System-Level Exploration with SpecSyn

Daniel D. Gajski
Inform. and Computer Science
University of California
Irvine, CA 92717
gajski@ics.uci.edu

Frank Vahid
Computer Science and Eng.
University of California
Riverside, CA 92521
vahid@cs.ucr.edu

Sanjiv Narayan
Ambit Design Systems
2500 Augustine Drive
Santa Clara, CA 95054
sanjiv@ambit.com

Jie Gong
Qualcomm Inc
6455 Lusk Boulevard
San Diego, CA 92121
jiej@qualcomm.com

Abstract

We present the SpecSyn system-level design environment supporting the specify-explore-refine (SER) design paradigm. This three-step approach 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 like standard and custom processors, and a partitioning of functionality among those components. Focusing on SpecSyn's exploration techniques, we emphasize its two-phase estimation approach and highlight experiments using SpecSyn.

1 Introduction

The focus of design effort on higher abstraction levels, driven by increasing system complexity, shorter design times, and migration of entire systems onto a single chip, demands a system-level design methodology and supporting tools. We can isolate three tasks in such a methodology. First, we must specify the system's functionality and constraints. Second, we must explore various system-level design alternatives, each an interconnection of system components and an assignment of functionality to them. System components include standard software processors, custom hardware processors, memories, and buses. Third, we must refine the original specification into a new system-level description with components. Subsequently, we implement each component, where software components require compilation and possibly thread scheduling, while hardware components require behavioral and register-transfer synthesis.

In current practice, these three tasks are carried out in an informal manner. The *specify-explore-refine* paradigm aims to provide a more precise approach to these tasks, enabling automated assistance. We have developed the SpecSyn environment, illustrated in Figure 1, to support the specify-explore-refine paradigm. In this paper, we focus on SpecSyn's design exploration.

2 Related work

Several system-level design environments have evolved recently. TOSCA [1] focuses on control-dominated systems. It parses a hierarchical finite-state machine into a process algebra internal format, which it partitions among system components manually or by clustering, supporting some formal transformations. It outputs software in a virtual instruction set to achieve processor independence. COSYMA [2] focuses on microcontroller-based systems. It converts an extended C input to a syntax graph, dynamically modifying the graph's granularity [3]. Fast indirect design metrics guide a simulated annealing partitioning among a software and hardware component with shared memory, with direct design metrics from a subsequent implementation guiding further iterations. Vulcan II [4] uses a similar architecture and applies a greedy partitioning heuristic with fast indirect metrics. Ptolemy [5] focuses on cosimulating inputs from different computation models, but also partitions a task-level dataflow graph among a similar architecture using a custom heuristic. Approaches in [6, 7] simultaneously allocate processors from a diverse library while partitioning a task graph among them. Summaries of these and other approaches can be found in [8]. Recent ideas on the system exploration problem were provided in [9], and to a refinement-based approach in [10].

SpecSyn possesses several unique features. First, SpecSyn uses a two-level estimation method to obtain fast yet accurate estimations, using direct design metrics, like total hardware size or process execution time, rather than indirect metrics. Second, SpecSyn supports a variety of system architectures, heuristics, estimation models, and cost functions; no one version of any of these is advocated for all systems. For example, a suite of heuristics is provided, and new ones can be added. Third, SpecSyn outputs a system-level description that ideally can be simulated, edited, and synthesized, thus supporting the SER methodology.

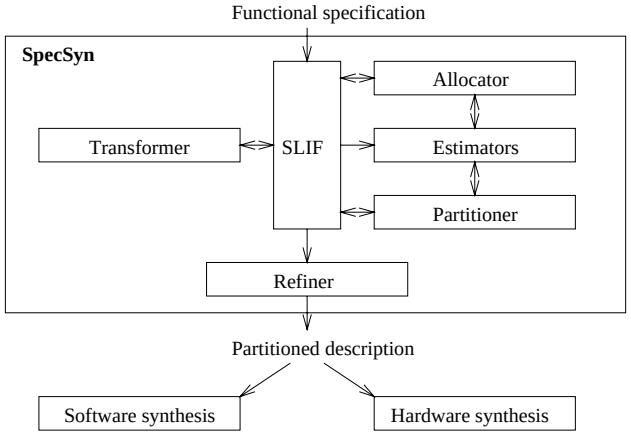

Fig. 1: The SpecSyn system-design environment

3 Specification

3.1 Computation model

We have focused on the Program-State Machine (PSM) computation model [11]. PSM can be thought of as Statecharts [12] with arbitrarily complex sequential programs in the leaf states. Thus, PSM represents a powerful computation model for both hardware and software, able to describe

1-58113-049-x-98/0006/$3.50

hierarchical/concurrent finite-state machines, a sequential program, or even communicating sequential processes.

