SLIF: A specification-level intermediate format for system design

Frank Vahid

Technical Report CS-94-06
September 20, 1994

Department of Computer Science
University of California
Riverside, CA 92521
vahid@cs.ucr.edu

Abstract

As methodologies and tools for chip-level design mature, design effort becomes focused on increasingly higher levels of abstraction. Presently much effort is focused on the system level of design, where the key design tasks include system component allocation, functional partitioning and transformation, and coarse estimation. However, the commonly used internal formats of functionality, such as the control-dataflow graph, are too fine-grained for the system level. We introduce a new, more abstract internal format, and we describe how it enables estimations of design metrics in an order of magnitude less time and memory, as well as enabling truly practical designer interaction. The format serves as the core of the SpecSyn system design environment, and it can be extended to handle a large scope of new and evolving system design problems.

Contents

1 Introduction **1**

2 SLIF definition **2**
2.1 Requirements of an internal format 2
2.2 Basic format 3
2.3 Annotations for high-level concurrency 6
2.4 Annotations for estimations 6
2.4.1 Performance 7
2.4.2 I/O 8
2.4.3 Size 8
2.5 SLIF definition with annotations 9

3 Estimation of quality metrics **10**
3.1 Execution time 10
3.2 Bitrate 11
3.3 Software, hardware and memory size 12
3.4 I/O 12

4 Related work **13**

5 Results **13**

6 Conclusions and future work **15**

List of Figures

1 Partial VHDL specification for a fuzzy-logic controller system 4
2 Basic SLIF-AG for the example system 5
3 Partial SLIF with system components and with some annotations 6
4 Results of building SLIF and obtaining estimations 14

.

1 Introduction

Development of behavioral synthesis tools, which help a designer convert a functional specification into a register-transfer design, have begun to mature. As a result of this maturation, new research efforts have evolved that focus on design support at an even earlier stage of design. We refer to such a stage as the system-level of design.

System design consists of several tasks that must be solved before behavioral synthesis or software development can take place. One task is the allocation of system components, such as processors, ASICs, memories and buses, to the design. A second task is the partitioning of the functional specification among those components. A third task is the transformation of the specification into one more suited for synthesis, such as merging processes into a single process for implementation with a single controller. The goal of these three tasks is to create a design that satisfies constraints on design metrics, such as performance, size, pins, design time, modifiability, and cost. The tasks are highly interdependent, and there is no single approach that leads to the best solution; one might try performing one task at a time, or one might try to intertwine them. Regardless of the approach taken, all three of the tasks rely on information provided by a fourth task: the rapid, coarse estimation of design metrics, including execution time, bitrates, ASIC size and pins, and memory size, which in turn determine the goodness of a particular allocation, partitioning or transformation decision. After these tasks have been completed, the resulting output is a set of interconnected system components, each functionally specified. ASIC components can then be implemented through behavioral synthesis, and processor components through software design.

In current practice, system design makes up a gray area of design methodology. The tasks are not well-defined, decisions are based on mental or hand-calculated estimations, and documentation of decisions is scarce. These factors lead designers to overdesign systems in order to ensure that performance constraints will be met, and they also lead to many component integration problems resulting from ambiguous component specifications. The shortcomings of current practice have led many researchers to propose a formalized top-down approach that starts from a simulatable functional specification that is then partitioned among system components with the aid of tools. All such approaches first read the functional specification into an internal format, and then apply system design tasks to that internal format.

In this paper, we describe the SLIF internal format, which was developed specifically to support system-design tasks and is used by the SpecSyn system-design tool. SLIF is unique, compared with other internal forms, for three reasons. First, SLIF represents the functional specification at a coarse level of granularity, presenting an abstraction of the specification that is appropriate for designer interaction, for tractable algorithm computation, and for fast estimation due to extensive preprocessing (as we shall see later). Second, estimation of design metrics can be performed entirely from SLIF, because SLIF represents not only functionality, but also the mapping of that functionality to a variety of system component types. Third, SLIF is oriented around accesses between pieces of the specification, rather

than around control and data dependencies between those pieces. The former orientation is more efficient for the system design problems of allocation, partitioning, and transformation, while the latter is better suited for fine-grained scheduling problems.

The paper is organized as follows. In Section 2, we provide a definition of SLIF, including the preprocessed annotations for estimation, along with a simple example. In Section 3, we demonstrate the usefulness of SLIF for obtaining rapid estimations, by defining equations for several design metrics based on the preprocessed annotations. In Section 4, we discuss related research. In Section 5, we summarize the time required build SLIF and to estimate from it for several examples. In Section 6, we provide conclusions.

