use for hardware-size estimation, a model adopted from previous design-based estimators. In Section III, we describe a new data structure that captures not only the design model, but also the contribution of each function to the design. In Section IV, we detail an algorithm for updating, in constant time, this data structure when a function is moved. In Section V, we summarize our results that show the speed of our method.

II. Estimation Design Model

The design model we use to obtain hardware-size estimates for a set of functions is a control-unit/datapath (CU/DP) model [2], [16], as shown in Fig. 2. The size for the model can be computed as the sum of the following: *functional-unit and storage-unit size* (including registers, register files and memories), *multiplexer size*, *state-register size*, *control-logic size*, and *wiring size*. Each is a function of one or more of the following basic *design parameters*:

1) *states* number of possible controller states,
2) *size_list* a list of datapath units, where each unit has an associated size,
3) *srcs_list* a list of datapath units, where the number of sources (i.e., outputs of other units or datapath inputs) that must at some time be input to each unit is specified,
4) *ctrl* the number of control lines between the controller and the datapath,
5) *active_list* a list of all control lines, where the number of states for which a control line must be asserted is associated with each control line,
6) *units* the number of units in the datapath, regardless of their types,
7) *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 *active_list*, and the *wiring size* may be a function of *units*, *size_list*, and *wires*. The details of these functions are beyond the scope of this paper; any function that uses the above design parameters could be used in conjunction with our method, and more than one form of each function may exist to support estimation for various technologies. To avoid going into specific details of those functions, in this article we assume that a function $HwSize$ exists, which uses the above parameters and which returns a hardware size with the appropriate units for the particular hardware technology, such as square microns, transistors, gates, or combinational logic blocks.

III. Incrementally Updatable Data Structure

We now describe the data structure that allows us to represent a roughly synthesized design using the above model, while at the same time allowing us to incrementally modify that design in *constant time* when a functional object is added or deleted. We assume that the specification consists of a single process with hundreds or thousands of sequential statements, including loops, branches, and procedure calls. We later describe a simple extension for multiple processes. We shall hereon refer to the specification pieces to be distributed among hardware and software components as functional objects.

A. Preprocessed Information

Our first task is create a hardware design that implements the entire set of functional objects, and to determine the contribution that each functional object makes to that design (in order to support incremental change). Since this information can be obtained before creating a partition, we call it *preprocessed information.*

To obtain the hardware design, we must allocate a set of functional units (FU's) and storage units (SU's), bind operations and data values to FU's and SU's, and schedule operations into control steps (not necessarily in the given order). The heuristics that we use to do these tasks should match the heuristics 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 heuristics instead.

To determine the contribution of each functional object to the design, we first consider the datapath. We create a list of FU's for each functional object. For example, if a functional object uses two adder units, then we append two adder units to that object's FU list. We create a similar list of SU's for each functional object. Turning to multiplexers, we note that the size of a multiplexer in front of an FU, SU, or datapath output is determined by how many possible *sources* (i.e., SU or FU outputs, or datapath inputs) may need to be input to that FU, SU, or datapath output. Thus for each functional object, we associate a list of sources contributed by that object to each FU, SU, and datapath output. Turning our attention to the control unit, we record the number of possible states for each functional object, and the number of states that each FU, SU, and datapath output is active.

At this point, an assumption that we wish to make explicit is that a functional object represents a coarse-grained computation, such as a process, procedure, or a large basic block, as also assumed in many new functional partitioning techniques (see Section I). The larger the number of statements in each object, the more accurate the estimations will be, since inter-object synthesis optimizations would then play a smaller role in the overall design. The reason is that we assume that the tasks of scheduling, allocation and binding for two functional objects will be roughly the same whether we consider each object independently or together, because in our approach, we perform those tasks on each functional object independently. On the other hand, lower levels of granularity, such as small basic blocks, would result in less accurate estimates since current synthesis techniques (such as path-based scheduling and percolation scheduling) optimize across basic block boundaries.

Fig. 3 shows the preprocessed information created for each procedure of the example in Fig. 1(a). Note that this example is trivially small, but that it sufficiently demonstrates our technique.

