

SpecSyn: An Environment Supporting the Specify-Explore-Refine Paradigm for Hardware/Software System Design

Daniel D. Gajski¹, Frank Vahid², Sanjiv Narayan³, and Jie Gong⁴

1. Dept. of ICS, Univ. of California, Irvine
2. Dept. of CS, Univ. of California, Riverside
3. Ambit Design Systems, Fremont, California
4. Motorola, Tempe, Arizona.

Technical Report CS-96-08

September 1, 1996

To appear in IEEE Transactions on VLSI System

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

System-level design issues are gaining increasing attention, as behavioral synthesis tools and methodologies mature. We present the SpecSyn system-level design environment, which supports the new specify-explore-refine (SER) design paradigm. This three-step approach to design includes precise specification of system functionality, rapid exploration of numerous system-level design options, and refinement of the specification into one reflecting the chosen option. A system-level design option consists of an allocation of system components, such as standard and custom processors, memories, and buses, and a partitioning of functionality among those components. After refinement, the functionality assigned to each component can then be synthesized to hardware or compiled to software. We describe the issues and approaches for each part of the SpecSyn environment. The new paradigm and environment are expected to lead to a more than 10 times reduction in design time, and our experiments support this expectation.

Contents

1	Introduction	1
2	Specification capture	4
2.1	Models and languages	4
2.2	Internal representation	5
3	Exploration	7
3.1	Allocation	8
3.2	Partitioning	9
3.2.1	Hardware and hardware/software partitioning	9
3.2.2	Heuristics	10
3.2.3	Manual partitioning and hints	10
3.2.4	Cost functions	11
3.3	Estimation	12
3.3.1	Pre-estimation and online-estimation	13
3.3.2	Performance	14
3.3.3	Hardware size	16
3.3.4	Software size	18
3.4	Transformations	19
4	Refinement	20
4.1	Interfacing	20
4.2	Memories	21
4.3	Arbitration	21
4.4	Generation	21
4.5	Validation	21
5	Experiments	23
6	Related work, current status, and future work	24
7	Conclusions	26

List of Figures

1	The specify-explore-refine approach to system design	2
2	The SpecSyn system-design environment	3
3	Partial VHDL specification of a fuzzy-logic controller example.	6
4	Basic SLIF-AG for the example.	6
5	Slif behavior/variable annotations for the example.	7
6	Slif channel annotations for the example.	8
7	An example allocation of components	9
8	Partitioning AG nodes among system components	12
9	Subtasks during exploration	13
10	Execution-time equations for the example	15
11	Evaluating execution times for the example.	16
12	CU/DP area model	17
13	Equation and terms for computing CU/DP area	17
14	Software size estimation: (a) processor-specific model, (b) generic model.	19
15	Refined fuzzy-controller VHDL partial specification.	23
16	Exploration for the fuzzy logic controller	29
17	Fuzzy logic controller industry design summary	30

1 Introduction

The focus of design effort on higher levels of abstraction, driven by increasing system complexity and shorter design times, has led to the need for a system-level design methodology and supporting tools. To better understand the system design problem, we can isolate three distinct tasks. First, we must specify the system’s functionality and constraints. Second, we must explore various system-level designs, each consisting of an interconnection of system components and an assignment of functionality to them. System components include standard processors, custom (ASIC) processors, memories, and buses. Third, we must define the system-level design for designers who will in turn create an implementation for each component.

In current practice, these three steps are carried out in an informal and ad-hoc manner. Specifications are usually written informally in English or some other natural language. Exploration is done manually using mental or hand-calculated estimations of quality metrics such as performance, size, and power. The system-level design is then described informally using block diagrams and English. Drawbacks of such informal techniques include the lack of early simulation, the lack of rapid feedback of quality metrics that result from design decisions, the lack of automated tools to explore more design alternatives while requiring less design time, and the lack of good documentation of each component’s functionality as well as of the design decisions to aid in concurrent design, component integration and redesign.

The response in the research community to the above drawbacks has been to introduce simulatable specifications earlier into the design process, and to use automated tools to assist in the exploration of design alternatives. The specify-explore-refine paradigm, which can also be thought of as a hierarchical modeling methodology, may further improve the situation. In such an approach, we first precisely specify the system’s functionality, explore numerous system-level implementations with the aid of tools, and then automatically generate a refined description representing any implementation decisions.

More specifically, the following tasks, illustrated in Figure 1, are necessary to create a system-level design:

- *Specification capture:* To specify the desired system functionality, we decompose the functionality into pieces by creating a conceptual model of the system. We generate a description of this model in a language. We validate this description by simulation or verification techniques. The result of specification capture is a *functional specification*, which lacks any implementation detail.
- *Exploration:* We explore numerous design alternatives to find one that best satisfies our constraints. To do this, we transform the initial description into one more suitable for implementation. We allocate a set of system components and specify their physical and performance constraints. We partition the functional specification among allocated components. For guidance in these exploration subproblems, we estimate each alternative design’s quality.
- *Specification refinement:* We refine the initial specification into a new description that reflects the decisions that we have made during exploration. To do this, we move each

variable into a memory, insert interface protocols between components, and add arbiters to linearize concurrent accesses to a single resource. Then we generate a system description detailing the system's processors, memories, and buses. We use cosimulation to verify that this refined description is equivalent to the initial specification. The result of specification refinement is a *system-level description*, which possesses some implementation details of the system-level architecture we have developed, but otherwise is still largely functional.

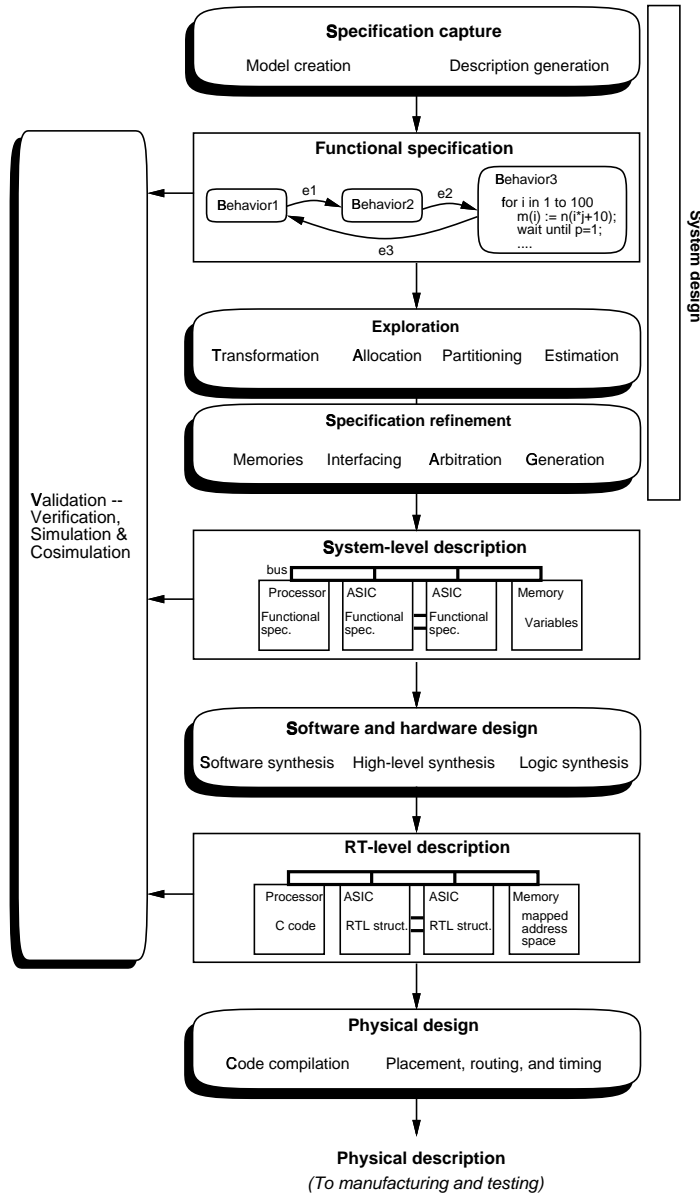


Figure 1: The specify-explore-refine approach to system design

Afterwards, we perform *software and hardware design*, where we create an implementation for each component, using software and hardware design techniques. A standard processor component requires software synthesis, which determines software execution order to satisfy resource and performance constraints. We can obtain a custom processor’s design through high-level (behavioral) synthesis [1, 2], which converts the behavioral description into a structure of components from a register-transfer (RT) library containing microarchitectural components, such as ALUs, registers, counters, register files and memories. The control logic and some RT components are synthesized with finite-state machine and logic synthesis techniques [3]. The result of software and hardware design is an *RT-level description*, which contains optimized C code for software and RT-level netlists for custom components. The RT-level description is then passed to *physical design* for final implementation.