2 SLIF definition

2.1 Requirements of an internal format

The key measures of an internal format's usefulness are the ability to define system design problems in terms of the format, and the ability to obtain rapid estimates from the format. The ability to define problems in terms of the format requires that the format be at the appropriate level of granularity. For example, during system design we wish to map coarse-level functions, such as an algorithm, to system components. For this problem, a format with the granularity of arithmetic operations is too detailed. Instead, a format at the granularity of coarse-level functions is more appropriate.

Likewise, the ability to obtain rapid estimates requires that as much estimation information as possible be determined before system design tasks are applied. For example, during system design we may wish to obtain rapid program memory estimates for a particular mapping of procedures to a processor. If we take the most accurate approach of compiling that set of procedures into the processor's instruction set, we suffer from long delays to obtain the estimate. Such delays are especially important when we apply automated partitioning algorithms that examine thousands of possible partitions. On the other hand, we can take a faster approach in which we initially compile each procedure into the processor's instruction set, before beginning system design. Then, for a given set of procedures mapped to the processor, we can simply add the number of instructions for each procedure to estimate the total number of instructions. While such an estimate will be slightly inaccurate due to the fact that inter-procedural optimizations are not considered, the loss in accuracy is acceptable when one considers the substantial gain in estimation speed. This example illustrates the requirement of a good internal format to be able to represent preprocessed estimation information to obtain speed without losing excessive accuracy.

In order to introduce SLIF in an easy to understand manner, we will first introduce a very basic version of SLIF that merely demonstrates the overall organization of the format, and we then proceed to annotate this simple version with information used for estimations.

2.2 Basic format

Our internal format consists of two parts: functional objects and system components. The functional objects represent functionality, while the system components represent implementation structure. Since the goal of system design is to map functional objects to system components, the two parts are tightly linked.

The functional objects are of system-level granularity: processes, procedures, variables and communication channels. For this reason, we refer to the format as the **System-Level Intermediate Format**, or **SLIF**. The functional objects are represented in SLIF as a directed graph. Each node of the graph represents a behavior or a variable from the specification. A behavior is a process or procedure in the specification; finer granularity can be obtained by treating basic blocks as procedures. A directed edge, or **channel**, represents an access by the source behavior to another behavior (a subroutine call) or to a variable or external port (a data read or write). It may also represent data being transferred from the source behavior to another behavior as specified by a message passing construct in the input specification. Because this graph is oriented around the various accesses among functional objects, we refer to it as an **access graph**, or **AG**.

As a simple example, consider the partial specification of a fuzzy-logic controller shown in Figure 1. Two inputs, *in1* and *in2*, must be converted to a single output, *out1*, using the rules of fuzzy logic. Such controllers are common in numerous consumer applications, such as video camera focus control, thermostat control, and automobile cruise control, in which smooth transitions are needed from one output value to another. The main process *FuzzyMain* first samples the values on the inputs *in1* and *in2* by writing them into variables *in1val* and *in2val*. It then calls a procedure *EvaluateRule* twice, once for each input, and that procedure fills the values of an array (*tmr1* or *tmr2*) based on the input value and on the values contained in another predefined array (*mr1* or *mr2*). After other tasks are performed on the new *tmr* arrays, including convolution (other tasks are omitted for brevity), a centroid value is computed, and that value is then output. The process then repeats after a specified time interval.

The basic SLIF-AG representation of this specification is shown in Figure 2. Each process, procedure and variable has its own node. Each variable access has its own edge; for example, the writing of *in1val* in *FuzzyMain* translates to an edge between those two nodes, while the two calls of *EvaluateRule* by *FuzzyMain* also translate to a single edge between those two nodes.

We immediately note several unique features of the SLIF-AG compared with other formats. First, the format does not describe the control or dataflow of the system, but instead describes the accesses. Note that the direction of the edge represents the initiator of the access, rather than the flow of data (an interesting implication is that a cycle would represent recursion). In this sense, it is very much like a call-graph commonly used for software profiling, with variables included in addition to procedures, rather than a control-dataflow graph used by high-level synthesis. Second, the contents of the behavior nodes are left unspecified. Instead, we will later derive abstractions of those contents for estimation

```
entity FuzzyControllerE is
  port ( in1, in2 : in integer;  out1:  out integer );
end;
...
FuzzyMain: process
        variable in1val, in2val : integer;
        type mr_array is array (1 to 384) of integer;
        variable mr1, mr2: mr_array; -- membership rules
        type tmr_array is array (1 to 128) of integer;
        variable tmr1, tmr2: tmr_array; -- truncated memb. rules
        ...
begin
        in1val := in1;  in2val := in2;
        EvaluateRule(1);
        EvaluateRule(2);
        Convolve;
        out1 <= ComputeCentroid;
        wait until ...
end process;


procedure EvaluateRule(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;
```

Figure 1: Partial VHDL specification for a fuzzy-logic controller system

purposes. Third, note that a variable is treated like a procedure, so an access to a variable is treated the same as a call to a special procedure that reads or writes that variable. Finally, note that the SLIF-AG is an abstraction of the input specification, suitable for system design tasks, but not intended for simulation.