The designer can specify constraints on behavior execution times and channel bitrates. Other design constraints, such as a component size and I/O limitations, are derived from each component's library entry.

3.2 Internal representation

The specification must be converted into an internal representation on which subsequent tools can operate. Because system specifications tend to be highly procedural, we use a representation similar to a call-graph used in software compilation, introduced by an example. Figure 2 shows a partial VHDL specification of a fuzzy-logic controller. Inputs *in1* and *in2* must be converted to output *out1* using fuzzy logic. The main process *FuzzyMain* repeatedly reads inputs into variables *in1val* and *in2val*, twice calls procedure *EvalRule* to fill arrays (*tmr1* or *tmr2*) based on the input and on another predefined array (*mr1* or *mr2*), convolves the *tmr* arrays, and computes and outputs a centroid value.

```
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;
```

Fig. 2: Fuzzy-logic controller example.

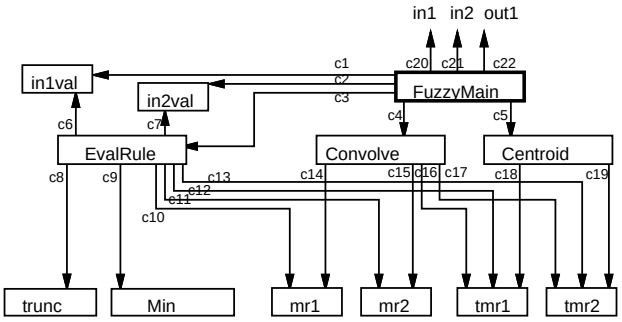

Fig. 3: Access graph for the example.

Figure 3 shows the internal representation. Each graph node represents a **behavior** or **variable**, where a behavior is a process or procedure, though for finer granularity we can exline [13] statement blocks like loops into new procedures. Each graph directed-edge represents a communication **channel**, which is a procedure call, a variable or 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. We call this the **Specification-level intermediate format (SLIF)**, since

Object	ict.8051	ict.xc4020	size.8051	size.xc4020
FuzzyMain	6	8	80	500
in1val	0	0	2	80
in2val	0	0	2	80
EvalRule	778	522	500	1600
Convolve	800	600	900	2000
...				

Object	accreq	bits	src	dst
c1	1	6	FuzzyMain	in1val
c2	1	6	FuzzyMain	in2val
c3	2	9	FuzzyMain	EvalRule
c4	1	0	FuzzyMain	Convolve
c5	1	0	FuzzyMain	Centroid
c6	1	16	EvalRule	in1val
...				

Fig. 4: Slif behavior/variable and channel annotations

its granularity is that of objects explicit in the specification. The SLIF part shown above is an **access graph**, or **AG**, since it represents the accesses among objects.

The SLIF is annotated with numerous values, as shown in Figure 4. A behavior and variable object has a list of size weights, one weight for each component type to which the object may be assigned. For example, a variable object has the number of memory words for each library memory component, while a behavior has square microns, gates, combinational-logic blocks, and bytes for each custom chip, ASIC, FPGA, and standard processor, respectively, in the component library. Behaviors and variables are annotated with computation times on each component. Each edge has access frequency weights obtained through profiling, and a bit weight representing the number of bits per transfer. Annotations are computed during pre-estimation, and are combined into quality metric estimates during online estimation, as discussed in Section 4.3.

4 Exploration

Exploration is the task of finding a set of potential architectures that satisfy constrained metrics and optimize other metrics. It consists of allocation, partitioning, transformation and estimation. These problems can be solved in various orders, and we usually iterate several times.

4.1 Allocation

Allocation is the task of adding components to the design. The SpecSyn allocator permits allocation of any number of standard processors, custom processors, memories, and buses. 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 [11]. 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. Allocation is currently manual, though simple scripts can be used to automatically sequence through numerous possible allocations and apply partitioning.