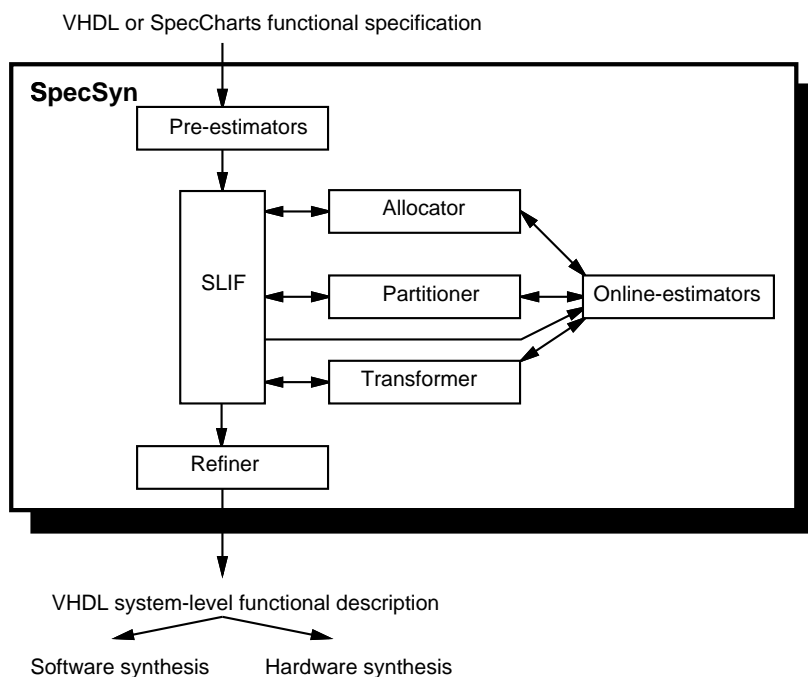


Figure 2: The SpecSyn system-design environment

We have developed the SpecSyn environment to support the specify, explore, and refine steps – the SER paradigm. The various parts of SpecSyn, illustrated in Figure 2, correspond to the various system-design subtasks described above. Discussion of how SpecSyn differs from many related efforts is found in Section 6; however, we point out two key differences here. First, SpecSyn outputs a system-level description, which differs from the input only by the addition of system-level architectural features. This output can thus be treated as though it were hand-written. Specifically, it can be easily read and understood, used as documentation, input to simulators, input to behavioral synthesis, input to real-time schedulers (and ideally compilers), or designed from manually. SpecSyn thus fits in well with current practice. Second, SpecSyn was developed as a general tool intended to support a wide variety of implementation component technologies, architectures, and heuristics, and

new versions of such items can be added.

In this paper, we present an overview of the SpecSyn environment, discussing relevant issues, previous work, and solutions for each part. We then present industry experiments using SpecSyn and the specify-explore-refine paradigm.

2 Specification capture

2.1 Models and languages

Specifying a system’s functionality is a difficult task, because the functionality is often complex and poorly understood. To ease the specification task, one decomposes the functionality into pieces according to some model, and captures that model in some language. This distinction between a model and language is important, since the choice of a model affects the ease of the specification task much more than does the choice of language. Common models include communicating sequential processes (CSP) [4], dataflow graphs, hierarchical finite-state machines (FSM), and Petri-nets. Common languages include C, VHDL, Verilog, and Statecharts [5]. Each language can capture many models, but certain languages excel for particular models. For example, Statecharts excels at capturing FSMs, even though VHDL and Verilog can also capture FSMs, albeit with more effort.

We observed that no existing model or language catered to the capture of embedded systems. Embedded systems are those systems whose functionality is determined mostly by interactions with the environment. Examples include most controller and telecommunication systems. We found that many such systems possess several characteristics, including *state-transitions*, *exceptions*, *forking*, and *program-like computations*, which are not all supported by any one existing model. State-transitions, exceptions and forking are supported by the hierarchical FSM model, while forking and program-like computations are supported by the CSP model. To overcome this lack of support, we developed the Program-State Machine model (PSM), which is essentially a combination of hierarchical FSM’s (Statecharts) and CSP. The model consists of a hierarchy of program-states. Each program-state can be decomposed into concurrent program-substates or sequential program-substates sequenced by arcs, as in Statecharts. However, unlike Statecharts, a third option is to decompose a program-state into sequential program statements. Because a program-state is not just a state but also a computation, two types of arcs are required: transition-on-completion, which is traversed when the computation has completed, and transition-immediately, which is traversed when the arc event occurs, regardless of the computation stage. We also developed the SpecCharts language, which is an extension to VHDL, to capture the PSM model. SpecCharts can be translated automatically to VHDL, which will be more complex than the original SpecCharts, but is simulatable and (ideally) synthesizable in a VHDL environment.

The choice of a language depends on more than just supported system characteristics, so SpecSyn also accepts the industry standard of VHDL as input. Though languages such as VHDL and Verilog lack support for certain embedded system characteristics, most notably for state-transitions, one can always use some more complex combination of other constructs, which of course is more time-consuming and error-prone, but not impossible. For example, we can always capture state transitions using sequential program constructs. Such capture using less appropriate constructs is analogous to capturing a record using

multiple scalar variables, capturing recursion using a stack, or capturing a parser using C’s sequential constructs; all such captures are possible (and in fact tools usually translate to such constructs during processing), but are tedious for humans to perform directly.

There are many other system characteristics that are not directly supported by languages such as SpecCharts, VHDL, Verilog, and C, including synchronous dataflow [6], queuing, complex timing constraints, and mixed analog/digital parts. No one language directly supports all characteristics, but hybrid models and languages that extend the number of supported characteristics, such as PSM and SpecCharts, seem to be a step in the write direction. For more information on PSM and SpecCharts, we refer the reader to [7, 8].

In addition to specifying functionality, the designer must also specify design constraints. SpecSyn permits minimum and maximum constraints to be specified on behavior execution times and channel bitrates. Ideally, one would also be able to specify overall design constraints, such as power, board size, dollar cost, and design cost (if these items could be quantified). More specific design constraints, such as a component’s size and I/O limitations, will be derived from the component’s library entry later.

2.2 Internal representation

The captured specification must be converted into an internal representation on which subsequent tools can operate. Representations commonly used for behavioral synthesis, including the control/dataflow graph (CDFG) and Value Trace [1], expose control and data dependencies between arithmetic-level operations, which may be too fine-grained for system design tasks. Most good partitioning heuristics would require long runtimes on the resulting large numbers of objects, and estimators could not obtain meaningful pre-estimates (see Section 3.3) for each object. Refinement into a readable system-level description also becomes a nearly impossible task. Thus, we chose to create a representation based on the coarser-granularity of procedural-level computations.

A second drawback of using behavioral synthesis representations stems from their focus on dependencies. Such dependencies are necessary for scheduling during behavioral synthesis, but are not essential to performing system design tasks. Representing dependencies between procedural-level objects requires us to replicate each object at each place that a procedure is called, since dependencies will differ for each call. This replication makes the system design task much more complex. Instead, we developed a representation that focuses on representing the accesses, rather than dependencies, among objects.

For example, consider the partial VHDL specification of a fuzzy-logic controller in Figure 3. Inputs *in1* and *in2* must be converted to output *out1* using fuzzy logic. The main process *FuzzyMain* first samples input values by writing them into variables *in1val* and *in2val*. It then calls procedure *EvalRule* twice, once for each input, and that procedure fills an array (*tmr1* or *tmr2*) based on the input and on another predefined array (*mr1* or *mr2*). After convolving the *tmr* arrays, a centroid value is computed and output. The process repeats after a time interval.

We represent this specification as the directed graph in Figure 4. Each graph node represents a **behavior** or a **variable** from the specification, where a behavior is a process or procedure, though for finer granularity we can consider statement blocks like loops (see Section 3.4). Each graph directed-edge represents a communication **channel** from the

```

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
  function Min ...
  ...
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;
end;

```

Figure 3: Partial VHDL specification of a fuzzy-logic controller example.

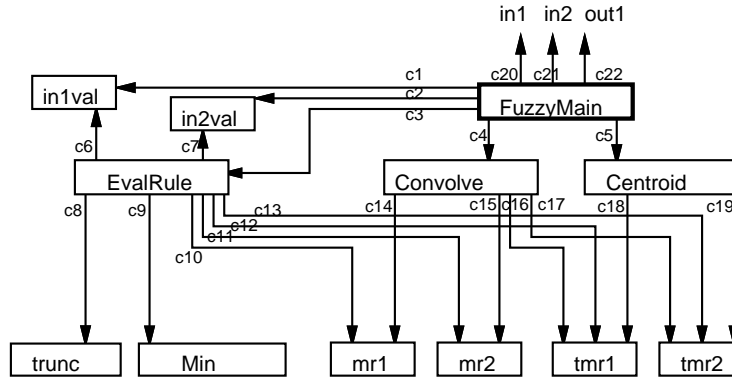


Figure 4: Basic SLIF-AG for the example.

specification, where a channel represents a procedure call, a variable/port read or write, or a message pass specified using send/receive constructs. For example, process *FuzzyMain*, procedure *EvalRule* and variable *in1val* are each represented by a node. The write of *in1val* in *FuzzyMain* translates to a single edge, while the two calls of *EvalRule* by *FuzzyMain* translate to another single edge. Nodes representing processes are tagged to distinguish them from procedure nodes (hence the *FuzzyMain* node is shown in bold).

We refer to the representation as the **Specification-level intermediate format**, or **SLIF**, since its granularity is that of behaviors and variables explicit in the specification. We refer to the part of SLIF shown so far as an **access graph**, or **AG**, since the relations between the behaviors/variables are defined by the accesses among those objects. The AG is similar to a procedure call-graph commonly used for software profiling, where an edge represents an access rather than a flow of data; the AG is more general since it also includes variables. Note that the AG uses only one node for *EvalRule* and one for *Min*, even though each behavior is called more than once with different dependencies for each call; thus, a

Object	ict_8051	ict_XC4020	ict_V100	size_8051	size_XC4020	size_V100
FuzzyMain	5	8		80	500	
in1val	0	0	0	2	80	2
in2val	0	0	0	2	80	2
EvalRule	778	522		500	1600	
Convolve	800	600		900	2000	
Centroid	2,500	2000		700	4500	
trunc	0	0	0	2	80	2
Min	8	3		30	850	
mr1	0	0	0	768	30720	768
mr2	0	0	0	768	30720	768
tmr1	0	0	0	256	10240	256
tmr2	0	0	0	256	10240	256

Figure 5: Slif behavior/variable annotations for the example.

large increase in the number of nodes is prevented using the AG. Sometimes we do want multiple nodes, however, which can be handled using a procedure cloning transformation to be discussed in Section 3.4.

SLIF is annotated with numerous values. We annotate each behavior and variable object with a list of size weights, one weight for each type of component to which the object may be assigned. For example, a variable object is annotated with the number of memory words required for storage in each library memory. A behavior is annotated with numbers of square microns, gates, and combintational-logic blocks for each custom chip, ASIC, and FPGA, respectively, on which the behavior could be implemented. (More complex annotations can be used to consider hardware sharing; see Section 3.3.3). In addition, a behavior is annotated with the number of bytes for each possible standard processor.