In addition to representing the functional objects in the form of an AG, SLIF also represents three basic types of structural system components: processors, memories and buses. A processor, which is either a standard component or a custom-designed controller/datapath, can be used to implement behaviors and variables. A memory can implement a set of variables. A bus can implement a set of channels. For example, Figure 3 shows a partition of several of the previous example's nodes among two memories, an ASIC, a processor and a bus.

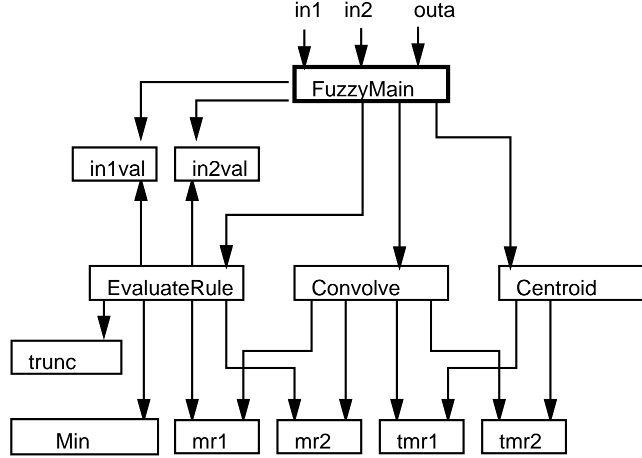

Figure 2: Basic SLIF-AG for the example system

We can now formally define the basic SLIF format as a sextuple:

$$< BV_{all}, IO_{all}, C_{all}, P_{all}, M_{all}, I_{all} >$$

In other words, the basic SLIF consists of a set of behaviors and variables, of input and output ports, of channels, of processors, of memories, and of buses.

The set of behaviors and variables BV_{all} is defined as: $B_{all} \bigcup V_{all}$, where $B_{all} = \{b_1, b_2, ...\}$ is the set of all behaviors, and $V_{all} = \{v_1, v_2, ...\}$ is the set of all variables. The set of input and output ports IO_{all} is defined as: $\{io_1, io_2, ...\}$. The set of all communication channels C_{all} is defined as: $\{c_1, c_2, ...\}$, where $c_i =< src, dst >$, $src \in B_{all}$ and $dst \in BV_{all} \bigcup IO_{all}$. In other words, each channel describes the access by an accessor behavior (src) to a behavior, variable or port (dst). The sets $BV_{all}, IO_{all}, C_{all}$ comprise the functional objects that are derived from the specification.

The set of buses I_{all} is defined as: $\{i_1, i_2, ...\}$, where $i_k =< C >$, $C \subset C_{all}$. In other words, each bus consists of a set of channels. The set of memories M_{all} is defined as: $\{m_1, m_2, ...\}$, where $m_k =< V >$, $V \subset V_{all}$. In other words, each memory consists of a set of variables. The set of (standard or custom ASIC) processors P_{all} is defined as: $\{p_1, p_2, ...\}$, where $p_k =< BV >$, $BV \subset BV_{all}$. In other words, each processor consists of a set of behaviors and variables. The three sets $I_{all}, M_{all}, P_{all}$ comprise the structural objects to which functional objects must be mapped.

A proper **partition** of functional objects among the system components requires that:

- $i_1.C \cup i_2.C \cup ... = C_{all}$, $i_j.C \cap i_k.C = \emptyset$ for all $j, k, j \neq k$,
- $m_j.V \cap m_k.V = \emptyset$ for all $j, k, j \neq k$,

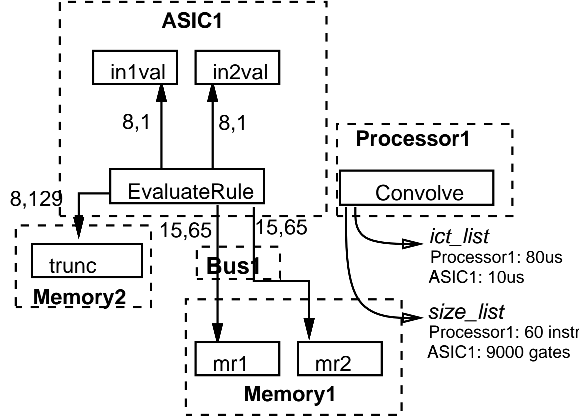

Figure 3: Partial SLIF with system components and with some annotations

- $p_1.BV \cup ...p_n.BV \cup m_1.V \cup ...m_o.V = BV_{all}$, $p_j.BV \cap p_k.BV = \emptyset$ for all $j, k, j \neq k$.

