Towards a model for hardware and software functional partitioning

Frank Vahid and Thuy dm Le
Department of Computer Science
University of California, Riverside, CA 92521

Abstract

We describe a model that supports the functional partitioning of a system-level functional specification among hardware and software components. The model includes only the information needed by partitioning, and thus can be communicated freely and generated automatically. Based on characteristics of several real examples, we describe a technique for automatically generating generic model instances, on which partitioning heuristics can be applied and fairly compared. Such comparisons will become increasingly important as research begins to focus on fast yet effective functional partitioning techniques. We describe a set of tools for converting a specification to the model, for generating generic model instances, and for applying and comparing partitioning heuristics, available via ftp. Use of these tools may greatly reduce duplicated efforts among researchers wishing to investigate hardware/software partitioning heuristics.

1 Introduction

Given a system's functional specification, written in a VHDL or C-like language, a system designer must assign specification pieces, such as procedures, processes, variables, and communication channels, to one or more system components, such as standard processors, custom-synthesized processors, memories, and buses. System components may be stand-alone packages or embedded within other components, as in the case of an embedded core processor and several custom hardware blocks and memories on a single ASIC.

Functional partitioning becomes necessary when a specification will be implemented with more than one system component. We may use more than one component for several reasons. First, we can perform *tradeoffs* between hardware and software. Specifically, we can reduce the execution time of a primarily software system by using a custom hardware component for time-consuming computations. Such a technique was used in [1] to reduce the execution time of an example from 22,403 down to 16,394 cycles. Conversely, we can reduce the hardware cost of a primarily hardware system by moving some hardware functions to a standard processor component. Second, we can decrease execution time by increasing *concurrency* with two custom/standard processors rather than one; a reduction from 10 down to 6.5 microseconds is demonstrated for an example in [2]. Third, we may use functional partitioning to better meet hardware *packaging constraints*. For example, results in [3] describe an example that would require two 832-pin FPGA's if structurally partitioned two ways (a clearly impractical number of pins), while only requiring two 164-pin FPGA's if functionally partitioned. Finally, we can use functional partitioning to achieve greatly improved *synthesis-tool performance*. In [3], the logic synthesis time for an example was reduced from 19 down to 1.25 hours using functional partitioning.

For the above reasons, hardware and software functional partitioning heuristics will likely become increasingly important. We say hardware and software, rather than hardware/software, because we may at times wish to partition among hardware components only, or among software components only. Recently, many techniques have been described that use existing or new heuristics for hardware/software partitioning [1, 4, 5, 6, 7, 8] as well as hardware/hardware partitioning [9, 10, 11, 12]. The former techniques focus on maximizing performance while minimizing hardware size, and the latter on satisfying packaging constraints while minimizing communication time. Other techniques assume a manual partitioning [13, 14, 15, 16, 17], but could certainly be extended to use heuristics. The work presented here is intended to greatly improve the ability to make fair comparisons of various heuristics. In particular, we need a well-defined model on which to apply partitioning heuristics.

A model represents the specification information required for partitioning. It represents specification pieces to be partitioned (thus defining the granularity of partitioning), as well as information used to evaluate partitions, such as the size and execution time of each piece. The model need not represent the full functionality. A well-defined model is needed for several reasons. First, we can use *automatic generation* to create large numbers of generic model instances, thus permitting application of a new partitioning heuristic on many examples, providing enough data for a good evaluation of average/worst/best case heuristic performance. If instead we used real examples, we would need several weeks to develop each the example, meaning that we could only develop a few examples, making heuristic evaluation difficult. Second, we can freely *communicate* model instances among researchers, because the model does not represent a system's full functionality and thus does not contain proprietary information typically found in real examples, even when the model is derived from a real example. Such communication enables fair comparison of partitioning heuristics. Finally, a well-defined model provides for partitioning from a *partial specification*, because the designer need only summarize the information necessary to build the model (e.g., the size and parameters of a procedure rather than its contents).

This paper is organized as follows. In Section 2, we describe the problem being addressed. In Section 3, we highlight the features of our model. In Section 4, we describe our technique for generating generic model instances. In Section 5, we describe a tool to support partitioning of the model, and experiments comparing existing heuristics for hardware/software partitioning. In Section 6, we provide conclusions.

0-8186-7243-9/96 $05.00

2 Problem definition

We assume the input specification is written in a sequential program-like language, such as VHDL or C. The specification is modular, consisting of many procedures; large sections of non-procedural code can be modularized using techniques in [18]. The specification may contain multiple processes and large arrays. The procedures describe task-level algorithms, as opposed to states or arithmetic components. The specification may possess some architectural-level structural detail, including predesigned components such as a DMA controller, priority interrupt controller, or FFT module, and may describe an initial partition among hardware and software components.