We annotate each behavior and variable object with computation time weights for each possible component, corresponding to a variable’s access time, or to a behavior’s execution time (excluding communication time). Times can be obtained through profiling. We also annotate each edge with access frequency weights, which can also be obtained through profiling. Furthermore, we associate a bits weight with each edge, indicating the number of bits sent during each transfer.

For all annotations, we might associate average, minimum and maximum values.

Annotations are computed during pre-estimation, and are combined into quality metric estimates during online estimation; Section 3.3 discussions these two estimation steps further.

3 Exploration

Given a functional specification, we must proceed to create a system-level design of interconnected components, each component implementing a portion of that specification. A design’s acceptability is evaluated by how well it satisfies constraints on design metrics, such as performance, size, power and cost. Since substantial time and effort are needed to evaluate a potential design, designers usually examine only a few potential designs, often

Object	accfreq	bits	src	dst
chan1	1	6	FuzzyMain	in1val
chan2	1	6	FuzzyMain	in2val
chan3	2	8	FuzzyMain	EvalRule
chan4	1	0	FuzzyMain	Convolve
chan5	1	0	FuzzyMain	Centroid
chan6	1	16	EvalRule	in1val
chan7	1	16	EvalRule	in2val
chan8	129	16	EvalRule	trunc
chan9	129	32	EvalRule	Min
chan10	65	32	EvalRule	mr1
chan11	65	32	EvalRule	mr2
chan12	32	16	EvalRule	tmr1
chan13	32	16	EvalRule	tmr2

Figure 6: Slif channel annotations for the example.

those that they can evaluate quickly because of previous experience.

By using a machine-readable specification, we can automatically explore large numbers of potential designs rapidly. Exploration of potential designs can be decomposed into four interdependent subproblems: allocation, partitioning, transformation and estimation. We need not solve these problems in the given order; in fact, we will usually need to iterate many times before we are satisfied with our system-level design.

3.1 Allocation

Allocation is the task of adding components to the design. Many possible components exist. A standard processor is programmable and comes with widely-used compilers and debuggers, but is usually slow or large. A special-purpose processor, such as a DMA controller or Fourier transform, performs a specific function. A custom processor is synthesized to quickly execute a set of functions, but is harder to design and modify. A memory stores variables. A bus implements communication between processors/memories.

The SpecSyn allocator permits allocation of any number of standard processors, custom processors, memories, and buses. Of course, allowing any allocation is only useful if the exploration tool understands the allocation; specifically, if the tool knows how to partition functionality among the components, knows how to estimate for such a partition, and can generate a refined description with behavior for each component. Incorporating such knowledge, especially that required for estimation, is very difficult, which is the reason that current tools only support a subset of possible allocations, such as a particular interconnection of a standard processor, memory, bus and custom processor [9, 10]. While SpecSyn permits a variety of allocations, its estimation models and heuristics must continually be improved to better apply to each.

Each component is characterized in a library by its constraints, and by a technology file. For example, a custom processor might be characterized by the maximum I/O pins and gates, and by a technology file describing an RT-component library. A standard processor is characterized by a maximum program memory size, a bus size, a maximum bus bitrate,

and a technology file describing how to map a generic instruction set to the processor's instruction set (see Section 3.3). A memory is characterized by the number of ports, number of words, word width, and access time. A bus is characterized by the number of wires, protocol, and maximum bitrates.

Ideally, we would also be able to allocate special-purpose processor components (e.g., DMA controllers), as well as hierarchical components, such as an ASIC which itself contains a standard processor core, memory, and several custom processor blocks.

Figure 7 demonstrates an example allocation. *StandardProc1* is an Intel 8051 with 4 kilobytes of on-chip memory, and *CustomProc1* is a Xilinx XC4010 FPGA with 160 I/O pins and 10,000 gates. Two 1 kbyte memories are also allocated.

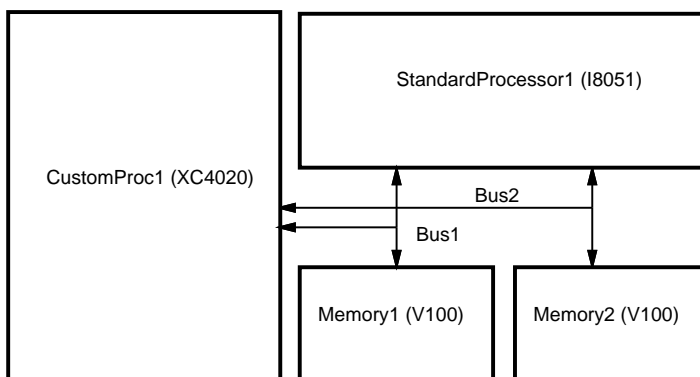


Figure 7: An example allocation of components

3.2 Partitioning

Given a functional specification and an allocation of system components, we need to partition the specification and assign each part to one of the allocated components. In fact, we can distinguish three types of *functional objects* that must be partitioned. One type is a *variable*, which stores data values. Variables in the specification must be assigned to memory components. The second is a *behavior*, which transforms data values. A behavior may consist of programming statements, such as assignment, if and loop statements, and it generates a new set of values for a subset of variables. Behaviors must be assigned to custom or standard processors. The third is the *channel*, which transfers data from one behavior to another. Channels must be assigned to buses. Specification partitioning strives to satisfy constraints, specified by the user or associated with allocated components.

3.2.1 Hardware and hardware/software partitioning

A variety of techniques have evolved to assist the designer perform functional partitioning. We can consider two categories of techniques: hardware partitioning and hardware/software partitioning. The hardware partitioning techniques aim to partition functionality among hardware modules, such as among ASICs or among blocks on an ASIC. Most such techniques partition at the granularity of arithmetic operations, differing in the partitioning heuristics

employed. Clustering heuristics are used in [11, 12], integer-linear programming in [13, 14], manual partitioning in [15], and iterative-improvement heuristics in [16, 17]. Other techniques for hardware partitioning operate at a higher-level of granularity, such as in [18] where processes and subroutines are partitioned among ASICs using clustering, iterative-improvement, and manual techniques. Experiments have shown tremendous advantages of functional partitioning over the current practice of structural partitioning [19].

Hardware/software partitioning techniques form the second functional partitioning category. These techniques focus on partitioning functionality among a hardware/software architecture. The techniques in [20], [10, 9], and [21, 22, 23, 24] partition at the statement, statement sequence and subroutine/task levels, respectively.

In SpecSyn, both the hardware and hardware/software partitioning techniques are supported, since one can allocate any combination of hardware and software components and assign pieces of the specification to those components.

3.2.2 Heuristics

Instead of using one particular partitioning heuristic, SpecSyn uses a partitioning engine called GPP (General Purpose Partitioner). GPP is a library of functions with uniform interfaces, implementing the basic control strategies of numerous common heuristics, including clustering, group migration (an extension of Kernighan/Lin), simulated annealing, clique partitioning, genetic evolution, as well as custom heuristics. These control strategies are distinct from data structures and cost functions. A particular partitioning problem can be solved by calling a heuristic with the appropriate data structure and cost function – for example, circuit partitioning can be solved by passing a hypergraph data structure and a min-cut cost function. Each SpecSyn partitioning problem, including variables to memories, channels to buses, and behaviors to processors, is performed by passing the appropriate data structure and cost function and then applying the existing heuristics.

SpecSyn’s approach to partitioning thus addresses the fact that heuristics, data structures and cost functions are continually evolving. A new partitioning problem can initially be solved using a general heuristic. Then, once the problem definition has matured, one can develop and easily integrate a new custom heuristic. A user, after some experimentation, can choose the heuristic(s) with the appropriate result quality and runtime.

3.2.3 Manual partitioning and hints

We have also focused on supporting manual partitioning because of the importance placed on designer interaction. Such support not only involves providing the ability to manually relocate objects, but also allowing user control of the relative weights of various metrics in the cost function (see below), and automatically providing hints of what changes might yield improvements to the current partition. SpecSyn currently supports two types of hints. Closeness hints provide a list of object pairs, sorted by the closeness of the objects in each pair. Closeness is based on a weighted function of various closeness metrics. There are currently six behavior closeness metrics supported [25]:

- *Connectivity* is based on the number of wires shared between the sets of behaviors. Grouping behaviors that share wires should result in fewer pins.

- *Communication* is based on the number of bits of data transferred between the sets of behaviors, independent of the number of wires used to transfer the data. Grouping heavily communicating behaviors should result in better performance, due to decreased communication time.
- *Hardware sharing* is based on the estimated percentage of hardware that can be shared between the two sets of behaviors. Grouping behaviors that can share hardware should result in a smaller overall hardware size.
- *Common accessors* is based on the number of behaviors that access both sets of behaviors. Grouping such behaviors should result in fewer overall wires.
- *Sequential execution* is based on the ability to execute behaviors sequentially without loss in performance.
- *Constrained communication* is based on the amount of communication between the sets of behaviors that contributes to each performance constraint. Grouping such behaviors should help ensure that performance constraints are met.
- *Balanced size* is based on the size of the sets of behaviors. Grouping smaller behaviors should eventually lead to groups of balanced size.

There are also three closeness metrics supported for variables and for channels:

- *Common accessors* is based on the number of behaviors that access both sets of variables/channels. Grouping such variables/channels should result in fewer overall wires.
- *Sequential access* is based on the occurrence of sequential, rather than concurrent, access of the variables/channels by behaviors. Grouping sequentially-accessed variables/channels into the same memory does not decrease performance, whereas grouping concurrently-accessed ones might decrease performance due to access conflicts.
- *Width similarity* is based on the similarity of the variables'/channels' bitwidths. Grouping variables/channels with similar bitwidths should result in fewer wasted memory/bus bits.

The other type of hint is called lookahead. Here, we generate all possible n modifications of the current partition, where an n modification is a sequence of n moves of any objects from one group to another (n is user-defined). We again provide a list of such modifications, sorted by the partition improvement gained by each as measured by a cost function.