In other words, a partition is a mapping of channels to buses, of behaviors to processors, and of variables to either processors or memories, such that each functional object is mapped to exactly one system component.

2.3 Annotations for high-level concurrency

A common feature of many systems, but one not found in the above example, is that of high-level concurrency. One form of high-level concurrency is that of concurrent processes, as found in VHDL, where each process is essentially a program that repeats forever. To represent such concurrency information in SLIF, we specially mark each AG node that corresponds to a process. For example, the *FuzzyMain* node of Figure 2 is shown in bold, indicating that it is a process node.

A second form of high-level concurrency is that created by a fork/join construct, as in Verilog, in which multiple procedures are called simultaneously during execution of a process. To describe such concurrency in SLIF, we associate a concurrency tag with each channel. Same-source channels with the same tag could be accessed concurrently. Such tags are also used to denote concurrently-accessible variables.

2.4 Annotations for estimations

The basic form of SLIF described above presents an abstraction of the specification that enables us to define the system design problem as a set of partitioning problems. However, it does not yet provide the information we need to obtain rapid estimations of various design metrics, such as area, pins and performance, for a given partition among system components. In other words, we need information that enables us to determine if a particular partition

is good. We shall now annotate SLIF with such information. In the following section, we demonstrate how such information can be used to rapidly compute design metrics.

2.4.1 Performance

For performance measurement, we need to know how many times a behavior accesses each channel. Therefore, we associate an access frequency weight **accfreq** with each edge, which indicates the number of times the access occurs during an average start-to-finish execution of the source behavior, as determined from a branch probability file. The branch probability file may be obtained manually or through profiling. We can also associate two other weights with each edge, corresponding to a maximum number and a minimum number of accesses. In the remainder of this paper, we shall only refer to average performance metrics, omiting the simple extensions for maximum and minimum performance for the sake of brevity.

We also need to know how many bits are transferred during each channel access, since the physical bus to which a channel is mapped may have fewer wires than the number of bits being transferred, which in turn would require breaking the transfer into several smaller transfers. Thus, we associate a **bits** weight with each edge, which indicates the number of bits transferred during an access. For access to a scalar, this is the number of bits into which the scalar would be encoded. For access to an array of scalar elements, this is the number of bits to encode an array element, plus the number of address bits needed to specify an element's address. For a more complex data item, such as a three-dimensional array, the data item is first transformed to an array of scalars. For access to another behavior, this is the number of bits needed to transfer all (if any) parameters, where the number of bits for each parameter is computed as for variables. For a message pass, this is the number of bits into which the message would be encoded.

Performance measurement also requires that we know the average execution time of each behavior. We can view execution time of a behavior as the sum of two parts: the time spent communicating over channels, and the time spent performing all other computations (**internal computation time**, or **ict**). We shall see later that communication time can be derived from the channel information that we added above, from the transfer time of the bus (described in the next paragraph), and from the execution time of other accessed behaviors. Since communication time is known, to determine execution time we merely need to annotate each behavior node with its ict. Since a behavior may execute at a different speed on different components, we annotate each behavior node with a list of ict weights, one weight for each type of system component on which that node could possibly be implemented. The ict of a behavior on a custom hardware component (such as a standard cell ASIC or an FGPA) can be estimated by synthesizing the behavior to a structure using that particular component's technology. Since there are many possible implementations, the designer may need to guide this step closely. Alternatively, the designer may simply specify an ict without going through the synthesis step. In a similar manner, the ict on a standard processor can be estimated through compilation. Finally, the ict of a variable node is the time to read or write the storage in which the variable is implemented. Each

variable node will thus also have a list of ict weights, one for each memory type or processor type on which it might be implemented.

We need to know the time it takes to transfer data over a particular bus. We associate two times with each bus. The **ts** (time-same) is the time to transfer data entirely within the same system component. The **td** (time-different) is the time to transfer data between two different system components. The td is usually larger than the ts. We could have permitted a more extensive set of annotations, where there would be a unique ts value for each component type, and a unique td value for each possible pair of component types, but we have not yet explored this possibility.

Finally, to obtain good performance measurement, we need to know which channels must be accessed sequentially and which could be accessed concurrently. Recall that the possibly concurrent behavior access of a fork/join construct has already been represented in SLIF by associating a tag with each channel, where channels with the same tag could be accessed concurrently. We are left with the job of determining which of the variables and procedures accessed by a behavior could be accessed concurrently. Such information can be estimated by scheduling the contents of the behavior. Such scheduling was also necessary in the previous paragraph when we mentioned that the internal computation time of a behavior could be obtained through synthesis, a part of which requires scheduling. We therefore create the channel tags from that schedule.