Our goal is to partition the procedures/processes, variables, and communication channels among allocated system components and buses, such that we minimize a given objective function value. System-level objective functions are hard to define, because we must consider many competing metrics, such as size, I/O (input/output), execution time, power, design time, modifiability, and monetary cost. We use a function that sums all constraint violations, and in this paper we consider constraints on the metrics of average execution time of a procedure or process (1 iteration), the size capacity of a hardware or software component, and the I/O capacity of a hardware component. Violation values must be zero or positive (negative values are forced to 0), and each value is normalized by dividing by the constraint value.

Figure 1 shows the roles of a model in a typical partitioning system intended to solve the above problem. The model will be derived from the input specification. It must explicitly represent the objects to be partitioned among components; in our case, those objects are procedures and processes, variables and communication channels. It must provide enough information to provide fast yet accurate metric estimations. It must be able to reflect the partitioning results. We shall now highlight the model we use for these roles.

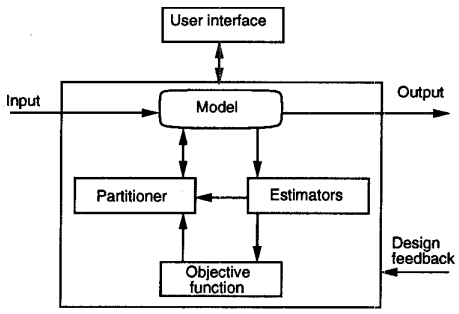

Fig. 1: Typical partitioning system configuration

3 Model

We use the SLIF – Specification-Level Intermediate Format – representation as our model for partitioning. SLIF was introduced in [19] and its usefulness was shown for estimating size, execution time, I/O and bitrate metrics. In its simplest form, SLIF is a directed acyclic graph, where each node represents a behavior or variable, and each directed edge represents an access initiated by a behavior to another behavior or variable. For example, Figure 2 shows a partial VHDL specification of a fuzzy controller, and Figure 3(a) shows its SLIF representation (*Convolve* has been omitted for readability of the figure). As a convention, behavior names are initially capitalized, while variables are not, and process names are underlined. Note that the edge from *EvalRule* to *in1val* points to in1val, even though data flows to *EvalRule*. In other words, the direction shows the initiator of the access, not the flow of data; data might actually flow in both directions. Thus, we call this an **access graph**. Such a representation of accesses, rather than data dependencies, is crucial to represent procedural specifications for partitioning. In particular, in the common case that a procedure is called from many locations, the access graph uses one node for the procedure with multiple incoming edges. If we instead represented data dependencies, then we would have to duplicate any procedure node that was called from more than one location, since data dependencies will be different for each call, resulting in a very large number of nodes.

```
FuzzyMain: process
    variable in1val, in2val : integer;
    type mr_array is array (1 to 384)
      of integer;
    variable mr1, mr2: mr_array;
    type tmr_array is array (1 to 128)
      of integer;
begin variable tmr1, tmr2: tmr_array;
    ...
    in1val := in1;  in2val := in2;
    EvalRule(1);
    EvalRule(2);
    Convolve;
    out1 <= Centroid;
    wait until ...
end process;
```

```
procedure EvalRule(num : in integer) is
  variable trunc : integer;  -- truncated value
begin
  if (num = 1) then
    trunc := Min(mr1(in1val), mr1(128+in1val));
  elsif (num = 2) then
    trunc := Min(mr2(in2val), mr2(128+in2val));
  end if;

  for i in 1 to 128 loop
    if (num = 1) then
      tmr1(i) := Min(trunc, mr1(256+i));
    elsif (num = 2) then
      tmr2(i) := Min(trunc, mr2(256+i));
    end if;
  end loop;
end;
```

Fig. 2: Example VHDL specification

The SLIF is hierarchical. We can group nodes into a new hierarchical node. These nodes may simply be nodes that should not be separated during partitioning. More importantly, the hierarchical node may represent a system component, such as a processor. Likewise, we can group edges into a hierarchical (undirected) edge. The new edge may represent a physical bus. For example, Figure 3(b) shows the SLIF after all behaviors and variables have been assigned to one of three component nodes: a custom hardware block, a standard processor block, and a memory block. In addition, several edges have been assigned to a single bus.