Figure 5 demonstrates an example allocation. *Standard-Proc1* 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.

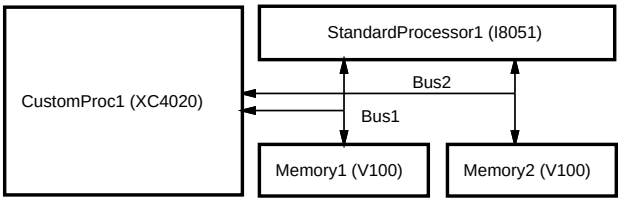

Fig. 5: An example allocation of components

4.2 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. *Variables* store data values and are assigned to memory components. *Behaviors* transform data values and are assigned to processors. *Channels* transfer data from one behavior to another and are assigned to buses.

SpecSyn uses a partitioning engine that takes a data structure, cost function, and data structure update procedure as inputs, and applies any of numerous built-in heuristics. In this way, heuristics, data structures and cost functions stay distinct, and thus can easily be added or improved.

4.2.1 Manual partitioning and hints

SpecSyn supports designer interaction by providing the ability to manually relocate objects, allowing user control of the relative weights of various metrics in the cost function, and automatically providing hints of what changes might yield improvements to the current partition. SpecSyn currently supports two types of hints.

Closeness hints are based on a weighted function of various closeness metrics. Seven behavior closeness metrics have been defined. For example, *connectivity* is based on the number of wires shared between the sets of behaviors. Grouping behaviors that share wires should result in fewer pins. Three variable/channel closeness metrics have also been defined, such as *sequentiality of access* of the variables/channels by behaviors, which aims to group items to avoid resource conflicts and performance degradation.

The second type of hints are *lookahead* hints. 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.

4.2.2 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:

$$\begin{aligned} Costfct &= k_1 \cdot F(c1.size, c1.size\_constr) \\ &+ k_2 \cdot F(c2.size, c2.size\_constr) \\ &+ k_3 \cdot F(c1.IO, c1.IO\_constr) \\ &+ k_4 \cdot F(b1.exectime, b1.exectime\_constr) \\ &\ldots \end{aligned} \quad (1)$$

where the k's are user-provided constants indicating the relative importance of each metric, F is a function indicating the desirability of a metric's value, $c1, c2$ are components, and $b1$ is a behavior. A common form of F returns the degree of constraint violation, normalized such that 0 indicates no violation, and 1 indicates very large violation.

The above cost function is very general, permitting us to satisfy constraints as well as to optimize certain metrics (through heavier weights), without requiring specific knowledge in a heuristic of the constraints or optimization metrics.

As an example of the results of partitioning, Figure 6 shows a partition of several of the previous example's nodes among two memories, a custom processor, a standard processor and a bus. Note that four communication channels have been partitioned onto *bus1*.

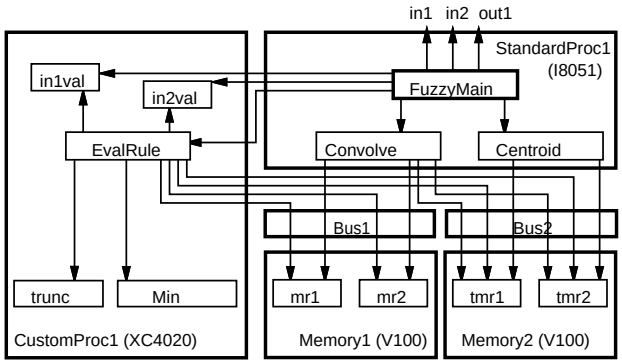

Fig. 6: Partitioning AG nodes among components

4.3 Estimation

Estimation of design quality metrics is required to determine if a particular system-level design (a partition of functions among allocated components) satisfies constraints, and to compare alternative designs.

SpecSyn uses a two-level technique to obtain fast yet accurate estimates of design metrics, as illustrated in Figure 7:

1. **Pre-estimation**: Each functional object (behavior, variable and channel) is annotated with information (see Section 3.2), such as the average frequency of channel access. Pre-estimation occurs only once at the beginning of exploration and is independent of any particular partition and allocation.
2. **Online-estimation**: Pre-estimated annotations are combined in complex expressions to rapidly obtain metric values for a particular partition and allocation (usually in constant time [14]). Online-estimation occurs hundreds or thousands of times during manual or automated exploration.