2.4.2 I/O

To measure the input/output (I/O) pins required on components for a given partition of behaviors and variables, bus width information is necessary. Thus, we associate a **bit-width** weight with each bus component. This weight corresponds to the number of physical wires of the bus. This bit-width weight of a bus differs from the bits weight of a channel in that the former represents physical wires, whereas the latter represents data to be transferred. For example, a 16-bit wide bus may be used by a channel that transfers 32 bits of data (thus requiring two 16 bit transfers), and may even be used by other channels that transfer varying numbers of bits.

We can also annotate each system component with a constraint on the maximum I/O that the component can implement, i.e., with the number of available I/O pins. For an ASIC, this number might be the the number of pins excluding power and ground, whereas for a standard processor, it might be the size of the bus.

2.4.3 Size

We now turn our attention to size measurement. We need to determine the size required by a set of behaviors that have been partitioned onto a particular component. For behaviors and variables partitioned onto a standard processor, size may mean the number of program and data bytes that the behaviors and variables will be compiled into. Alternatively, on a custom processor, size may mean the number of gates, cells, transistors, or combinational

logic blocks, depending on whether the component technology is that of gate arrays, standard cells, custom layout, or FPGAs. For variables partitioned onto a memory, size may mean the number of words. Because size means something different for each type of component technology, we annotate each behavior and variable node with a list of **size** weights, one weight for each type of system component on which that node could possibly be implemented. As with execution times, we can either synthesize or compile each behavior into the appropriate technology to obtain the appropriate weights.

We shall see in the next section that size weights can be summed to obtain a size estimate for a set of behaviors on a component, but such summing may prove inaccurate in some cases. For example, such summing is fairly accurate in the case of a standard processor, since a behavior set's number of bytes can be approximated as the sum of each behavior's number of bytes. On the other hand, such summing may be inaccurate for datapath-intensive behaviors on a custom processor, since such behaviors will likely share much hardware among them, causing a simple summation of each behavior's size to result in an overestimate. There is a solution to this shared hardware problem, which we describe in [1]; we omit the details here as they are beyond the scope of this paper. For the purposes of this paper, we shall assume that hardware is estimated as a sum of weights, which is reasonably accurate for non-signal-processing examples.

As was the case with I/O, we can annotate each system component with a size constraint. An ASIC might have a constraint on the maximum number of gates, whereas a standard processor might have a constraint on the program size.

2.5 SLIF definition with annotations

To incorporate the above annotations, we now extend the definition of Section 2.2 as follows:

bv_i	$=< process, ict_list, size_list >$,
$process$	$\in \{true, false\}$ – true denotes a process behavior,
ict_list	$= \{ict_1, ict_2, ...\}$,
ict_k	$=< comp, val >$ – the ict val ($val \in Natural$) on the component $comp$, ($comp \in P \bigcup M$),
$size_list$	$= \{size_1, size_2, ...\}$,
$size_k$	$=< comp, val >$ – the size val ($val \in Natural$) on the component $comp$, ($comp \in P \bigcup M$),
c_i	$< src, dst, accfreq, bits >$,
i_k	$< C, bitwidth, ts, td >$,
m_k	$=< V, sizecon >$, where con ($con \in Natural$) is the size constraint,
p_k	$=< BV, sizecon >$, where con ($con \in Natural$) is the size constraint

An example of channel annotations is shown in Figure 3, where we have annotated the edge from *EvaluateRule* to *in1val* with a bits value of 8 for the integer and an access frequency of 1. On the other hand, the edge to the array *mr1* has a bits value of 15 (7 address bits plus 8 data bits) and an access frequency of 65. An example of an *ict_list* is also shown for *Convolve*, whose internal computation time is 80 us on the given processor type, and only 10 us on the given ASIC type.

3 Estimation of quality metrics

Given the above SLIF definition, we now turn our attention to using SLIF for system design. Recall that the key tasks of system design include allocation, partitioning, transformation, and estimation. SLIF directly supports the tasks of allocation and partitioning, because it represents system components, and it represents the mapping of functional objects to those components. We plan to demonstrate SLIF's support of transformations in a future publication; briefly, a transformation, such as procedure inlining or process merging, would require modification of certain nodes and edges, along with recomputation of certain annotations. In this section, we focus on the estimation task, demonstrating how SLIF supports rapid estimation of design metrics for a given partition of functional objects among system components, by defining equations for those metrics based on SLIF annotations. We consider the metrics of execution time, bitrate, software size, hardware size, memory size, and I/O. We must point out that each equation represents just one way to compute a particular metric. There is likely to be more than one method to compute a given metric from the SLIF information, with some methods being more accurate than others but requiring more computation time or a more complex implementation.