SLIF is heavily annotated for the purpose of metric estimations. First, each node may be bound to a library component. In the example, node *Block1* is bound to a Xilinx XC4000 FPGA, and edge *Bus1* is bound to an 8-bit ISA bus. Note that any of the procedure nodes could be bound to a component also, meaning that the procedure would be implemented using a predesigned component. Second, each node may have a size annotation. In the example, procedure node *Centroid* is annotated with a size of 9000 gates when bound to an XC4000, and a size of 60 instructions when bound to an Intel386 processor. Third, each edge node may have an internal computation

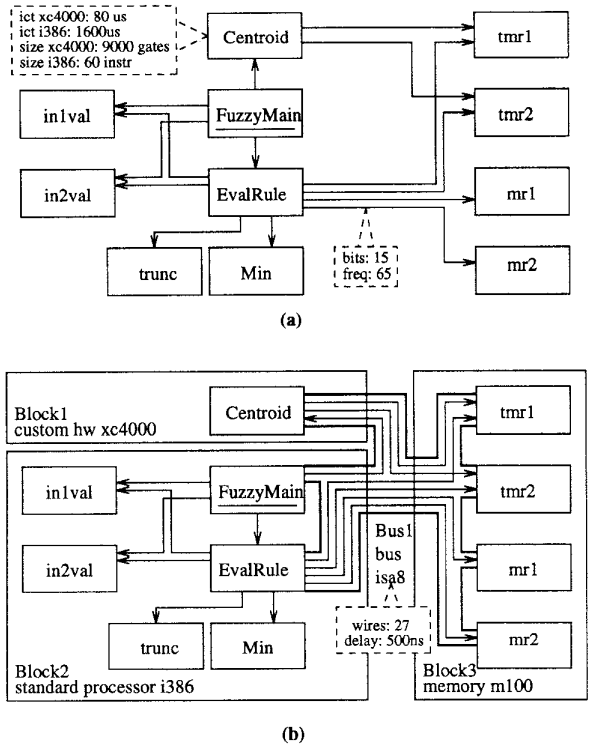

Fig. 3: SLIF model: (a) access graph, (b) partitioned.

time (ict) annotation, indicating the start-to-finish computation time of the node excluding time for accesses to other nodes. In the example, *Centroid* executes in 80 microseconds on an XC4000, and 1600 microseconds on an Intel386. A variable's ict represents the time to fetch the data from its (usually memory) component. Fourth, each non-hierarchical edge may have bits and frequency annotations. The bits value represents the number of data bits that must be transferred during each access. Frequency represents the average number of accesses over the edge during an execution of the edge's source node. In the example, the access of *EvalRule* to *mr2* requires a transfer of 15 bits (address and data), and the access is made 65 times on the average. Finally, each edge may have a wires annotation, indicating the number of physical wires. In the example, *Bus1* has a width of 27 wires, which is the number of wires involved for an 8-bit ISA bus (including address, data and control).

Annotations are determined using estimators, synthesis tools, or manually. For all cases where average values are used, we could also associate minimum and maximum values.

Estimations of metric values are determined directly from the annotations. For example, the size metric (instructions) for a processor node is determined as the sum of the sizes of the children nodes. Other metrics involve more complex combinations of annotations; for details, we refer the reader to [19]. We also point out that annotations can be much more complex than just numbers, to account for interaction between nodes when implemented on the same component. For example, complex annotations for performing hardware size estimation while accounting for hardware sharing are described in [20].

Note that annotations provide a simple means to override an estimation; the user just changes the annotation. They also provide a means for partitioning in the absence of a complete specification; the user can create an empty procedure, and then provide necessary annotations manually.

4 Generating generic model instances

4.1 Overview

A generic model is one generated from statistical data rather than from a particular specification. It does not represent a "real" example, yet has characteristics similar to those of a real example. A generic model, of nearly any size, can be generated automatically in a matter of seconds. A real example, on the other hand, may require weeks or months to develop. Generic models therefore provide a means to quickly create a large number of examples on which we can compare partitioning heuristics. In addition, we can generate generic models of varying sizes, to examine how a heuristic's performance is affected by the problem size.

We chose six examples from which to obtain statistical data, which would later be used to generate generic models. The examples were: (1) *ans*, a telephone answering machine controller characterized by many small procedures and loose timing constraints; (2) *ether*, an Ethernet coprocessor characterized by multiple processes and tight timing constraints, with much data manipulation; (3) *fuzzy*, a fuzzy controller with large arrays and a single timing constraint, with much data computation; (4) *itv*, an interactive television processor with some tightly-constrained processes and many large, loosely constrained procedures; (5) *mwt*, a microwave transmitter controller with several levels of procedure call nesting and very loose constraints; and (6) *vol*, a volume-measuring medical instrument controller characterized by a sequence of procedures with a single timing constraint, with some data computation. The examples represent a mix of control and control/data systems, with a variety of sizes ranging from 222 to 1021 lines of VHDL, and 31 to 124 SLIF nodes. Each example required an average of roughly 1 person-week to develop in an untested (draft) form, and another week in a tested form. Three of the examples are proprietary.