In most other approaches, exploration consists of only one level of estimation, with another level coming only after RT-level design. We now discuss SpecSyn estimation models for three metric types: performance, hardware size, and software size.

4.3.1 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

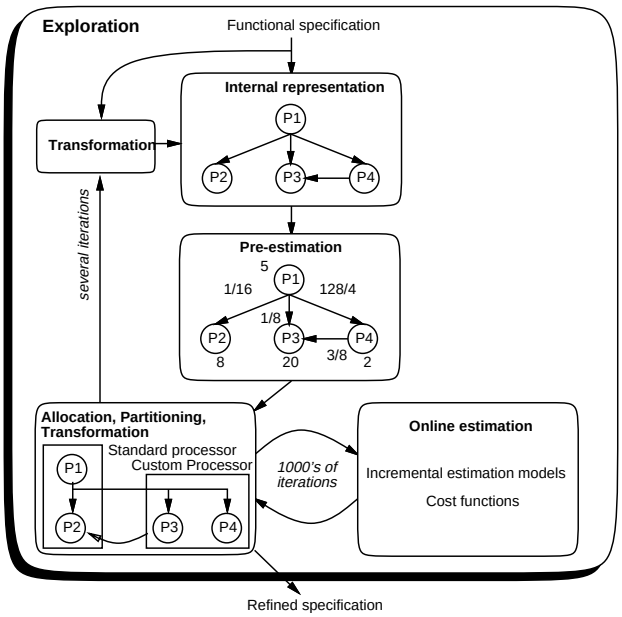

Fig. 7: Subtasks during exploration

execution time on a particular component, assuming all accesses to other 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).

More precisely, execution time is computed as follows:

$$\begin{aligned} b.exectime &= b.ict + b.commtime \qquad (2)\\ b.commtime &= \textstyle\sum_{c_k \in b.outchannels} c_k.accfreq \times \\ &\quad (c_k.ttime_{bus} + (c_k.dst).exectime) \\ c_k.ttime_{bus} &= \lceil bus.time \times (c_k.bits \div bus.width) \rceil \end{aligned}$$

In other words, a behavior b's execution time equals its *ict* on the current component ($b.ict$), plus its communication time ($b.commtime$). The communication time equals the transfer time over a channel for each accessed object ($c_k.ttime_{bus}$), plus the execution time of each accessed object ($(c_k.dst).exectime$), times the number of such accesses ($c_k.accfreq$). 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.

For example, the execution-time equation for *FuzzyMain* of Figure 3 would be:

$$\begin{aligned} FuzzyMain.et &= FuzzyMain.ict \\ &+ c1.accfreq * (c1.tt + in1val.et) \\ &+ c2.accfreq * (c2.tt + in2val.et) \\ &+ c3.accfreq * (c3.tt + EvalRule.et) \\ &+ c4.accfreq * (c4.tt + Convolve.et) \\ &+ c5.accfreq * (c5.tt + Centroid.et) \\ in1val.et &= in1val.ict + 0 \\ in2val.et &= in2val.ict + 0 \\ EvalRule.et &= EvalRule.ict \\ &+ c8.accfreq * (c8.tt + trunc.et) \\ &+ \ldots \end{aligned} \qquad (3)$$

Pre-estimation – A behavior's internal computation time can be computed during pre-estimation through profiling and scheduling [15]. Profiling determines the execution count of each basic block. A schedule for each basic block is then estimated for each possible processor component, using compilation for standard processors and synthesis for custom processors. 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. Channel access frequencies are also determined through profiling. Bus times and widths are already associated with each bus.

Online estimation – Given a partition of every functional object to a component, the actual *ict*, bus widths, 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 may now be on different components. We only 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 updated.