3.2.4 Cost functions

Partitioning heuristics are guided by cost functions. A variety of cost functions can be supported. The following supported cost function focuses on satisfying constraints:

$$\begin{aligned} \text{Costfct} &= k_1 \cdot F(\text{component1.size}, \text{component1.size_constr}) \\ &+ k_2 \cdot F(\text{component2.size}, \text{component2.size_constr}) \end{aligned}$$

$$\begin{aligned}
&+ k_3 \cdot F(\text{component1.IO}, \text{component1.IO_constr}) \\
&+ k_4 \cdot F(\text{behavior1.exectime}, \text{behavior1.exectime_constr}) \\
&\dots
\end{aligned} \tag{1}$$

where the k 's are user-provided constants indicating the relative importance of each metric, and F is a function indicating the desirability of a metric's value. A common form of F returns the degree of constraint violation, normalized such that 0 indicates no violation, and 1 indicates very large violation. This form of F causes the cost function to return zero when a partition meets all constraints, making the goal of partitioning to obtain a cost of zero.

The above cost function is very general, permitting us to satisfy constraints as well as to optimize certain metrics, without requiring specific knowledge in a heuristic of the constraints or optimization metrics. For example, if we wish to optimize execution time while satisfying size and I/O constraints, we can simply weigh size and I/O very heavily, so that violations of those constraints will not be tolerated. If we wish to focus first on just execution time, and then later on power, we can give the power constraint an initial weight of 0.

As an example of the results of partitioning, Figure 8 shows a partition of several of the previous example's nodes among two memories, an ASIC, a processor and a bus. Note that four communication channels have been partitioned onto *bus1*.

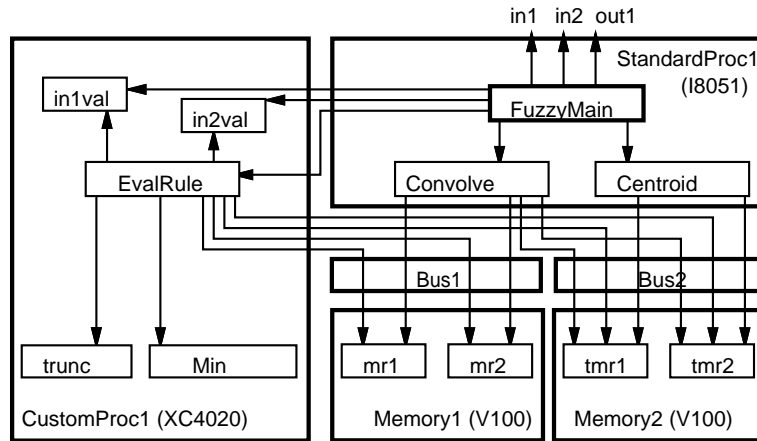


Figure 8: Partitioning AG nodes among system components

3.3 Estimation

Estimation of metric values is required to determine if a particular system-level design (a partition of functions among allocated components) satisfies constraints, and to compare alternative designs. In this section, we describe our two-level approach to fast and accurate estimation, and we provide details of our estimation models.

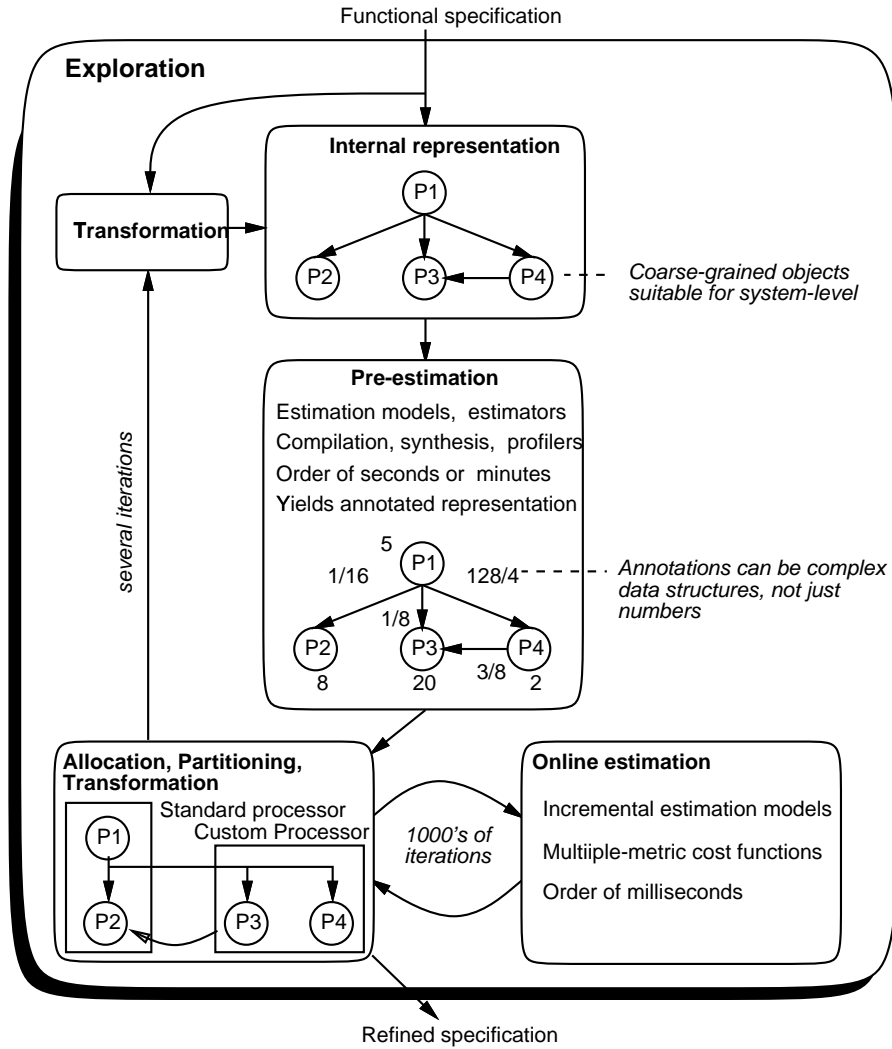


Figure 9: Subtasks during exploration

3.3.1 Pre-estimation and online-estimation

In general, more accurate estimates require more time, but time is very limited during exploration. (When comparing alternative options, fidelity is often more important than accuracy; see [8, 26]). High accuracy can be achieved through synthesis, compilation, and simulation, i.e., by generating a refined description, creating an RT-level design using synthesis and compilation, measuring gates or bytes for size metrics, and performing simulations for performance metrics. However, the minutes or hours required by such an approach makes it unsuitable during exploration, when hundreds to tens of thousands of designs must be examined.

To decrease estimation time, an implementation *model* can be used, which is an implementation abstraction from which metric values can be derived, but which does not contain complete implementation details. SpecSyn uses a two-level technique to obtain

metric values, as illustrated in Figure 9:

1. **Pre-estimation:** Each functional object (behavior, variable and channel) is annotated with information (see Section 2.2), such as the number of bytes for a behavior when compiled to a particular processor, the average frequency of channel access, or the number of channel bits. Pre-estimation occurs only once at the beginning of exploration, is independent of any particular partition and allocation, and may take seconds to minutes.
2. **Online-estimation:** Pre-estimated annotations are combined in complex expressions to obtain metric values for a particular partition and allocation. Online-estimation occurs hundreds or thousands of times during manual or automated exploration, so must usually take just milliseconds.

In most other approaches, exploration consists of only one level of estimation (or two levels where one is trivial), with another level coming only after RT-level design.

We now discuss SpecSyn implementation models for three metric types: performance, hardware size, and software size.

3.3.2 Performance

In SpecSyn’s performance model, a behavior’s execution time is calculated as the sum of the behavior’s *internal computation time* (*ict*) and *communication time*. The *ict* is the execution time on a particular component, assuming all accessed behaviors and variables take zero time. The communication time includes time to transfer data to/from accessed behaviors and variables, as well as the time for such accessed behaviors to execute (e.g., the time for a called procedure to execute and return). This model leads to some inaccuracy, since some computation and communication could occur in parallel, but the model seems to provide reasonable accuracy while enabling rapid estimations.

More precisely, execution time is computed as follows:

$$\begin{aligned}
 b.exe\!ctime &= b.ict_p + b.com\!mtime & (2) \\
 b.com\!mtime &= \sum_{c_k \in b.out\!channels} c_k.acc\!freq \times \\
 &\quad (c_k.tt\!ime_{bus} + (c_k.dst).exe\!ctime) \\
 c_k.tt\!ime_{bus} &= \lceil bus.time \times (c_k.bits \div bus.width) \rceil \\
 bus.time &= bus.timesame \\
 &\quad \text{if } (c_k.dst).p = p, \\
 &= bus.time\!diff \text{ otherwise.}
 \end{aligned}$$

In other words, a behavior’s execution time equals its *ict* on the current component ($b.ict_p$), plus its communication time ($b.com\!mtime$). The communication time equals the transfer time over a channel for each accessed object ($c_k.tt\!ime_{bus}$), plus the execution time of each accessed object ($(c_k.dst).exe\!ctime$), times the number of such accesses ($c_k.acc\!freq$). The transfer time over a channel is determined from the bus data transfer time ($bus.time$) and the width of that bus ($bus.width$); if the data bits exceeds the bus width, then multiple transfers are used (as computed by the division). The $bus.time$ is usually less when the communication is within one component.