4.2 Statistical method

We obtained statistical data from the above examples. This data would be used to generate the structure of a generic SLIF access graph, i.e., the nodes and edges. It would also be used to generate the annotations of node ict and size, and edge frequency and bits. We also sought to find correlations among the annotations, in particular between a node's ict value for a hardware implementation and its ict value for a software implementation; likewise for a node's size in hardware and software.

We now provide some brief background of the statistical techniques used in this paper. Given a large number of data points to be plotted on x and y axes, we can summarize the data by choosing intervals along the x-axis and creating a histogram, as shown in Figure 4. The histogram illustrates the relative frequencies of set of observations on a range of values. It records the percentage or fraction of data in a particular interval relative to the other intervals. To display the shape of the distribution without the lumpiness of the histogram, we use a smooth curve to approximate the histogram, as shown in Figure 4. This is the probability function that describes the distribution.

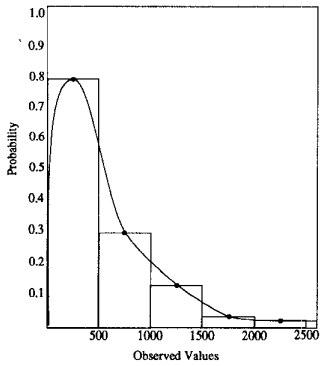

Fig. 4: Distribution of probability histogram

We use a simple linear regression in parts of our generation procedure. Such a regression has one explanatory variable and one response variable. The explanatory variables can be a node's hardware size or internal computation time, whereas the response variable can be the size of node's software size or internal computation time. Mathematically, a simple linear regression equation expresses the relationship of these two variables, having the form:

$$y = bx + a$$

The method for calculating a linear regression equation by the least-square regression method can be found in any introductory statistics textbook. Note that the regression equation does not describe the nature of association between explanatory and response variables.

We use a correlation coefficient to measure the strength and direction of the linear association of two variables. Two variables have positive association when the above-average values of one variable accompany the above-average values of other variable; otherwise, they have a negative association. The range of a correlation coefficient is between -1 and 1, where 1 is perfect positive association, and -1 is perfect dissociation.

4.3 Access graph generation

We now describe how we generate model instances from the statistical data. We start by generating an unannotated access graph. The user specifies the number of nodes n, and we must then generate some number of edges e that connect the nodes. One might consider computing e as a multiple of n (our data showed this multiple to be 1.2), and then randomly selecting two nodes for each edge, ensuring that no cycles are created. However, such a graph does not reflect the real examples' characteristic that some nodes have many outgoing edges (such as the main process node), and most nodes have just a few incoming edges.

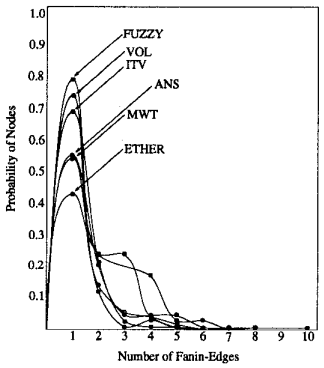

Fig. 5: Distribution of nodes and fanin for real examples

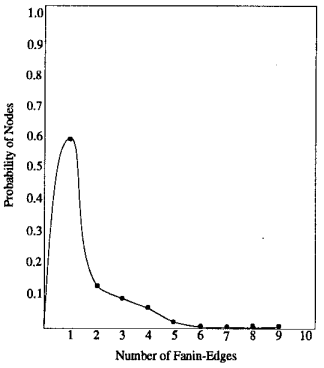

Fig. 6: Distribution of nodes and fanin for generated examples

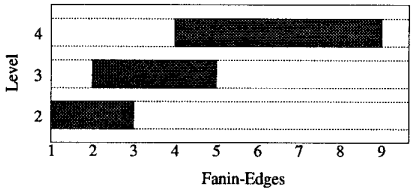

Fig. 7: Range of fanin and level

For a more realistic graph structure, we measured the fanin of each node. The *fanin* of a node is the number of incoming edges. Figure 5 shows the probability distribution of fanin for the real examples. The distribution shows that most nodes have a fanin of 1 or 2. For example, 80% of *fuzzy*'s nodes and 40% of *ether*'s nodes have a fanin of

1. Between 10% and 30% of all nodes have a fanin of 2, and only a small percentage of nodes have a fanin of 3 or greater.