3.1 Execution time

An execution time estimate for a behavior can be computed from the internal computation time (ict) information and from the channel annotations. The simplest method requires assuming that a behavior's channel accesses occur sequentially. This assumption holds for many examples.

We use a procedure $GetBvComp(bv)$ that returns the processor or memory component pm to which bv has been mapped. Specifically, it returns the p, $p \in P_{all}$, such that $bv \in p.BV$, or the m, $m \in M_{all}$, such that $bv \in m.V$. A procedure $GetBvIct(bv, pm)$ finds the ict_k in the $bv.ict_list$ for which $ict_k.comp$ equals the given processor or memory pm, and then returns $ict_k.val$. A procedure $GetBehChans(b)$ returns all channels accessed by the behavior b. Specifically, it returns the set C, $C \subset C_{all}$, for which each channel $c \in C$ has $c.src = b$. A procedure $GetChanBus(c)$ returns the bus to which the channel c has been mapped. Specifically, it returns the bus i, $i \in I_{all}$, such that $c \in I.C$. Execution time can then be computed as follows:

$$Exectime(b) = GetBvIct(b,p) + Commtime(b) \tag{1}$$

$$\begin{aligned}
p &= GetBvComp(b),\\
Commtime(b) &= \textstyle\sum_{c_k \in GetBehChans(b)} c_k.accfreq \times (Transfertime(c_k,p)\\
&\quad + Exectime(c_k.dst)),\\
Transfertime(c_k,p) &= \lceil bdt_time \times (c_k.bits \div GetChanBus(c_k).bitwidth) \rceil,\\
bdt_time &= GetChanBus(c_k).ts \text{ if } GetBvComp(c_k.dst) = p,\\
&= GetChanBus(c_k).td \text{ otherwise},
\end{aligned}$$

In other words, a behavior's execution time equals the behavior's internal computation time on the particular system component ($GetBvIct(b,p)$), plus the behavior's communication time ($Commtime(b)$). The communication time, in turn, equals the time spent transfering data across a channel for each accessed object ($Transfertime(c_k,p)$), plus the execution time of each accessed object ($Exectime(c_k.dst)$), multiplied by the number of times each access occurs ($c_k.accfreq$). The time spent transfering data over a channel, in turn, is determined from the data transfer time of the bus to which the channel has been mapped, and from the bit-width (wires) of that bus; if the number of bits to be transferred exceeds the number of wires, then multiple transfers are assumed (as computed by the division). The bus data transfer time (bdt_time) depends on whether the communicating objects are on the same component.

3.2 Bitrate

We now turn our attention to computing the rate at which bits are transferred over a particular channel or over a particular bus. Such a rate is called a bitrate. The bitrate of a channel can be computed as the number of bits transferred during a start-to-finish execution of the source behavior. We make use of the $Exectime(b)$ procedure defined above. Bitrate of a channel can then be computed as follows:

$$ChanBitrate(c) = \frac{c.accfreq \times c.bits}{Exectime(c.src)} \tag{2}$$

In other words, the channel's bitrate is the number of bits transferred per channel access multiplied by the number of such accesses during a start-to-finish execution of the channel's source behavior, divided by the total execution time of that behavior. Note that the execution time is computed as described earlier, where we considered both the internal computation time as well as the communication time.

The bitrate of a bus can be computed as the sum of that bus' channel bitrates, as follows:

$$BusBitrate(i) = \sum_{c \in i.C} ChanBitrate(c) \tag{3}$$

More sophisticated bitrate estimation equations can be formulated to take into account the maximum bitrate capacity of a bus. In such techniques, if the bitrate capacity is exceeded, then we need to slow down the transfers. For further details on more sophisticated bitrate estimation techniques that derive their information from SLIF, see [2].

3.3 Software, hardware and memory size

The tasks of estimating the software size for a standard processor component, the hardware size for a custom processor component, and the memory size for a standard memory component are essentially the same when we use the size weights defined above. Each task requires adding the appropriate weight of each functional object for the particular component type. Those weights have already been computed and associated with each object for each possible component type, so only lookups are necessary.

We use a procedure $GetBvSize(bv, pm)$, which finds the $size_k$ in the $bv.size_list$ for which $size_k.comp$ equals the given processor or memory pm, and then returns $size_k.val$. The size of a processor component p or a memory component m is then computed as follows:

$$Size(p) = \sum_{bv_i \in p.BV} GetBvSize(bv_i, p) \tag{4}$$

$$Size(m) = \sum_{v_i \in m.V} GetBvSize(v_i, m) \tag{5}$$

Therefore, if p is a standard processor, then the sizes being summed correspond to the number of bytes for each behavior or variable when compiled into that processor's instruction set. If p is a custom processor component, then the sizes being summed correspond to the number of gates or transistors for each behavior or variable when synthesized using that component's technology library. For the case of a memory m, the sizes being summed correspond to the number of words for each variable when mapped to that memory. (As mentioned earlier, we can also use a more sophisticated technique for estimating hardware size).