FuzzyMain.et	= FuzzyMain.ict
	+ chan1.accfreq * (chan1.tt + in1val.et)
	+ chan2.accfreq * (chan2.tt + in2val.et)
	+ chan3.accfreq * (chan3.tt + EvalRule.et)
	+ chan4.accfreq * (chan4.tt + Convolve.et)
	+ chan5.accfreq * (chan5.tt + Centroid.et)
inv1val.et	= in1val ict + 0
inv2val.et	= in2val ict + 0
EvalRule.et	= EvalRule.ict
	+ chan8.accfreq * (chan8.tt + trunc.et)
	chan9.accfreq * (chan9.tt + Min.et)
	chan10.accfreq * (chan10.tt + mr1.et)
	chan11.accfreq * (chan11.tt + mr2.et)
	chan12.accfreq * (chan12.tt + tmr1.et)
	chan13.accfreq * (chan13.tt + tmr2.et)
Convolve.et	= <i>Convolve.ict</i> +333
Centroid	= <i>Centroid.ict</i> +454
trunc	= trunc ict + 0
Min	= Min ict + 0
mr1	= mr1 ict + 0
mr2	= mr2 ict + 0
tmr1	= tmr1 ict + 0
tmr2	= tmr2 ict + 0

Figure 10: Execution-time equations for the example

Figure 10 shows the execution-time equation for *FuzzyMain* of the earlier example. For simplicity, the example uses fixed numbers for *Convolve* and *Centroid* communication times, whereas actually further equations should be used.

Pre-estimation – A behavior’s internal computation time can be computed during pre-estimation through profiling and scheduling. Profiling determines the execution count of each basic block, where a basic block is a sequence of statements not containing a branch. (Note: SpecSyn currently generates default execution counts, which can then be modified by the user). Each basic block is then scheduled into steps for each possible processor component, using compilation for standard processors and synthesis for custom processors. (Compilation techniques are discussed further in Section 3.3.4). The summation over all blocks of each block’s execution count times steps yields the total steps for the behavior. Multiplying by the step time, i.e., the clock period, yields an ict value. Note that processors using pipelining, caching or interrupts would require further refinements of the ict model. Each behavior is annotated with an ict value for each possible component.

Channel access frequencies are also determined through profiling. Any variable accesses or procedure call parameters can be encoded into bits as during synthesis. Bus times and widths are already associated with each bus.

Figure 5 and Figure 6 showed the annotations obtained during pre-estimation for the fuzzy-controller example.

Online estimation – Given a partition of every functional object to a component,

Object	comp	bind	ict	et	
FuzzyMain	StdProc1	8051		5	8,494
in1val	CustProc1	XC4020		0	0
in2val	CustProc1	XC4020		0	0
EvalRule	CustProc1	XC4020		522	2,197
Convolve	StdProc1	8051		800	1,133
Centroid	StdProc1	8051		2500	2,954
trunc	CustProc1	XC4020		0	0
Min	CustProc1	XC4020		3	3
mr1	Mem1	V100		0	0
mr2	Mem1	V100		0	0
tmr1	Mem2	V100		0	0
tmr2	Mem2	V100		0	0

Figure 11: Evaluating execution times for the example.

the actual ict, bus values, and bus times become known. Thus, a behavior’s execution time equation can be evaluated. When a partitioning heuristic moves an object, the object’s ict value will change, and bus times may also change since objects previously on the same component will now be on different components, and possibly vice-versa. We simply need to change those values and re-evaluate the equation. In addition, any other equations that include the object’s execution time must also be update. If care is taken to maintain links from an object to all terms that change when the object is moved, then the updates can be done very quickly.

Figure 11 shows the results of evaluating the execution time equations for the fuzzy controller example. Using the allocation and partition of Figure 8, each object is assigned to a component (*comp*), each of which was bound to a library item (*bind*); based on this assignment, the current *ict* values are shown. Using these ict’s, and the communication times based on the tt’s (not shown), the execution time equations of Figure 10 are evaluated (*et*). Thus, *FuzzyMain* executes in 8,494 time units for the given allocation and partition.

3.3.3 Hardware size

When several behaviors are assigned to a custom processor, we must estimate the size (e.g, number of gates) required by that processor. The most accurate estimate is achieved by performing synthesis, but as discussed above, such an approach is too slow during exploration. Instead, some tools use a **weight-based** approach, in which pre-estimation consists of annotating each behavior and variable with a weight, and then a simple online-estimation sums the weights [9, 10]. Such an approach is fast, but may be inaccurate since it doesn’t consider hardware sharing. Other research efforts [11, 12, 27, 15] use a **design-based** approach, in which an online-estimation roughly synthesizes a design for a given partition, omitting time-consuming synthesis tasks such as logic optimization. While more accurate, such estimators may require several to tens of seconds, which may be too slow for exploration of thousands of options.

SpecSyn uses an incremental update technique to achieve both the accuracy of design-based estimators and the speed of weight-based estimators. The technique takes advantage

of the fact that many iterative-improvement partitioning heuristics, while exploring thousands of partitions, move only a few objects between one iteration and the next. Thus, using extensive information gathered during pre-estimation, we incrementally modify a custom processor’s design in just milliseconds (constant-time).

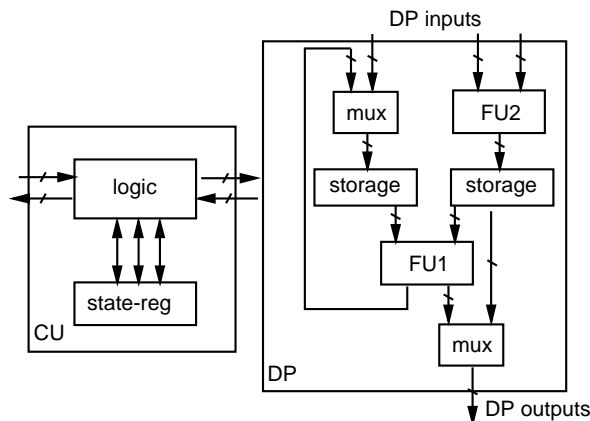


Figure 12: CU/DP area model

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

Figure 13: Equation and terms for computing CU/DP area

SpecSyn uses a hardware design model similar to those in [8, 11, 12], consisting of a control-unit/datapath (CU/DP) as shown in Figure 12. The CU/DP area can be computed as the sum of the following terms: *Functional-unit (FU) size*; *Storage-unit size* including registers, register files and memories; *Multiplexer size*; *State-register size*; *Control-logic size*; and *Wiring-size*. As shown in Figure 13, each term is a function of basic parameters, including the number of possible states, the number of control lines, the number of states each control line is active, the number of bits and words for each storage unit, the number of bits and type of each functional unit, the number of sources of each storage-unit input, functional-unit input, and datapath output, the number of datapath connections, and the number of datapath components. For example functions, see [28].

Pre-estimation – The parameters are computed for each functional object during pre-

estimation, by performing rough synthesis on each object. Each object is then annotated with the computed parameters. Such computation can take seconds or minutes.

Given an initial partition of functional objects among custom processors, we can obtain a rough design of each processor by intelligently combining its objects' parameter annotations. For example, we can determine the number of possible processor states S by summing the objects' possible states, and then create a state register of size $\log(S)$ bits. As another example, we can determine the number of FU's by unioning the objects' FU's (since sequential behaviors can share FU's). Note that the terms, such as state register size and number of FU's, are not obtained by simple addition; in fact, terms may actually be non-linear with respect to the parameters. See [28] for details on computing all the terms from the objects' parameters.

Online-estimation – When a partitioning heuristic removes an object from a processor, we update that processor's terms. Some terms can be updated simply by examining the object's annotations. For example, the number of possible processor states is reduced by the object's number, and the state register size recomputed using the log function. On the other hand, other terms require further examination. For example, an object might require a particular FU, but removing that object only removes that FU if no other object uses the FU; thus, we keep track of which objects use each FU. Likewise, removing an object might not eliminate a multiplexor, but might reduce its size since certain sources are no longer needed; thus, we keep track of which objects require each source. Updating a processor's design for removal of an object, as well as the complementary action of adding an object, can be done in constant time [28].

Note that we can merge the information from the functional objects because of their coarse granularity; otherwise, the ignored inter-object effects would result in poor accuracy.

3.3.4 Software size

A straightforward model of a processor's software size is that of the summation of the processor's functional objects' sizes. While neglecting inter-procedural optimization, such optimization is likely not large, so this model yields fairly accurate estimates.

Pre-estimation – Ideally, we could determine a functional object's size by simply compiling the object for each possible target standard processor, as shown in Figure 14(a). Unless the target processor is the same as the host machine processor on which SpecSyn is running, such compilation will require a cross-compiler, i.e., a compiler that runs on one processor but generates code for another. However, a cross-compiler may not be available on the host machine. For example, suppose the host machine is a Sparc and the target processor an Intel 8051. We probably do not have an 8051 compiler that runs on the Sparc; instead, we probably have one that runs on an x86 processor.

SpecSyn supports a method for estimating software size even when a cross-compiler is not available. The method uses a generic processor model and a single compiler, as shown in Figure 14(b). A functional object's size is first compiled into generic three-address instructions. Using available processor-specific technology files listing the number of bytes that each generic instruction would require in each processor, the estimator computes the software size. A target processor's technology file can be developed based on the size information of the processor's instruction set; note that developing such a file is substantially

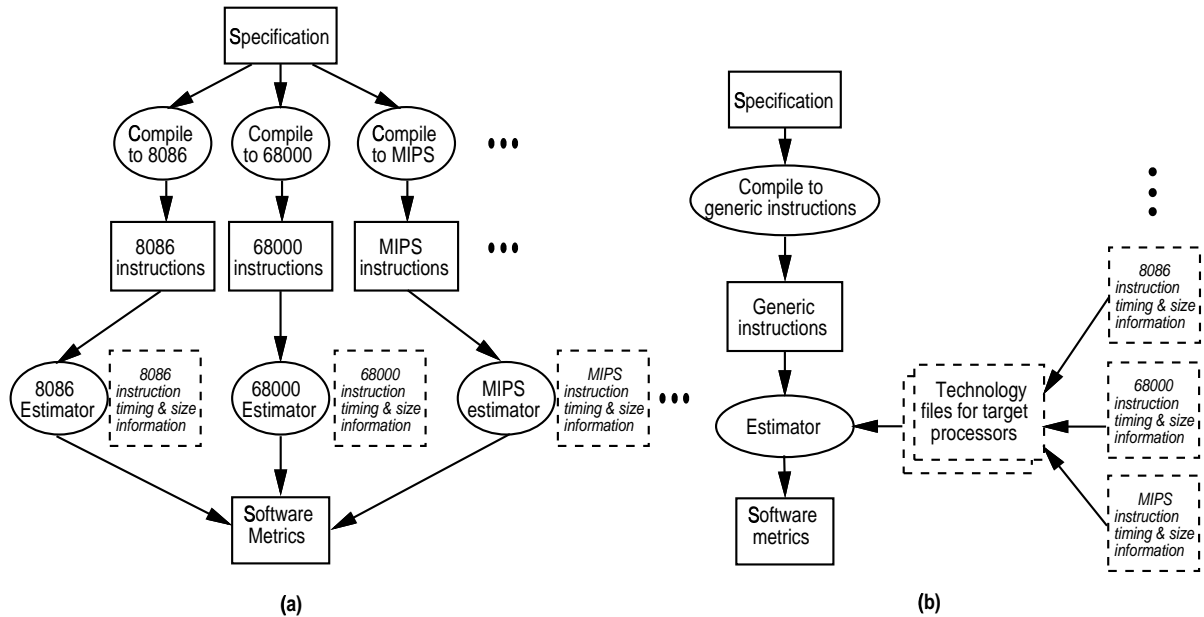


Figure 14: Software size estimation: (a) processor-specific model, (b) generic model.

simpler than developing a back-end compiler. Details on deriving technology files for specific processors are given in [29].