From the real examples, we generate the average percentage of nodes that have each possible fanin value, as shown in Figure 6. For example, the figure shows that 60% of nodes have a fanin of 1, 13% have a fanin of 2, 10% have a fanin of 3, etc.

We then noted that a node's fanin was related to its *level* in the access graph. The *level* of a node is the maximum number of edges that must be traversed to reach the node from a root node (i.e., a node with no incoming edges), plus 1. In the real examples, a node with a high level typically had higher fanin. These higher-fanin nodes represented commonly-accessed procedures or variables. Figure 7 shows the relationship between fanin and level for the real examples. Level 2 nodes had fanins between 1 and 3, level 3 nodes between 2 and 5, and level 4 nodes between 4 and 9.

Using the fanin information and level data described above, we can generate the edges of the generic model instance as follows. First, we assign a fanin for each node, such that the percentage of nodes with a particular fanin matches the distribution of Figure 6. For example, 60% of nodes will be assigned a fanin of 1. Second, we assign each node to a level based on its fanin. For example, a node with a fanin of 3 could be assigned to level 2 or level 3, as shown in Figure 7. Third, we create a number of incoming edges for each node equal to the fanin; each such edge originates from a randomly-selected node of a lower level.

4.4 Annotation generation

Having generated a realistic unannotated access graph, we turn our attention to generating annotations for the nodes and edges of the graph.

4.4.1 Internal computation time

We consider generating an internal computation time (ict) value for each node. We first determined each real example's node hardware ict value, i.e., the ict for a custom hardware implementation of the node, determined using the method in [19]. The distribution of ict values is shown in Figure 8, where we plot the probability that a node has a particular ict value (using intervals of 5 clocks). The figure shows that between 60% and 90% of nodes had an ict between 1 and 5 clock cycles (excluding the *fuzzy* example, whose values were much higher). Most ict values were between 1 and 25. For each generated SLIF model, we randomly choose one ict distribution from the real examples. From the chosen ict distribution, we generate the ict annotations for each node by randomly choosing a value from the ict interval with respect to the probability of the distribution. This gives annotations that reflect the real examples. Figure 9 shows the ict value distribution for five generated examples, containing 20, 40, 60, 80 and 100 nodes, respectively. (We tried randomly selecting an equation for each node, rather than each model, but this led to poor results).

We then must generate a node's software ict value. We could generate a node's software ict value using an approach similar to the method used to generate the hardware ict value. However, such an approach would ignore the observed positive correlation between a node's hardware ict and software ict values. Such a correlation is intuitive, since a node representing a complex computation would typically need more time in both hardware and software than other simpler nodes. For example, Figure 10 shows hardware versus software ict values for the *ether* example. Note the strong correlation. We have drawn the regression line, which is described as the function $ict(hardware) = 0.228 + 0.022 * ict(software)$. The regression equations and correlation coefficients for all six real examples are shown in Table 1. Given a node, we generate a software ict value from the hardware ict value by selecting the equation from the same example that we used to generate the ict of hardware above.

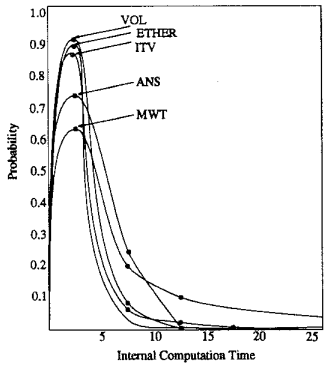

Fig. 8: Hardware ict distribution for real examples

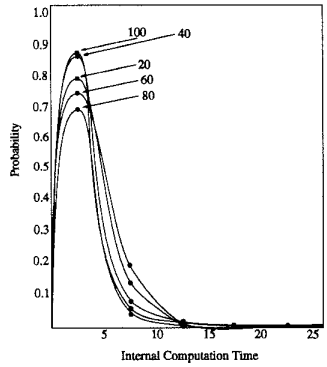

Fig. 9: Hardware ict distribution for generated examples

4.4.2 Size

We now consider generating a size value for each node. We use the same approach as used for ict. Specifically, we first examine the distribution of hardware sizes for the nodes of the real examples, where node size is determined as described in [19, 20]. The distribution is shown in Figure 11 with intervals of 500 gates. Node sizes ranged between

Ex	Nodes	Regression equation	Corr.
ANS	44	ict_hw = 1.12 + 0.0404 ict_sw	+0.58
ETHER	124	ict_hw = 0.228 + 0.0220 ict_sw	+0.92
FUZZY	69	ict_hw = 0.24 + 0.0191 ict_sw	+0.98
ITV	84	ict_hw = -0.070 + 0.0406 ict_sw	+0.89
MWT	31	ict_hw = 0.462 + 0.0188 ict_sw	+0.83
VOL	38	ict_hw = 0.245 + 0.0330 ict_sw	+0.78