3.4 I/O

I/O is the number of wires crossing the boundary of a system component. This number is usually relevant for ASICs.

$$I_O(p) = \sum_{i_k \in CutBuses(p)} i_k.bitwidth \tag{6}$$

$CutBuses(p)$	$\subset I, i_k \in CutBuses(p)$ iff
	$i_k \bigcap CutChans(p) \neq \emptyset$,
$CutChans(p)$	$\subset C, c_l \in CutChans(p)$ iff
	$c_l.src \in p$ and not $c_l.dst \in p$ or
	$c_l.dst \in p$ and not $c_l.src \in p$

In other words, the number of wires crossing a component boundary is equal to the total bitwidth of the buses that cross that boundary. The buses that cross that boundary ($CutBuses(p)$) are those that implement at least one channel that crosses the boundary. The channels that cross that boundary ($CutChans(p)$) are those that connect a behavior or variable of component p with another behavior, variable or port not in component p.

4 Related work

There are several research efforts that focus on performing system design tasks. Several efforts have focused on partitioning functionality among a hardware/software architecture [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13] for partitioning functionality among hardware modules [14, 15, 16, 17, 18, 19, 20, 21, 22, 23], for partitioning functionality among multiple processors [24], and for transformation during system design [25, 26]. Many of these efforts use an intermediate format intended to expose control and data dependencies between fine-grained operations, such as the Value Trace [16, 25], the Sequencing Intermediate Format [14], and variations of a control-dataflow graph [17, 19]; those formats were originally developed for high-level synthesis (see also [27, 28, 29]). Those formats may be too fine-grained for system-design tasks. To our knowledge, SLIF is one of the first published internal formats intended for the system-level design tasks. An exception is the hierarchical-transition graph format used in [26] to merge two processes into one; we are considering associating such graphs with each SLIF node to support process merging transformations.

5 Results

In this section, we provide CPU times for building a SLIF representation and for performing estimations on several examples, and we compare the size and estimation efficiency of SLIF with other formats,

The table in Figure 4 summarizes the time required to build a SLIF representation for several examples. The examples include a telephone answering machine (*ans*), an ethernet coprocessor (*ether*), a fuzzy-logic controller (*fuzzy*), and a volume-measuring medical instrument (*vol*). For each example, we indicate the number of lines of the VHDL behavior specification, the number of behavior and variable functional objects, the number of channels, the time required to build the SLIF representation (*T-slif*), and the time required to obtain size, pin, bitrate and performance estimates (*T-est*) for a partition of functional ob-

jects among a processor-asic architecture. All times are in seconds on a Sparc 2. The results demonstrate that the SLIF, with all its annotations, can be built in just a few seconds for even large examples. This time is very reasonable when we consider that the SLIF is built only once, when a system-design tool is first started. The results also demonstrate that size and performance estimates can be computed in less than a hundredth of a second. Such speed enables rapid feedback during interactive design, and permits the use of algorithms that explore thousands of possible designs.

	Lines	BV	C	T-slif	T-est
Ans	632	45	64	2.20	0.00
Ether	1021	123	112	10.40	0.00
Fuzzy	350	35	56	0.46	0.00
Vol	214	30	41	0.34	0.00

Figure 4: Results of building SLIF and obtaining estimations

To demonstrate the efficiency of SLIF over other formats, we compared the size of two other formats with that of SLIF for the fuzzy-logic controller example. The SLIF-AG for the example required 35 nodes and 56 edges. The ADD format [30], which is similar in form and complexity to the VT format, required over 450 nodes and 400 edges. The CDFG format required over 1100 nodes and 900 edges. The difference in complexity greatly affects the types of partitioning algorithms that can be applied. For example, if an n^2 algorithm is to be applied, then the SLIF-AG, VT or ADD, and CDFG formats would require 1225, 202500, and 1210000 computations, respectively. Clearly, latter two are not practical for an interactive tool.

SLIF's higher abstraction level also permits extensive preprocessing. For example, suppose we are given a set of nodes to be implemented on an ASIC, and we wish to estimate the ASIC's size. Such size estimation can be done by synthesizing the set of nodes into a register-transfer structure. Using SLIF, we can synthesize each node beforehand, so size estimation only requires adding the previously-determined node sizes, which in turn requires only a fraction of a second. For the other two formats, it does not make sense to synthesize each node beforehand, since each node is of the granularity of an arithmetic operation. If we somehow assigned a size to each node and summed these sizes, we would grossly overestimate total size, since the sharing of functional units would be neglected entirely. To obtain a size estimate for a set of nodes, we instead have to perform a rough synthesis on that entire set of nodes. This synthesis requires several seconds. While this time is feasible for interactive design, it is not feasible when we use algorithms that examine thousands of possibilities.