Functional object	States	Destination	Sources	Active states
Procedure1	5	A	C adder1	3
		comparator1	A D	1
		adder1	C D E	2
		storage1	comparator1	1
Procedure2	2	A	adder1	1
		adder1	F D	1
		B	'0'	1

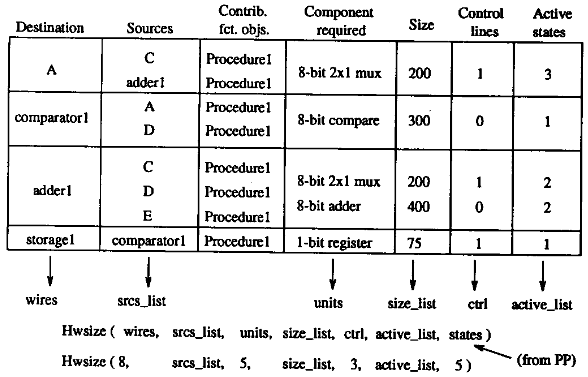
Fig. 3. Preprocessed information for functional objects O.

More formally, the data structure of preprocessed information, or PP, is a four-tuple $\langle O, DPI, DPO, U\rangle$. DPI is a set of datapath inputs $\{dpi_1, dpi_2, \cdots\}$, and DPO is a set of datapath outputs $\{dpo_1, dpo_2, \cdots\}$. U is a set of available functional and storage units $\{u_1, u_2, \cdots\}$. Each unit u_i is a pair $= \langle size, ctrl\rangle$, where *size* is a natural number representing the size of the unit (in transistors, gates, or whatever type is assumed by the estimation functions), and *ctrl* is a natural number representing the number of control lines on that unit.

O is a set of functional objects $\{o_1, o_2, \cdots, o_n\}$. Each functional object o_i is a pair $\langle states, dsts\rangle$. *states* is a natural number representing the number of possible control states for the functional object. *dsts* is a set of destinations, $\{dst_1, dst_2, \cdots\}$, written to by the object. A destination dst_i is a three-tuple $\langle id, srcs, active\rangle$. The destination identifier *id* is the particular FU, SU, or DP-output that *dst* represents, so $id \in DPO \bigcup U$. *active* is a natural number representing the number of states for which the destination is active for this object. *srcs* is a set of sources, $\{src_1, src_2, \cdots\}$, that the object assigns to this destination. Each src_i is either a datapath input or a unit, so $src_i \in DPI \bigcup U$.

B. Design Information

Given the preprocessed information PP, we can focus on creating a design for the subset of functional objects that have been mapped to hardware. We need to assemble the datapath and controller. Specifically, the datapath FU's required to implement the hardware objects are determined as the union of the FU's needed by each object. For example, if one object requires units *u1* and *u2*, and another requires units *u1* and *u3*, then the datapath FU's will be *u1, u2*, and *u3*. The datapath SU's are determined similarly. The multiplexer sizes are determined for each destination by taking the union of the sources contributed to that destination 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. For example, Fig. 4 shows this information for the case when *Procedure1* from Fig. 1(a) is the only functional object mapped to hardware.



Destination	Sources	Contrib. fct. objs.	Component required	Size	Control lines	Active states
A	C adder1	Procedure1 Procedure1	8-bit 2x1 mux	200	1	3
comparator1	A D	Procedure1 Procedure1	8-bit compare	300	0	1
adder1	C D E	Procedure1 Procedure1 Procedure1	8-bit 2x1 mux 8-bit adder	200 400	1 0	2 2
storage1	comparator1	Procedure1	1-bit register	75	1	1

wires
srcs_list
units
size_list
ctrl
active_list
(from PP)

Hwsize (wires, srcs_list, units, size_list, ctrl, active_list, states)

Hwsize (8, srcs_list, 5, size_list, 3, active_list, 5)

Fig. 4. Hardware design information for procedure1.

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 Fig. 4.

We will now define our data structure that maintains the design information in an incrementally updatable manner. The design information data structure D is a five-tuple $\langle usize, units, ctrl, wires, dsts\rangle$. The first four items are natural numbers. *usize* represents the total size of all the FU's, SU's, and multiplexers. *units* represents the total number of all FU's, SU's, and multiplexers. *ctrl* represents the total number of control lines between the controller and the datapath. *wires* represents the number of wires in the datapath.

The fifth item, *dsts*, is a set of all destinations in the design, $\{dst_1, dst_2, \cdots\}$. Each destination dst_i is a three-tuple $\langle id, src\_cons, active\rangle$. The identifier *id* indicates the unit or DP output that this destination represents, so $id \in DPO \bigcup U$. *active* is a natural number that indicates the total number of states that this destination is active. *src_cons* is a set $\{src\_con_1, src\_con_2, \cdots\}$, where each src_con_i is a pair $\langle src, con\rangle$. *src* is a source, from the preprocessed information PP, that must be input to the destination dst_i. *con* is a set of functional objects (i.e., $con \subset PP.O$), where each functional object requires a path from the source to the destination. In other words, the objects are the contributors of the source to the destination.