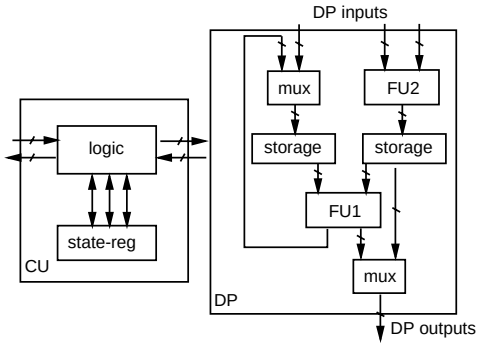

Fig. 8: CU/DP area model

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

Fig. 9: Equation and terms for computing CU/DP area

4.3.2 Hardware size

SpecSyn uses a hardware design model similar to those in [11, 16, 17], consisting of a control-unit/datapath (CU/DP) as shown in Figure 8. We present our estimation technique [14] briefly here. 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 9, each term is a func-

tion of basic parameters, such as the number of possible states and control lines.

Pre-estimation – The parameters relevant to each functional object are computed (through approximate synthesis) and annotated to each object. 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 FU's by taking the union of the objects' FU's (since sequential behaviors can share FU's).

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; similar analysis can be performed for muxes.

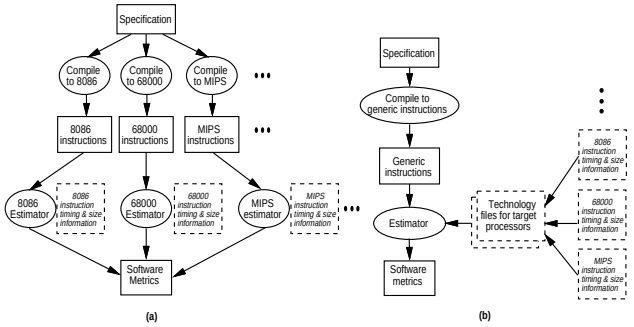

Fig. 10: Software size estimation: (a) processor-specific model, (b) generic model.

4.3.3 Software size

Software size refers to the number of bytes of program memory required when a behavior is compiled into the instruction set of a specific processor.

Pre-estimation – While compilation can be used to obtain program size for a behavior on a given processor, SpecSyn uses a generic processor model as shown in Figure 10(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. Details on derivation of technology files for specific processors are given in [15]. Some experiments comparing the generic model with the processor-specific model yielded inaccuracy of roughly 7% [15].

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.

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.

5 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. Tasks include generations of interfaces [11], memories, and arbiters. The refined output can serve as input to simulation and cosimulation tools.

```
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';
      ....
```

```
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(mr1_val1, mr1_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;
```

Fig. 11: Refined fuzzy-logic controller VHDL description.

In Figure 11, we show part of a refined specification for the system design shown in Figure 6. The interface of the fuzzy controller remains unchanged. However, its contents now consist of more details than in the original specification of Figure 2, such as new subroutines that read data from *Memory1* and write data to *Memory2* using detailed communication protocols for memory accesses. Note the large amount of detail that must be added to the specification. Also, note that the designer has access to that detail, since a refined specification was generated, which can be viewed and edited. Due to space limitations, we refer the reader to [11, 18] for details on refinement.

6 Experiments

SpecSyn, under development since 1989, consists of over 150,000 lines of C code. Its main interface is 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 various design tasks, with results reflected in the displayed values, and violated constraints highlighted. SpecSyn has been released to several universities and companies.

We conducted experiments exploring design alternatives for several industrial examples. We present results for a fuzzy-logic controller example [19]. Four library components were available: a standard processor (Intel 8051) and three custom processors with 50k, 100k, and 150k gates, each with a cost. We automatically generated all possible allocations of these components below a certain cost. For each allocation, we partitioned using simulated annealing and a cost function that sought to meet size and pin constraints and minimize execution time.