Note that the same generic processor approach would be applied for software performance estimation. Specifically, the technology file of the target processor would include not only the bytes but also the number of steps for each generic instruction.

Some experiments comparing the generic model with the processor-specific model yielded inaccuracy of roughly 7% [29].

Online estimation – Online software size estimation consists simply of increasing or decreasing the processor size by the size of the added or removed functional object.

3.4 Transformations

A functional specification serves the purpose of precisely defining a system’s intended behavior. Such a specification usually will be read by humans as well as input to synthesis tools. Unfortunately, a specification written for readability may not directly lead to the best synthesized design. As a result, designers often try to juggle synthesis considerations with readability considerations while writing the initial functional specification. Such juggling usually leads to lower readability, less portability, and more functional errors; hence, many of the advantages of a top-down approach are greatly diminished, ultimately leading to longer design times.

To solve this problem, SpecSyn provides a suite of automated transformations. As shown in Figure 9, transformations can be applied on the SLIF or on the specification. SLIF transformations occur in an “inner loop” along with allocation, partitioning and online estimation, being applied thousands of times. Specification transformations occur in an “outer loop,” which is followed by rebuilding of the SLIF and re-annotation.

One specification transformation is *procedure exlining* [30]. Exlining is the inverse of procedure inlining; namely, replacing sequences of statements by procedure calls. Since procedures determine SLIF granularity, exlining is a means for achieving finer-granularity. There are two types of exlining. *Redundancy exlining* seeks to find and replace redundant statement sequences. *Distinct-computation exlining* seeks to break a large procedure into several smaller procedures, even though each may only be called once. Redundancy exlining is a very hard problem; presently, we encode each statement into a character string indicating the statement type, symbolic target and sources, concatenate each such string into one large one, and then use the *agrep* approximate pattern matching tool to find potential redundancies. Not all matches found by such an approach are necessarily redundancies, so user interaction is required. Distinct-computation exlining is in fact very similar to the problem addressed in [12]. Statements can be clustered together based on a number of closeness metrics. Simulated annealing can be used to further improve the statement clusterings.

A second specification transformation is *procedure inlining*, which achieves coarser granularity and distributes computations among calling behaviors, eliminating potential computation bottlenecks. Other possible transformations include *process merging* [31], where two processes are sequentialized into one to reduce hardware size, and *process splitting* [32], where one process is split into two concurrent ones. We plan to investigate such process transformations. A variety of other optimizing transformations with origins in software compilation could also be applied [33, 34].

Turning to SLIF transformations, *pre-clustering* [25] merges nodes that should probably never be separated, thus achieving coarser granularity. *Procedure cloning* duplicates procedure nodes so that each calling behavior has its own copy, without necessarily inlining that copy; such cloning is analogous to logic duplication during logic-level partitioning. *I/O node insertion* inserts a node for sending or receiving data to external input/output ports; such nodes enable better distribution of I/O among components, similar in idea to parallel I/O chips. The latter two transformations are currently under investigation.

4 Refinement

Refinement is the generation of a new specification for each system component after exploration has yielded a suitable allocation and partition. The refined specification should be both readable and simulatable, enabling further verification and synthesis. We now describe specification refinement tasks required after system design.

4.1 Interfacing

An important task is interface generation. Abstract communication channels were assigned to physical buses. **Interface refinement** determines the buswidth and the protocol for the bus that will implement the channels. A bus (such as a PC ISA bus) may already have these items fixed, in which case they are simply looked up. Alternatively, a bus may be flexible, in which case the best width and protocol must still be determined; algorithms and techniques have been reported in [35, 36]. After determining the protocol to meet design constraints, structure can be created for the protocol using techniques in [37, 38, 39].

4.2 Memories

Another task is memory refinement associated with the implementation of variables assigned to memories. The variable accesses must be replaced by references to the corresponding memory locations.

4.3 Arbitration

A third task, arbiter generation, inserts an arbiter behavior where there is a resource contention, i.e., where two behaviors could access the same memory or bus simultaneously.

Note that, while during partitioning we abstracted communication implementation to the problem of mapping channels to buses, during refinement we must now deal with more complex communication issues involving protocols and arbitration. Such complex communication results in new behaviors (protocols and arbiters), which may later be synthesized, or possibly mapped to existing communication components like USART's or direct-memory-access controllers.

4.4 Generation

The final task of refinement is the actual generation of a refined description. The new description should be readable, modifiable, simulatable, and synthesizable. We use the following technique to generate a refined description. First, we create a VHDL entity for each system component. Second, for each behavior that represented a process in the original specification, we create a VHDL process inside the component to which the behavior has been assigned. Third, we describe activation for separated behaviors, i.e., those behaviors that have been assigned to a component different from their calling behaviors. The simplest approach to achieving such activation would be to create a single process for each such behavior, where the process would wait until it was activated via a control signal, would execute the behavior, and then would indicate completion via another control signal. However, such an approach results in an excessive number of processes (one for each separated behavior) and control signals. A better approach is to combine all separated behaviors that we know to be sequential (i.e., all those behaviors that belong to the same process), and that have been assigned to the same component, into a single process. This process would wait until it was activated, would execute one of its behaviors based on a newly-introduced mode signal, and would indicate its completion. Fourth, we insert communication protocols and arbiters, as described above. We use VHDL send/receive procedures to hide the protocol details, and use additional VHDL processes to describe the arbiters.

4.5 Validation

To verify the system-design decisions, we can simulate the refined specification. When certain components use different models of computation than other components or contain different levels of details, different simulation approaches must be combined to obtain a simulation of the complete system. Such combination is called cosimulation. A variety of approaches to cosimulation are described in the literature, such as in [40, 41, 42, 43]. The refined specification can serve as input to most of these approaches.

In Figure 15, we show a refined specification for the system design shown in Figure 8. Due to space limitations, the figure shows only a part of the refined specification. The interface of the fuzzy controller remains unchanged. However, its contents now consist of many more details than in the original specification of Figure 3. For example, the top-level view of the controller now consists of instantiations of an ASIC, two memories, and a processor component, along with the interconnections among those components. The ASIC component, in turn, is defined as an entity with several ports. The first two ports, *in1* and *in2*, simply connect with the external inputs with the same names. The next two ports, *startEvalRule* and *doneEvalRule*, would be used by the *FuzzyMain* process on the processor to activate the *EvalRule* procedure on the ASIC. The last three ports shown, *num_chan*, *mr_chan* and *tmr_chan*, are composite data types that describe the signals necessary for fetching the *num* parameter from *FuzzyMain*, for fetching *mr1* and *mr2* data from *Memory1*, and for storing *tmr1* and *tmr2* data to *Memory2*. The ASIC's behavior consists of a single process, which waits for an activation signal, fetches the *num* parameter, and calls *EvalRule* with that parameter.

EvalRule is a procedure found in this process, identical to the procedure in Figure 3, except that the *mr* and *tmr* arrays can no longer be accessed as global variables. Instead, they must be accessed using new subroutines that read data from *Memory1* and write data to *Memory2*. Those subroutines describe the detailed communication protocol for such memory accesses, and would usually be found in a communication-protocol VHDL package. Note that since the *mr1* and *mr2* arrays have been merged into the same memory *Memory1*, offsets (*MR1OFFSET* and *MR2OFFSET*) must be added to any array addresses; likewise for *tmr1* and *tmr2*, which both reside in *Memory2*.