Relative to the number n of functional objects, the complexity of building PP and D is $O(n)$. For the industry examples that we have examined, n has ranged from 15 to 120. 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.

IV. 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. We 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, respectively, for the given object destination's unit, 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 are 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). If there is more than one input on the destination, we assume the sources are uniformly distributed among those inputs.

Algorithm 4.1 *UpdateDesignInfoForObjectAdd*(D, o):

```
for each $dst_o \in o.dsts$ loop
  —Add destination to design if it doesn't yet exist
  $dst_d = SeekDesignDst(D, dst_o.id)$
  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
  —Update mux sources and sizes
  $muxsize\_bef = GetMuxSize(src\_cons, dst_o.id)$
  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
  —Update control line active states for this dst
  $dst_d.active = dst_d.active + dst_o.active$
end loop
—Update controller states
$D.states = D.states + o.states$
return
```

The algorithm performs the following for each destination written in o. First, it adds that destination to the design if it doesn't already exist. Such an addition requires updating the number and size of DP units, and the number of control lines between the CU and DP. Second, it unions the sources of that destination with the corresponding design destination's sources. If such a union adds sources, then we must update the number of DP wires and the size of the destination's multiplexer. If previously no multiplexer was needed, but after adding a source a multiplexer is needed, then the number of DP units is incremented. Third, the algorithm increases the number of states for which the destination must be asserted by the number of states for which o asserts that destination. After repeating the above three steps for all destinations, the algorithm updates the number of possible controller states by the number of states for o. The algorithm for deleting a functional object is complementary to that for adding an object; we have omitted it for brevity.

Destination	Sources	Contrib. fct. objs.	Component required	Size	Control lines	Active states
A	C adder1	Procedure1 Procedure1, **Procedure2**	8-bit 2x1 mux	200	1	**4**
comparator1	A D	Procedure1 Procedure1	8-bit compare	300	0	1
adder1	C D E **F**	Procedure1 Procedure1 Procedure1 **Procedure2**	8-bit 2x1 mux **8-bit 2x1 mux** 8-bit adder	200 **200** 400	1 **1** 0	2 **2** 2
storage1	comparator1	Procedure1	1-bit register	75	1	1
B	**'0'**	**Procedure2**	--			**1**

$\downarrow$ wires $\quad\downarrow$ srcs_list $\quad\downarrow$ units $\quad\downarrow$ size_list $\quad\downarrow$ ctrl $\quad\downarrow$ active_list

Hwsize (wires, srcs_list, units, size_list, ctrl, active_list, states)

Hwsize (10, srcs_list, 6, size_list, 4, active_list, 7) $\leftarrow$ (from PP)

Fig. 5. Hardware design information after procedure2 is added.

Fig. 5 illustrates several changes we make to the design information when adding *Procedure2* to the hardware. First, we create a new destination B. Second, we increase the adder's active states from 3 to 4. Third, we associate a new source with the adder, resulting in the need for another multiplexer. We then update the parameters to the $HwSize$ function accordingly.

The algorithm executes in constant time, if we assume that the number of destinations per object is roughly constant for a given example. This assumption holds unless each functional object accesses every data item and external port. However, since functional objects (such as procedures) serve to modularize a specification, such a situation is highly unlikely. Instead, each object will likely access a small (constant) number of data items and ports.

Multiple processes can be handled with a straightforward extension. Since we assume each process will use its own controller and datapath, then we simply keep separate design information for each process, and we then add the sizes of all CU/DP's in hardware. The additional processes therefore do not affect the constant-time characteristics of the estimation. We could also handle partitioning among multiple hardware components (such as among ASIC's or among blocks on an ASIC) simply by maintaining separate design information for each ASIC.

V. Results

We have implemented a design-based incremental 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 ASIC's, 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.