Table 1: Internal comp. time regression equations

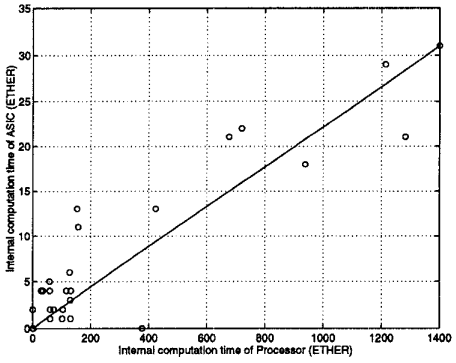

Fig. 10: Regression line of ict for ether example

1 and 2500 gates. Similar to the ict generation above, for each generated model SLIF, we randomly select one size distribution from the real examples. From the chosen size distribution, we generate the size annotations for each node by randomly choosing a value from the size interval with respect to the probability of distribution. The distribution for five generated examples is shown in Figure 12.

As was the case with ict values, there is a strong positive correlation between a node's hardware size and software size in the real examples. Figure 13 plots hardware versus software size for the *ether* example, and shows the strong correlation. Once again, we determine regression equations for the real examples, as shown in Table 2, and select for each SLIF model the equation corresponding to the example of the chosen hardware size distribution.

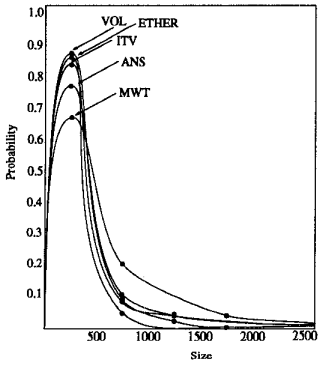

Fig. 11: Distribution of hardware size for real examples

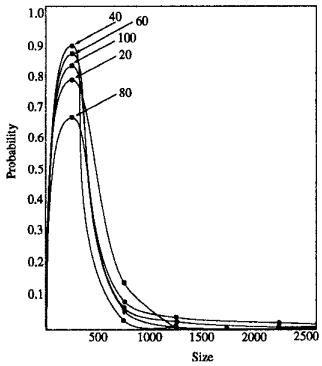

Fig. 12: Distribution of hardware size for generated examples

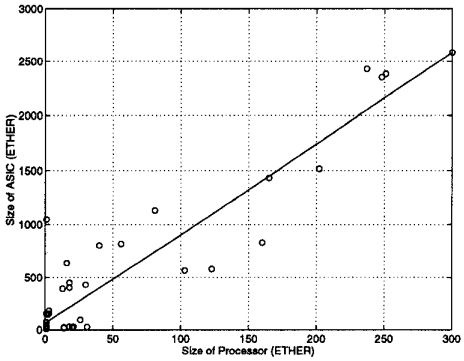

Fig. 13: Regression line of size for ether example

4.4.3 Edge bits, frequency and wires

We now turn our attention to the edge annotations of bits, frequency and wires. We determined the range of values of edge bits from the real examples. For each generated edge, we randomly select a bits value from that range, assuming a uniform distribution. Frequency and wires values are generated similarly.

4.4.4 Execution-time constraints

In addition to creating the annotated SLIF, we also generate execution-time constraints. The user provides a range for the number of such constraints. For each constraint,

Ex	Nodes	Regression equation	Corr.
ANS	44	size_hw = 39.1 + 16.4 size_sw	+0.70
ETHER	124	size_hw = 61.9 + 8.4 size_sw	+0.93
FUZZY	69	size_hw = 729.0 + 0.862 size_sw	+0.35
ITV	84	size_hw = 28.4 + 8.06 size_sw	+0.71
MWT	31	size_hw = 7.2 + 11.8 size_sw	+0.94
VOL	38	size_hw = 63.3 + 9.36 size_sw	+0.69

Table 2: Size regression equations

we randomly select a node; this node will have its execution time constrained. We then determine the execution time for this node when all nodes are partitioned into one custom hardware block. This execution time is the sum of the internal computation time and the communication time, where communication time is the time to transfer data to/from accessed nodes plus the execution time of those accessed nodes, as described in [19]. We use this execution time, times a small factor, as the constraint value.