There are two important points to note in this example. First, note the large amount of detail that must be added to the specification as a result of creating a system-level design. Presently, designers must handwrite this detail, resulting in longer specification times. Moreover, if the system-level design serves as the first captured specification, then we can expect many more functional errors, since the specification writer must consider many detailed issues that detract from a focus on the system's functionality. Second, it is crucial that the designer be given access to these newly-introduced details. Many of those details involve important design decisions that the designer must be aware of and must be able to change; for these reasons, generation of a refined specification can be seen as extremely important. After refinement, the functional specification of each component is just that – a specification, not an implementation. This means that for a software component (as well as a hardware component), there may be more than one process in the component's functional specification. These processes will need to be merged into a single control thread, but such merging is part of the implementation task for the component. Thus, the refined specification is a unique and important intermediate representation of functionality, necessary to verify the system-level allocation and partitioning decisions we have made, without yet requiring detailed implementation decisions for each component. Further details on refinement can be found in [8, 44].

```

entity FuzzyControllerE is
  port ( in1, in2 : in integer; out1: out integer );
end;
...
component ASIC1E is
  port ( in1, in2 : in integer; startEvalRule : in bit;
        doneEvalRule : out bit; num_chan : int_chan;
        mr_chan, tmr_chan : addr_int_chan; ...);
end;
component Memory1E is
  port (mr_chan : addr_int_chan);
end;
component Memory2E is ...
component Processor1E is ...
< port maps > ...
...
entity ASIC1E is
  port ( in1, in2 : in integer; startEvalRule : in bit;
        doneEvalRule : out bit; num_chan : int_chan;
        mr_chan, tmr_chan : addr_int_chan; ...);
end;
...
process
  variable in1val, in2val : integer;
  function Min ...
  variable num : integer;
  ...
begin
  wait until startEvalRule='1';
  num := ReadNum(num_chan);
  EvaluateRule(num);
  doneEvalRule='1';
  ...
end;
...
procedure EvaluateRule(num : in integer) is
  variable trunc : integer; -- truncated value
  variable mr_val1, mr_val2 : integer;
  variable tmr_val : integer;
begin
  if (num = 1) then
    mr_val1 := ReadMemory1(in1val + MR1OFFSET);
    mr_val2 := ReadMemory1(128 + in1val + MR1OFFSET);
    trunc := Min(mr_val1, mr_val2);
  elsif (num = 2) then
    mr_val1 := ReadMemory1(in1val + MR2OFFSET);
    mr_val2 := ReadMemory1(128 + in1val + MR2OFFSET);
    trunc := Min(mr_val1, mr_val2);
  end if;
  for i in 1 to 128 loop
    if (num = 1) then
      mr_val1 := ReadMemory1(256+i + MR1OFFSET);
      tmr_val := Min(trunc, mr_val1);
      WriteMemory2(i + TMR1OFFSET, tmr_val);
    elsif (num = 2) then
      mr_val1 := ReadMemory1(256+i + MR2OFFSET);
      tmr_val := Min(trunc, mr_val1);
      WriteMemory2(i+TMR2OFFSET, tmr_val);
    end if;
  end loop;
end;
end;

```

Figure 15: Refined fuzzy-controller VHDL partial specification.

5 Experiments

We have conducted a series of experiments to explore design alternatives for several industrial examples. Here we present results for one particular example: a fuzzy-logic controller [45].

Four library components were available: a standard processor (Intel 8051) and three custom processors with 50k, 100k, and 150k gates. Each component had an associated dollar cost. For the experiment, we automatically generated all possible allocation combinations of these components below a certain dollar cost. For each allocation, we partitioned automatically using simulated annealing and a cost function that sought to meet all size and pin constraints while minimizing execution time.

Figure 16 shows results for the fuzzy-logic controller for 35 different allocations. Allocation 1 consisted of just the 8051 standard processor, and had an execution time of over 150,000 microseconds, so its point is not shown on the graph. Allocation 2 consisted of just one 50k gate custom processor, but the processor's size constraint was violated so that point is not shown either. Allocation 3 consisted of one 8051 standard processor and one 50k gate custom processor, resulting in an execution time of 18,115 us. Allocation 4 consisted of just one 100k gate processor, but again this resulted in a size violation, so the point is not shown. Allocation 5 consisted of one 8051 and one 100k gate custom processor, yielding an execution time of 7721 us. Subsequent higher-cost allocations yield no better execution time. For example, allocation 31 consisted of one 8051, one 50k custom processor, one 100k custom processor, and two 150k custom processors, and yielded an execution time of

9785 us. Conceptually, we should have been able to achieve 7721 us by just using the 100k custom processor, but the simulated annealing formulation simply did not find a solution using just that custom processor along with the 8051; instead, functions were assigned to multiple custom processors, requiring inter-processor communication and hence the longer execution time.

SpecSyn thus aids the designer to get a feel for the design space, enabling him to focus on promising points. The above data was generated automatically in 1 hour running on a Sparc 2. There are numerous other types of tradeoffs that can also be generated.

SpecSyn was used by an industry engineer to design the fuzzy-logic controller. The partitioning results obtained matched favorably with those obtained by another engineer who did a manual partition. The system-level design obtained by SpecSyn consisted of 5 FPGAs. Each was implemented using high-level synthesis, and NeoCAD tools were used to complete the design. Details of this experiment can be found in [45]. We summarize them briefly in Figure 17. The SpecCharts language was used for the initial specification. Note the reduction in the number of lines when using SpecCharts as opposed to VHDL for the specification (see [8] for other experiments which demonstrate the reduction in specification time, specification errors, comprehension time, and lines of code). Also note the large increase in the number of lines for the refined specification; since this is automatically generated, the designer is relieved from the tedious effort of having to write the refined specification himself. Finally, note the very large size of the VHDL after its structural implementation; such a large amount of information is very difficult to work with, so starting with a functional specification enables a tremendous increase in designer comprehension. The entire implementation was obtained in roughly 100 man-hours with the aid of SpecSyn and high-level synthesis, which is nearly a 10 times reduction in design time from the 6-months required to obtain the design manually.

6 Related work, current status, and future work

Several other system-level design environments have also evolved. TOSCA [46, 47] focuses on control-dominated systems. A hierarchical FSM input is converted to a process algebra internal format based on a CSP-like model, which is partitioned among an architecture consisting of a standard processor, memory, system bus, and some number of custom processors that can share local buses. Partitioning is performed manually or using a hierarchical clustering heuristic, incorporating formal transformations such as parallelization. Several metrics guide the partitioning. Processes partitioned to software are output in a virtual instruction set (VIS), which is later translated for a particular processor, thus achieving some processor independence. Synthesis is applied to the output and the results used to guide further iterations. The VIS is similar to SpecSyn's generic instruction set, except that SpecSyn only uses the set for estimation purposes; SpecSyn outputs software at the algorithmic level, in accordance with the SER methodology.

COSYMA [9] focuses on microcontroller-based systems. An extended C input is converted to a basic-block and statement-level graph, which is partitioned among an architecture consisting of a standard processor, custom processor, memory and bus. Fast indirect metrics guide the simulated annealing partitioning, the resulting implementation is then analyzed using more complex metrics, and the results are used to guide further iterations.

Vulcan II [10] uses a similar architecture and applies a greedy partitioning heuristic with fast indirect metrics. Recent focus has been on analyzing input constraints for use during partitioning and synthesis.

A large number of other approaches exist. Summaries can be found in [48, 26].

SpecSyn's possesses many unique features, several of which are most salient. First, SpecSyn outputs a system-level description in order to support the SER methodology. Second, SpecSyn is intended to support a variety of system architectures, heuristics, estimation models, and cost functions; no one version of any of these items is advocated for all possible systems. For example, a suite of heuristics is provided, with easy ability to add new ones. Third, SpecSyn uses a two-level estimation method in which considerable effort is spent on both pre-estimation and online-estimation.

SpecSyn currently consists of over 150,000 lines of C code, and has been under development since 1989. Its main interface consists of a spreadsheet-like display showing each component and functional object along with annotations, constraints and metric values for each. Menu options permit designers to perform any of a number of design tasks, whose results are then reflected by updating values in the display; violated constraints are flagged for the user. SpecSyn has been released to several universities and to over 20 companies, and experiments with industry examples are ongoing.

Some limitations lend themselves to future work. First, SpecSyn does not currently support scheduling of the coarse-grained behaviors on the processors to which they are assigned, since in manual design, the system-level allocation and partition decisions are usually made before such scheduling decisions. However, in an automated approach, such scheduling might prove useful. Second, SpecSyn does not currently incorporate the post-synthesis metric values back into subsequent explore/refine iterations. Such incorporation could prove very useful. Third, a method should be introduced to allow designers to provide manual metric estimations. Such a method could be as simple as accepting numbers for use during pre-estimation, or as complex as using designer-defined expressions for combining annotations during online-estimation. Fourth, a method for design from partial specifications should be implemented. The method for allowing manual metric estimations would likely form a large part of this method. Fifth, estimation models must be continually improved to account for additional architectural features, such as pipelining, caching, and real-time operating systems, and to model fixed-processors like DMA controllers, Fourier transform blocks, Ethernet controllers, MPEG decoders, etc. Some work on pipelining has been reported in [49, 50]. Sixth, exploration might be improved by considering ranges of designs during partitioning, rather than a single point in the range as is currently done. Seventh, transformations, such as parallelization, need to be developed and integrated with partitioning, as they play a key role in enabling good final implementations. Eighth, as package borders continually change and more components find their way onto a chip, a general method of partitioning and estimating for hierarchical components needs to be developed. Finally, a variety of input languages, such as C, Statecharts, and synchronous-dataflow-based languages, need to be supported.

7 Conclusions

We have introduced a specify-explore-refine paradigm for system design. Our specification technique focuses on understandable specifications, which in turn encourages the use of front-end languages such as SpecCharts. Our approach to exploration uses pre-estimation and online-estimation to achieve both fast and accurate estimates, supports a variety of partitioning heuristics, and is intended to be continually extended, enabling a designer to examine numerous alternative designs quickly. Our refinement techniques automatically insert details into the specification that would otherwise have been manually written by the designer, thus relieving the designer of tedious effort. We expect that this paradigm and tool will eventually result in a 100-hour design cycle, and our experiments demonstrate the feasibility of such a dramatic reduction in design time from current practice.