The speed of our incremental estimation data structure and algorithm on several examples is illustrated in Fig. 6. Examples include a microwave-transmitter controller (mwt), a telephone answering machine (ans), the DRACO peripheral interface (draco), and an Ethernet coprocessor (ether). To provide a notion for the size of each example, we indicate the number of functional objects to be partitioned, the number of specification lines, and the final size of one hardware ASIC (in gates) after partitioning, as estimated by our $HwSize$ function. Incidentally, the first three examples consisted of one process, while the Ethernet coprocessor example contained 14 processes. For each example, we first measured the time to build the preprocessed information. We then applied the group migration heuristic [17], using the cost function specified in [10]. Shown in the table are the number of moves that the heuristic examined, and the CPU time (in seconds on a Sparc1) required to update the estimation information and obtain a new hardware size estimate for each move. 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. The last two columns demonstrate the increased speed compared with a previous design-based estimator [16]. That estimator requires roughly 3 s for a given partition, which is the same magnitude of time required by several other design-based estimators [2], [3]. Multiplying by the number of moves yields a predicted estimation time; note the unacceptably long times for the large number of moves examined. The last column shows the speedup of our estimator over those previous ones, ranging from 426 to 755; such speedup is obtained while using the same design model.

example	# fct. objects	# lines code	time for preproc.	# moves examined	final size of ASIC1	avg. time per move	est time	prev est time	speedup
mwt	28	603	33.2	639	9231	.007	4.5	1917	426
ans	61	726	63.5	19564	14918	.006	117.4	58692	500
draco	15	302	12.6	1855	6241	.006	11.1	5565	501
ether	64	967	26.0	24251	42095	.004	96.3	72753	755

Fig. 6. Results show the method's speed and constant-time computation.

We also conducted experiments to determine the effect of performing scheduling and allocation on each behavior individually, rather than considering all behaviors at the same time as in previous, slower design-based estimators. For the *ether* and *ans* examples, we inlined all subroutines; for the *mwt* example, such inlining generated an enormous output due to the many nested levels of subroutine calls, so we instead considered a subset of the specification consisting of four subroutines. We then applied the same scheduling and allocation tool to those inlined versions. Results of estimating all-hardware implementations are summarized in Fig. 7; since we are considering all behaviors, the numbers are likely the worst case. Note that the number of states *States,* the number of control lines *Ctrl,* and the functional unit and multiplexor component areas *Comparea* are quite close, and the total sizes computed by the $Hwsize$ function have an average error of only 7%. We also compared these estimates with what would have been obtained using previous weight-based techniques: we performed scheduling and allocation for each behavior, computed the size of each behavior, and then summed those sizes over the entire design. Note that the weight-based estimates are extremely inaccurate, with an average error of 80%. Those estimates greatly underestimate the control and routing area, while overestimating the total component area. Weight-based techniques assume that the behaviors combine in a linear manner, but the behaviors in fact share many components, and the PLA and routing sizes grow nonlinearly (hence, there is no simple factor by which we can multiply the weights to improve the accuracy over all cases).

It is difficult to compare our estimates with implementation values. The reason is that there are many possible implementations for a given set of functions that trade off speed and size, so choosing the implementation to compare with is hard. A second difficulty is that because we are dealing with large, industry examples, obtaining a

	Weight based	Incremental design-based				Standard design-based			
Example	Total	States	Ctrl	Comparea	Total	States	Ctrl	Comparea	Total
ether	30029	208	218	15060	176680	220	232	19186	198445
mwt	7562	54	41	3465	11260	54	49	3665	12090
ans	14895	124	122	8061	64813	106	104	7035	62088

Fig. 7. Comparison with weight-based and standard estimates.

real implementation takes many months. A third difficulty lies in the fact that there are many possible $HwSize$ functions that can be used in conjunction with our design parameters. Nonetheless, we compared our size estimations for part of the answering machine example with an implementation. The implementation was developed by a designer who hand-designed the datapath and hand-specified the controlling state-machine; the state-machine was then implemented with the KISS synthesis tool. We estimated 7804 gates, while the implementation consisted of 5372 gates. A second rough comparison can be made with an industry design of a fuzzy-logic controller. We estimated 129 000 gates, whereas the actual implementation consisted of five 20 000 gate FPGA's. We hope to obtain more comparisons as the tool is used in more designs.

VI. Conclusions

We have introduced a method to rapidly estimate hardware size during functional partitioning. The method includes a data structure representing a design model, and an algorithm that incrementally updates that data structure during functional partitioning, thus yielding rapidly computed design parameters that can be input to any number of hardware estimation functions. The method is the first to achieve both advantages of being based on a design model, and of computing estimates in constant time; previous approaches achieved one advantage or the other, but not both. The method therefore enhances the usefulness of hardware as well as hardware/software functional partitioning tools in real design environments. The general method of developing an incrementally updatable design model for estimation purposes may be applicable to many other estimation problems, such as estimation of hardware or software power consumption, hardware or software execution time, and bus bitrates. Thus, the method may become increasingly significant as design effort shifts toward system-level design exploration.