6 Conclusions and future work

We have presented the SLIF specification-level internal format and shown its suitability for the system design problems of allocation, partitioning, and estimation (we plan to demonstrate its suitability for transformation in future work). SLIF has proven to be an efficient internal format in the SpecSyn system design tool. SpecSyn permits rapid exploration of partitions of functionality among processors, ASICs, memories and bus components, providing rapid estimates of size, I/O, and performance metrics for each option examined. SpecSyn has been released to over 20 companies and universities and has been used experimentally in several industry designs. While feedback from designers regarding the accuracy of the estimations has been positive, we hope to measure the accuracy more quantitatively in the future. We also plan to continue to extend SLIF to represent more sophisticated architectures, such as those including hierarchical components, pipelined processors, and memory hierarchies.

References

[1] F. Vahid, S. Narayan, and D. Gajski, "Constant-time cost evaluation for behavioral partitioning." UC Irvine, Dept. of ICS, Technical Report 92-29,1992.

[2] S. Narayan and D. Gajski, "Synthesis of system-level bus interfaces," in *Proceedings of the European Conference on Design Automation (EDAC)*, 1994.

[3] 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.

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

[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] A. Kalavade and E. Lee, "A hardware/software codesign methodology for DSP applications," in *IEEE Design & Test of Computers*, 1993.

[7] K. ten Hagen and H. Meyr, "Partitioning and surmounting the software-hardware abstraction gap in an asic design project," in *Proceedings of the International Conference on Computer Design*, pp. 462–465, 1993.

[8] W. Ecker, "Using VHDL for HW/SW co-specifications," in *Proceedings of the European Design Automation Conference (EuroDAC)*, pp. 630–635, 1992.

[9] M. Edwards and J. Forrest, "A development environment for the cosynthesis of embedded software/hardware systems," in *Proceedings of the European Conference on Design Automation (EDAC)*, pp. 469–473, 1994.

[10] 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.

[11] D. Gajski, F. Vahid, and S. Narayan, "A system-design methodology: Executable-specification refinement," in *Proceedings of the European Conference on Design Automation (EDAC)*, 1994.

[12] U. Steinhausen, R. Camposano, H. Gunther, P. Ploger, M. Theibinger, H. Veit, H. Vierhaus, U. Westerholz, and J. Wilberg, "System synthesis using hardware-software co-design," in *International Workshop on Hardware-Software Co-Design*, pp. 1–11, 1993.

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

[14] 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.

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

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

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

[18] P. Gupta, C. Chen, J. DeSouza-Batista, and A. Parker, "Experience with image compression chip design using unified system construction tools," in *Proceedings of the Design Automation Conference*, pp. 250–256, 1994.

[19] C. Gebotys, "Optimal scheduling and allocation of embedded VLSI chips," in *Proceedings of the Design Automation Conference*, pp. 116–119, 1992.

[20] N. Kumar, R. Vemuri, and R. Vemuri, "Partitioning for multicomponent synthesis from VHDL specifications," in *VHDL International Users' Forum*, pp. 19–28, 1993.

[21] Y. Chen, Y. Hsu, and C. King, "MULTIPAR: Behavioral partitioning for synthesizing application-specific multiprocessor architectures," in *Proceedings of the European Conference on Design Automation (EDAC)*, pp. 14–18, 1992.

[22] 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.

[23] 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.

[24] S.Prakash and A. Parker, "Synthesis of application-specific multiprocessor architectures," in *Proceedings of the Design Automation Conference*, pp. 8–13, 1991.

[25] R. Walker and D. Thomas, "Behavioral transformation for algorithmic level IC design," *IEEE Transactions on Computer-Aided Design*, October 1989.

[26] J. Hagerman and D. Thomas, "Process transformation for system level synthesis." Technical Report CMUCAD-93-08, 1993.

[27] J. Van-Eijndhoven and L. Stok, "A data flow graph exchange standard," in *Proceedings of the European Conference on Design Automation (EDAC)*, pp. 193–199, 1992.

[28] P. Eles, K. Kuchcinski, Z. Peng, and M. Minea, "Compiling vhdl into a high-level synthesis design representation," in *Proceedings of the European Design Automation Conference (Euro-DAC)*, pp. 604–609, 1992.

[29] D. Knapp and A. Parker, "A unified representation for design information," in *Proceedings of the International Symposium on Computer Hardware Description Languages and their Applications*, 1985.

[30] V. Chaiyakul and D. Gajski, "High-level transformations for minimizing syntactic variances," in *Proceedings of the Design Automation Conference*, 1993.