References

- [1] D. Gajski, N. Dutt, C. Wu, and Y. Lin, *High-Level Synthesis: Introduction to Chip and System Design*. Boston, Massachusetts: Kluwer Academic Publishers, 1991.
- [2] J. Vanhoof, K. VanRompae, I. Bolsens, and H. DeMan, *High-level synthesis for real-time digital signal processing*. Boston: Kluwer Academic Publishers, 1993.
- [3] G. DeMicheli, A. Sangiovanni-Vincentelli, and P. Antognetti, *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*. Martinus Nijhoff Publishers, 1987.
- [4] C. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.
- [5] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring, "STATEMATE: A working environment for the development of complex reactive systems," in *Proceedings of the International Conference on Software Engineering*, pp. 396–406, 1988.
- [6] E. Lee and D. Messerschmitt, "Synchronous data flow," in *Proceedings of the IEEE*, pp. 1235–1245, 1987.
- [7] F. Vahid, S. Narayan, and D. Gajski, "SpecCharts: A VHDL front-end for embedded systems," in *IEEE Transactions on Computer-Aided Design*, pp. 694–706, 1995.
- [8] D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and design of embedded systems*. New Jersey: Prentice Hall, 1994.
- [9] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," in *IEEE Design & Test of Computers*, pp. 64–75, December 1994.
- [10] R. Gupta and G. DeMicheli, "Hardware-software cosynthesis for digital systems," in *IEEE Design & Test of Computers*, pp. 29–41, October 1993.
- [11] M. McFarland and T. Kowalski, "Incorporating bottom-up design into hardware synthesis," *IEEE Transactions on Computer-Aided Design*, pp. 938–950, September 1990.
- [12] E. Lagnese and D. Thomas, "Architectural partitioning for system level synthesis of integrated circuits," *IEEE Transactions on Computer-Aided Design*, vol. 10, pp. 847–860, July 1991.
- [13] 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.
- [14] Y. Chen, Y. Hsu, and C. King, "MULTIPAR: Behavioral partition for synthesizing multi-processor architectures," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 2, pp. 21–32, March 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] 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.
- [17] 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.
- [18] F. Vahid and D. Gajski, "Specification partitioning for system design," in *Proceedings of the Design Automation Conference*, pp. 219–224, 1992.
- [19] F. Vahid, T. Le, and Y. Hsu, "A comparison of functional and structural partitioning," in *International Symposium on System Synthesis*, 1996.
- [20] 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.
- [21] 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.
- [22] P. Eles, Z. Peng, and A. Daboli, "VHDL system-level specification and partitioning in a hardware/software co-synthesis environment," in *International Workshop on Hardware-Software Co-Design*, pp. 49–55, 1992.
- [23] 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.
- [24] J. D'Ambrosio and X. Hu, "Configuration-level hardware/software partitioning for real-time embedded systems," in *International Workshop on Hardware-Software Co-Design*, pp. 34–41, 1994.
- [25] F. Vahid and D. Gajski, "Clustering for improved system-level functional partitioning," in *International Symposium on System Synthesis*, pp. 28–33, 1995.
- [26] D. Gajski and F. Vahid, "Specification and design of embedded hardware-software systems," in *IEEE Design & Test of Computers*, pp. 53–67, 1995.
- [27] J. Rajan and D. Thomas, "Synthesis by delayed binding of decisions," in *Proceedings of the Design Automation Conference*, 1985.
- [28] 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.
- [29] J. Gong, D. Gajski, and S. Narayan, "Software estimation using a generic processor model," in *Proceedings of the European Design and Test Conference (EDTC)*, pp. 498–502, 1995.
- [30] F. Vahid, "Procedure exlining: A transformation for improved system and behavioral synthesis," in *International Symposium on System Synthesis*, pp. 84–89, 1995.
- [31] J. Hagerman and D. Thomas, "Process transformation for system level synthesis." Technical Report CMUCAD-93-08, 1993.
- [32] R. Walker and D. Thomas, "Behavioral transformation for algorithmic level IC design," *IEEE Transactions on Computer-Aided Design*, pp. 1115–1128, October 1989.
- [33] A. Nicolau and R. Potasman, "Incremental tree height reduction for high level synthesis," in *Proceedings of the Design Automation Conference*, pp. 770–774, 1991.
- [34] M. Girkar and C. Polychronopoulos, "Automatic extraction of functional parallelism from ordinary programs," in *IEEE Transactions on Parallel and Distributed Systems*, pp. 166–178, 1992.

- [35] S. Narayan and D. Gajski, "Synthesis of system-level bus interfaces," in *Proceedings of the European Conference on Design Automation (EDAC)*, 1994.
- [36] S. Narayan and D. Gajski, "Protocol generation for communication channels," in *Proceedings of the Design Automation Conference*, pp. 547–551, 1994.
- [37] G. Borriello and R. Katz, "Synthesis and optimization of interface transducer logic," in *Proceedings of the International Conference on Computer-Aided Design*, 1987.
- [38] J. Akella and K. McMillan, "Synthesizing converters between finite state protocols," in *Proceedings of the International Conference on Computer Design*, 1991.
- [39] J. Sun and R. Brodersen, "Design of system interface modules," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 478–481, 1992.
- [40] D. Becker, R. Singh, and S. Tell, "An engineering environment for hardware/software co-simulation," in *Proceedings of the Design Automation Conference*, pp. 129–134, 1992.
- [41] R. Gupta, C. Coelho, and G. DeMicheli, "Synthesis and simulation of digital systems containing interacting hardware and software components," in *Proceedings of the Design Automation Conference*, pp. 225–230, 1992.
- [42] A. Kalavade and E. Lee, "A hardware/software codesign methodology for DSP applications," in *IEEE Design & Test of Computers*, 1993.
- [43] S. Sutarwala and P. Paulin, "Flexible modeling environment for embedded systems design," in *International Workshop on Hardware-Software Co-Design*, pp. 124–130, 1994.
- [44] J. Gong, D. Gajski, and S. Bakshi, "Model refinement for hardware-software codesign," in *Proceedings of the European Design and Test Conference (EDTC)*, 1996.
- [45] L. Ramachandran, D. Gajski, S. Narayan, F. Vahid, and P. Fung, "Towards achieving a 100-hour design cycle: A test case," in *Proceedings of the European Design Automation Conference (EuroDAC)*, pp. 144–149, 1994.
- [46] A. Balboni, W. Fornaciari, and D. Sciuto, "Partitioning and exploration strategies in the toasca co-design flow," in *International Workshop on Hardware-Software Co-Design*, pp. 62–69, 1993.
- [47] 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.
- [48] W. Wolf, "Hardware-software co-design of embedded systems," *Proceedings of the IEEE*, vol. 82, no. 7, pp. 967–989, 1994.
- [49] S. Bakshi and D. Gajski, "A component selection algorithm for high-performance pipelines," in *Proceedings of the European Design Automation Conference (EuroDAC)*, pp. 400–405, 1994.
- [50] S. Bakshi and D. Gajski, "A memory selection algorithm for high-performance pipelines," in *Proceedings of the European Design Automation Conference (EuroDAC)*, pp. 124–129, 1994.

This work was supported by the National Science Foundation (grant #MIP-8922851) and the Semiconductor Research Corporation (grant #92-DJ-146).

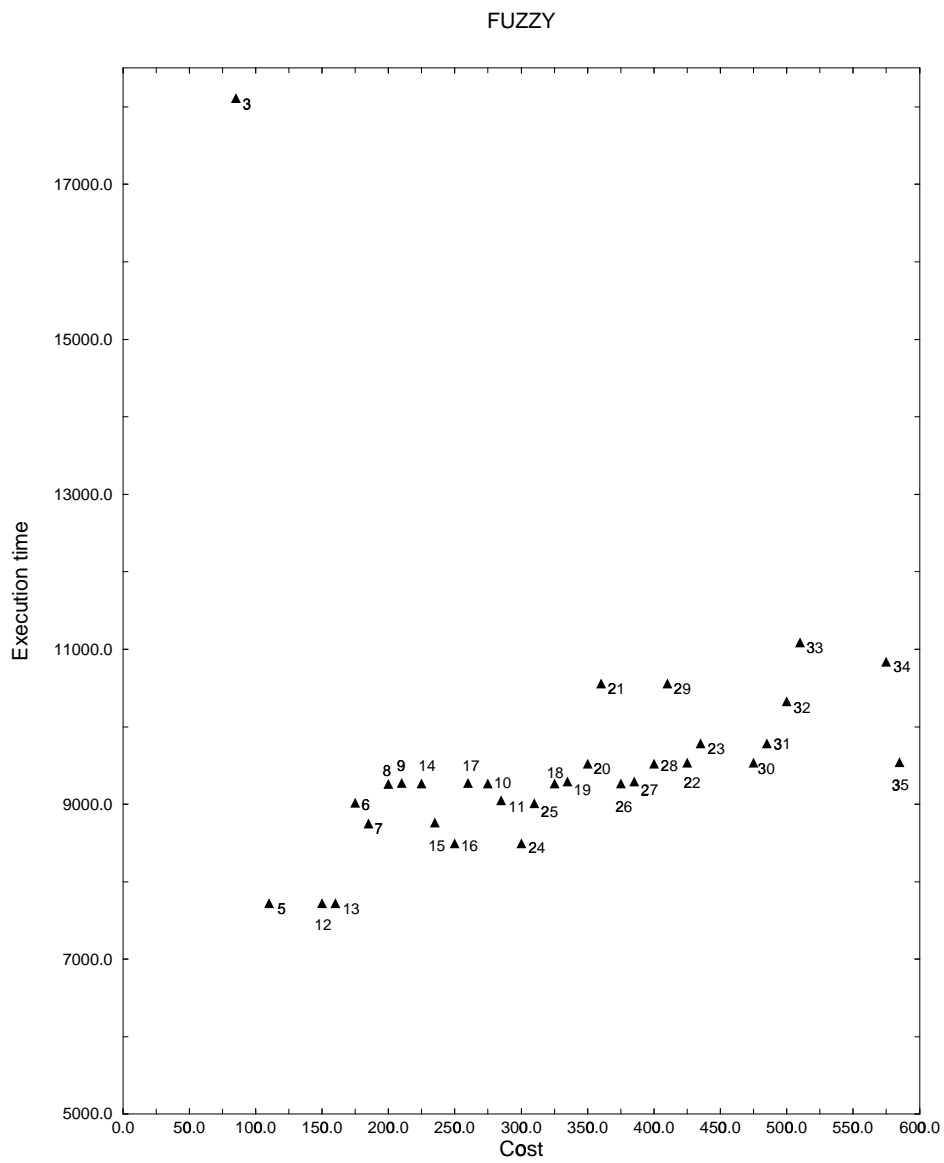


Figure 16: Exploration for the fuzzy logic controller

SpecCharts spec	350
VHDL spec	598
Refined VHDL spec	1495
Structural VHDL spec	17500

Figure 17: Fuzzy logic controller industry design summary