Acknowledgment

The authors would like to thank S. Narayan for his development of the estimation tools on which this work is based.

References

[1] W. Wolf, "Hardware-software co-design of embedded systems," *Proc. IEEE,* vol. 82, pp. 967–989, 1994.
[2] E. Lagnese and D. Thomas, "Architectural partitioning for system level synthesis of integrated circuits," *IEEE Trans. Computer-Aided Design,* pp. 847–860, July 1991.
[3] K. Kucukcakar and A. Parker, "CHOP: A constraint-driven system-level partitioner," in *Proc. Design Automat. Conf.,* 1991, pp. 514–519.
[4] S. Antoniazzi, A. Balboni, W. Fornaciari, and D. Sciuto, "A methodology for control-dominated systems codesign," in *Int. Workshop Hardware-Software Co-Design,* 1994, pp. 2–9.
[5] X. Xiong, E. Barros, and W. Rosentiel, "A method for partitioning UNITY language in hardware and software," in *Proc. Europ. Design Automat. Conf. (EuroDAC),* 1994.
[6] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," *IEEE Design Test Comput.,* pp. 64–75, Dec. 1994.
[7] R. Gupta and G. DeMicheli, "Hardware-software cosynthesis for digital systems," *IEEE Design Test Comput.,* pp. 29–41, Oct. 1993.
[8] A. Kalavade and E. Lee, "A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem," in *Int. Workshop Hardware-Software Co-Design,* 1994, pp. 42–48.
[9] Y. Chen, Y. Hsu, and C. King, "MULTIPAR: Behavioral partition for synthesizing multiprocessor architectures," *IEEE Trans. Very Large Scale Integr. Syst.,* vol. 2, pp. 21–32, Mar. 1994.
[10] F. Vahid and D. Gajski, "Specification partitioning for system design," in *Proc. Design Automat. Conf.,* 1992, pp. 219–224.
[11] D. Thomas, J. Adams, and H. Schmit, "A model and methodology for hardware/software codesign," *IEEE Design Test Comput.,* pp. 6–15, 1993.
[12] P. Gupta, C. Chen, J. DeSouza-Batista, and A. Parker, "Experience with image compression chip design using unified system construction tools," in *Proc. Design Automat. Conf.,* 1994, pp. 250–256.
[13] T. Ismail, M. Abid, and A. Jerraya, "COSMOS: A codesign approach for communicating systems," in *Int. Workshop on Hardware-Software Co-Design,,* 1994, pp. 17–24.
[14] J. D'Ambrosio and X. Hu, "Configuration-level hardware/software partitioning for real-time embedded systems," in *Int. Workshop Hardware-Software Co-Design,* 1994, pp. 34–41.
[15] P. Eles, Z. Peng, and A. Doboli, "VHDL system-level specification and partitioning in a hardware/software co-synthesis environment," in *Int. Workshop on Hardware-Software Co-Design,* 1992, pp. 49–55.
[16] D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems.* Englewood Cliffs, NJ: Prentice–Hall, 1994.

Efficient Semicustom Micropipeline Design

Alessandro De Gloria and Mauro Olivieri

***Abstract*—We present the analytical model and the electrical characterization of a controllable delay component for a micropipeline architecture suitable for being designed with a semicustom design approach. An interesting feature of the component is that it is lockable, i.e., it can be controlled in an on/off fashion, permitting synchronous operation for testing purposes by means of an opportune architecture model.**

I. Introduction

The micropipeline approach [10] improves conventional pipeline features by introducing a self-timed synchronization mechanism, based on the interchange of signals among the stages of the pipeline. Thus it allows signal locality in the control path of a circuit by avoiding the use of a centralized clock.

Micropipelines are not yet mature to be a standard industrial design technique, due to the mechanism used to avoid the global clock. Micropipelines adopt a scheme based on the bundled-data convention, using *ad hoc* bundling delays for each stage of the pipe. This leads to circuits in which the topology of the final layout takes an important role for guaranteeing correct operation.

The industrial exploitation of micropipelines is even more difficult if we consider the testing phase. Research on asynchronous testing is presently at an early stage [7]–[9]. To overcome this limitation,

Manuscript received April 22, 1994; revised November 30, 1994 and March 1, 1995.

The authors are with the Department of Biophysical and Electronic Engineering (DIBE), Genova 16145, Italy.

IEEE Log Number 9413467.