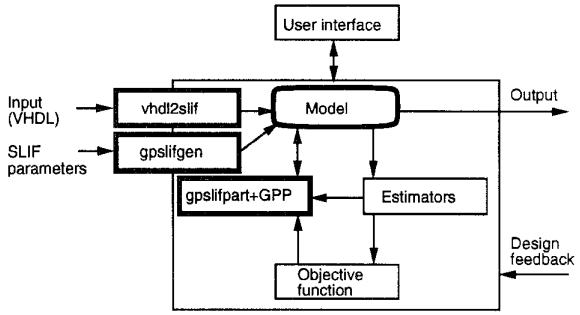

Fig. 14: Tools for SLIF creation and partitioning

5 Implementation and experiments

We have implemented several tools to support the creation and partitioning of SLIF models, as illustrated in Figure 14. First, we have developed a VHDL to SLIF converter, `vhdl2slif`, which reads a VHDL specification, and then builds and outputs a textual SLIF model. Second, we have developed a generic SLIF generator, `gpslifgen`, which outputs a generic textual SLIF model whose arbitrary size is user-specified. Third, we have developed a SLIF partitioning tool, `gpslifpart`, which partitions the SLIF among a set of user-provided components, given a set of user-specified constraints. This tool uses the *GPP* (General Purpose Partitioner) engine to perform the partitioning. GPP currently provides several heuristics, including a greedy heuristic, group migration, simulated annealing, genetic evolution, and hierarchical clustering. New heuristics can be easily incorporated into GPP. All of these tools are available via ftp at cs.ucr.edu.

Examples	Random		Greedy		GM		SA	
	C	T	C	T	C	T	C	T
ANS	6911	0	10	2	0	7	0	34
ETHER	3985	1	279	7	151	516	151	448
FUZZY	6998	0	31	7	41	101	19	208
ITV	1658	0	100	6	75	144	75	501
MWT	1028	0	16	2	10	18	10	337
VOL	719	0	319	2	224	39	202	453

Table 3: Hardware/software partitioning of real examples

We have performed two experiments to demonstrate the usefulness of the SLIF model, gpslifgen and gpslifpart. In the first experiment, we compared four partitioning heuristics on their ability to perform a hardware/software partitioning of the six real examples described above, among an Intel 8086 processor and a Xilinx 4000 FPGA. For each heuristic, we measured heuristic CPU time and output quality (as measured using the violation-minimizing objective function described in Section 2). The four heuristics were: *Random*, a random partitioning, used as an initial partition and thus providing a reference to see how much cost reduction other heuristics obtained; *Greedy*, a greedy improvement heuristic that accepts cost decreasing moves of a node from hardware to software, or vice-versa, until no cost-decreasing move can be found; *GM*, a group migration heuristic [21] that uses the Kernighan/Lin control strategy [22] to overcome local minima; *SA*, a simulated annealing implementation using the parameters described in [3]. Results are shown in Table 3.

Examples	Random		Greedy		GM		SA	
	C	T	C	T	C	T	C	T
30	3575	0	146	2	76	13	11	60
40	8557	0	78	2	61	125	21	156
50	877	0	130	2	0	8	0	26
60	1477	0	58	3	0	49	0	37
70	12110	0	68	5	49	374	52	178
80	10022	1	13	4	0	26	0	47
90	363	1	45	9	4	459	0	266
100	3901	1	7	9	0	115	0	110

Table 4: Hardware/software part. of generated examples

We then generated 8 generic examples ranging in size from 30 to 100 nodes, and applied the same four heuristics. Results are shown in Table 4. The results show that the relative performance of the various heuristics is approximately the same for the real and generated examples. Specifically, *greedy* is fast but usually yields inferior partitions; GM yields partitions close to those of SA for most examples in less time, but sometimes yields much worse partitions or uses more time. Though both the real and generated examples enable us to evaluate the heuristics, the examples were created with a very different amount of effort: only a few minutes for the generated examples, but nearly two months for the real examples.

We performed a second experiment, similar to the first, in which we performed a hardware/hardware partitioning of the real and generated examples. Results are shown in Table 5 and Table 6. Once again, results show that relative performance of the heuristics for the real examples is approximately the same as for the generated examples.

Examples	Random		Greedy		GM		SA	
	C	T	C	T	C	T	C	T
ANS	210	0	210	2	210	38	205	283
ETHER	356	0	64	10	64	369	53	531
FUZZY	487	0	423	5	404	161	404	690
ITV	704	0	180	6	130	186	138	629
MWT	895	0	6	2	3	14	0	139
VOL	881	0	411	3	411	28	398	476

Table 5: 2-way hardware partitioning of real examples

At this point, we mention some limitations of our model and techniques. First, the model does not currently support scheduling of the nodes. Second, the generation techniques are based on only six examples; a possibly very useful tool would be one that creates the statistical data

Examples	Random		Greedy		GM		SA	
	C	T	C	T	C	T	C	T
30	708	0	198	1	103	9	16	110
40	1114	0	73	1	0	17	0	3
50	916	0	210	2	136	63	63	209
60	446	0	52	5	0	140	0	122
70	1175	0	319	6	128	229	101	352
80	550	0	154	4	2	741	0	102
90	886	1	264	9	142	295	146	263
100	732	1	175	6	7	253	0	154

Table 6: 2-way hw. partitioning of generated examples

for any given set of examples, and then updates `gpslifgen` to generate examples with the characteristics of that data. We might even want to look at a large number of real examples to determine if there exists a set of equivalence classes into which examples could be classified.