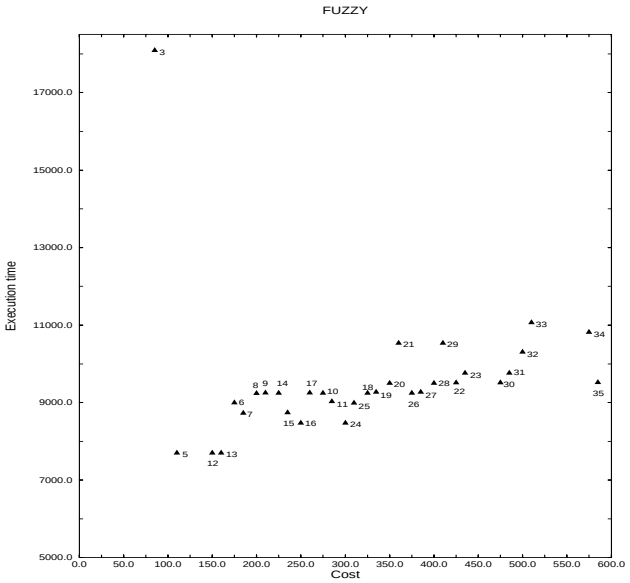

Fig. 12: Exploration for the fuzzy-logic controller

Figure 12 shows results for 35 allocations. Constraint-violating allocations are omitted. Allocation 3 was one 8051 standard and one 50k custom processor, resulting in a fuzzy-controller execution time of 18,115 microseconds. Allocation 5 was one 8051 and one 100k processor, yielding 7721 microseconds. Higher-cost allocations yielded no better execution time. SpecSyn thus aids the designer to understand the design space, enabling a focus on promising points. The above data was generated in 1 hour on a Sparc 2.

SpecSyn was used by an industry engineer to design the fuzzy-logic controller [19]. The partitioning results matched favorably with those obtained by another engineer who did a manual partition. 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 from the 6-months required for the manual design.

SpecSyn was used in the design of a recent commercial product: a 1 ms scan, 10k step programmable controller [20]. Additional SpecSyn information can be found in [21].

7 Conclusions

We described the SpecSyn environment supporting the specify-explore-refine system-design paradigm. Our exploration approach uses pre-estimation and online-estimation to achieve fast and accurate estimates, supports various partitioning heuristics, and is intended to be continually extended, enabling a designer to quickly examine many alternative designs. This paradigm and tool may eventually result in a 100-hour design cycle, and our experiments demonstrate the feasibility of such a design-time reduction. Future work may involve estimation models for pipelining and caching, new transformations, support of fixed cores, and incorporation of post-synthesis metrics into design iterations.

References

[1] A. Balboni, W. Fornaciari, and D. Sciuto, "Partitioning and exploration strategies in the tosca co-design flow," in *Int. Workshop on Hardware-Software Co-Design*, pp. 62–69, 1996.

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

[3] J. Henkel and R. Ernst, "A hardware/software partitioner using a dynamically determined granularity," in *DAC*, 1997.

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

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

[6] J. Hou and W. Wolf, "Process partitioning for distributed systems," in *Int. Workshop on Hardware-Software Co-Design*, pp. 70–75, 1996.

[7] S.Prakash and A. Parker, "Synthesis of application-specific multiprocessor architectures," in *DAC*, pp. 8–13, 1991.

[8] W. Wolf, "Hardware-software co-design of embedded systems," *Proceedings of the IEEE*, vol. 82, no. 7, pp. 967–989, 1994.

[9] O. Bentz, J. Rabaey, and D. Lidsky, "A dynamic design estimation and exploration environment," in *DAC*, 1997.

[10] J. Rowson and A. Sangiovanni-Vincentelli, "Interface-based design," in *DAC*, 1997.

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

[12] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming 8*, pp. 231–274, 1987.

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

[14] F. Vahid and D. Gajski, "Incremental hardware estimation during hardware/software functional partitioning," *IEEE Transactions on VLSI Systems*, vol. 3, no. 3, pp. 459–464, 1995.

[15] 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.

[16] M. McFarland and T. Kowalski, "Incorporating bottom-up design into hardware synthesis," *IEEE Transactions on CAD*, pp. 938–950, September 1990.

[17] E. Lagnese and D. Thomas, "Architectural partitioning for system level synthesis of integrated circuits," *IEEE Transactions on CAD*, vol. 10, pp. 847–860, July 1991.

[18] J. Gong, D. Gajski, and S. Bakshi, "Model refinement for hardware-software codesign," in *Proceedings of the European Design and Test Conference (EDTC)*, 1996.

[19] L. Ramachandran, D. Gajski, S. Narayan, F. Vahid, and P. Fung, "Towards achieving a 100-hour design cycle: A test case," in *EuroDAC*, pp. 144–149, 1994.

[20] L. Matsushita Electric Works, "Fp10sh programmable controller." Datasheet, 1996.

[21] D. Gajski, F. Vahid, S. Narayan, and J. Gong, "Specsyn: An environment supporting the specify-explore-refine paradigm for hardware/software system design," *IEEE Transactions on VLSI Systems*, vol. 6, no. 1, pp. 84–100, 1998.