6 Conclusion

We have described SLIF, a model suitable for functionally partitioning a specification among hardware and/or software components. The model, unlike a dataflow model, is well-suited for procedural specifications. We have detailed a technique for generating arbitrary-sized general model instances having characteristics similar to real examples. The model and its supporting tools enable fair comparison of functional partitioning heuristics, and such comparison will become more important as researchers begin to focus on developing such heuristics. The supporting tools provide the infrastructure for developing and comparing heuristics, so may greatly reduce the time needed by researchers for such development and comparison.

With the existing models and tools, we plan to investigate heuristics for partitioning. In particular, we plan to evaluate popular heuristics such as Kernighan/Lin, in order to make control strategy improvements specifically for functional partitioning. We also plan to investigate the effects of SLIF transformations on partitioning results. Such transformations include procedure cloning to reduce chip I/O, and process splitting to improve performance.

References

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

[2] H. Weng, "A study on subprogram synthesis and evaluation," Master's thesis, University of California, Riverside., September 1995.

[3] T. Le, "Experiments on functional partitioning for packaging constraints and synthesis tool performance," Master's thesis, University of California, Riverside., December 1995.

[4] A. Kalavade and E. Lee, "A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem," in *International Workshop on Hardware-Software Co-Design*, pp. 42–48, 1994.

[5] F. Vahid, J. Gong, and D. Gajski, "A binary-constraint search algorithm for minimizing hardware during hardware-software partitioning," in *Proceedings of the European Design Automation Conference (EuroDAC)*, pp. 214–219, 1994.

[6] X. Xiong, E. Barros, and W. Rosentiel, "A method for partitioning UNITY language in hardware and software," in *Proceedings of the European Design Automation Conference (EuroDAC)*, 1994.

[7] Z. Peng and K. Kuchcinski, "An algorithm for partitioning of application specific systems," in *Proceedings of the European Conference on Design Automation (EDAC)*, pp. 316–321, 1993.

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

[9] Y. Chen, Y. Hsu, and C. King, "MULTIPAR: Behavioral partition for synthesizing multiprocessor architectures," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 2, pp. 21–32, March 1994.

[10] R. Gupta and G. DeMicheli, "Partitioning of functional models of synchronous digital systems," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 216–219, 1990.

[11] C. Gebotys, "An optimization approach to the synthesis of multichip architectures," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 2, no. 1, pp. 11–20, 1994.

[12] F. Vahid and D.Gajski, "Specification partitioning for system design," in *Proceedings of the Design Automation Conference*, pp. 219–224, 1992.

[13] P. Athanas and H. Silverman, "Processor reconfiguration through instruction-set metamorphosis," *IEEE Computer*, vol. 26, pp. 11–18, March 1993.

[14] T. Ismail, K. O'Brien, and A. Jerraya, "Interactive system-level partitioning with Partif," in *Proceedings of the European Conference on Design Automation (EDAC)*, 1994.

[15] K. Kucukcakar and A. Parker, "CHOP: A constraint-driven system-level partitioner," in *Proceedings of the Design Automation Conference*, pp. 514–519, 1991.

[16] S. Antoniazzi, A. Balboni, W. Fornaciari, and D. Sciuto, "A methodology for control-dominated systems codesign," in *International Workshop on Hardware-Software Co-Design*, pp. 2–9, 1994.

[17] 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.

[18] F. Vahid, "Procedure exlining: A transformation for improved system and behavioral synthesis," in *International Symposium on System Synthesis*, pp. 84–89, 1995.

[19] F. Vahid and D. Gajski, "SLIF: A specification-level intermediate format for system design," in *Proceedings of the European Design and Test Conference (EDTC)*, pp. 185–189, 1995.

[20] F. Vahid and D. Gajski, "Incremental hardware estimation during hardware/software functional partitioning," in *IEEE Transactions on Very Large Scale Integration Systems*, pp. 459–464, 1995.

[21] D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and design of embedded systems.* New Jersey: Prentice Hall, 1994.

[22] B. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, February 1970.