Automatic Model Refinement for Fast Architecture Exploration

Junyu Peng, Samar Abdi, and Daniel Gajski
Center for Embedded Computer Systems
University of California, Irvine, CA 92697
{pengj,sabdi,gajski}@ics.uci.edu

Abstract

We present a methodology and algorithms for automatic refinement from a given design specification to an architecture model based on decisions in architecture exploration. An architecture model is derived from the specification through a series of well defined steps in our design methodology. Traditional architecture exploration relies on manual refinement which is painfully time consuming and error prone. The automation of the refinement process provides a useful tool to the system designer to quickly evaluate several architectures in the design space and make the optimal choice. Experiments with the tool on a system design example show the robustness and usefulness of the refinement algorithm.

1 Introduction

In the recent years, the dramatic increase of behavioral and structural complexity of SoC designs has raised the abstraction level of system specification. Along with the higher levels of abstraction comes the need for efficient system level synthesis of functional specification to target architectures. The wide variety of available target architectures makes the job of making the optimal choice all the more complicated. This calls for a methodology to efficiently explore design spaces and fast tools for refinement of functional system specification to an architecture model, so that more architectures may be explored and evaluated. Our SpecC [1] system-level design methodology is aiming at refining an initial, functional system specification into a detailed implementation description ready for manufacturing.

SpecC methodology consists of a set of models and transformations (Figure 1). The executable models represent the same system at different levels of abstraction at different phases of the design process. The transformations are a series of well-defined steps through which higher level models are gradually refined into lower level models. Our methodology starts with the capture of the intended functionality in the form of *specification model* which describes the functionality as well as the performance, power, cost and other constraints of the intended design. *Architecture exploration*, which synthesizes the specification into an *architecture model*, includes the design tasks of allocation, partitioning of behaviors, channels, and variables, and scheduling. *Communication synthesis* synthesizes the abstract communications between behaviors in the architecture model into an implementation. In the resulting *communication model*, communication is described in terms of actual wires and timing is described with bus protocols.

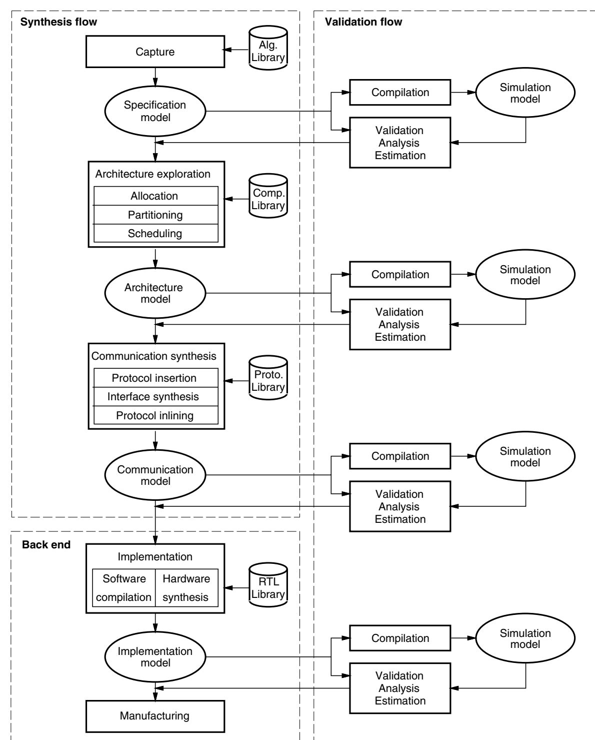

Figure 1. SpecC Methodology.

As shown in Figure 2, architecture exploration generally

Proceedings of the 15th International Conference on VLSI Design (VLSID'02)
0-7695-1441-3/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

is an interactive process to find a target architecture that satisfies all design constraints. For each iteration, first, a target architecture is selected and specification is mapped to the architecture either manually by designers or automatically by tools. Then, the specification model is refined into an architecture model to reflect the design decisions. Finally, the derived architecture model is simulated to validate the refinement step and to estimate design metrics for evaluation. To speed up architecture exploration, all three steps need to be automated as much as possible. Many algorithms and tools were proposed to automate architecture selection (HW/SW partitioning, scheduling, etc.). A few tools are also available for simulation and estimation. However, nowadays, model refinement are performed manually by designers. In practice, it is a time-consuming and error-prone task. The iterative nature of architecture exploration requires multiple cycles of the refinement step which puts more burden on designers. In our methodology models and associated refinement rules are formally defined. It, therefore, becomes practical to develop algorithms and tools to automate the model refinement process. The tool can relieve the designer from doing time-consuming work and shorten design cycle, thereby increase productivity.

Figure 2. Architecture Exploration Flow

In this paper we will focus on automatic refinement from specification model into architecture model. We will identify a set of refinement tasks and describe a tool that implements these tasks. The input to our refinement tool is the executable specification of an abstract functional model, the target architecture and the partitioning of behaviors and their allocation to the components in the target architecture. The tool produces a simulatable architecture model that may be used to obtain performance metrics. These performance metrics are important for evaluating the designer's choice of target architecture and design decisions.

The rest of the paper is organized as follows. Section 2 talks about previous work in architecture exploration. Section 3 will focus on the design environment in more detail. We will illustrate refinement rules with an example in section 4. The implementation of the automation tool and experiment result is shown in section 5. Section 6 summarizes the paper with conclusions.

2 Related Work

With the growth in complexity of SoC designs, the abstraction levels in the design flow have risen above the traditional RTL. There has been much work in defining standard abstraction levels for system level design [5]. Issues in modeling are addressed in [2], [3], [4] which looks at various models of computation and system prototyping.

However, with the move towards defining standards in semantics for different levels of abstraction in system level design [5], we see an increasing need and possibility for refinement automation tools in this domain. But no much work has been focussed on this topic.

3 Architecture Refinement Environment

3.1 Specification model and architecture model

Specification model is written by the user to specify the desired system functionality. The specification model is a purely functional model, free of any implementation details. Design constraints can be specified in the specification model in forms of annotation. A simple example specification model is shown in Figure 3.

Figure 3. An Example Specification Model

Architecture model accurately describes the selected system architecture. Each allocated system component (DSP, ASIC, Memory, ...) is represented by a corresponding top-level behavior. Inter-component communication is realized via abstract channels which will be mapped to bus

Proceedings of the 15th International Conference on VLSI Design (VLSID'02)
0-7695-1441-3/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

protocols later in communication synthesis. Architecture model can be simulated to validate the correctness of architecture refinement. More importantly, through simulation we can get estimation results in terms of performance. For the software part, the code can be compiled to the target processor assembly and an instruction set simulator can be used for estimation. For the hardware part, behavioral synthesis can be used to estimate hardware cost.

3.2 Architecture allocation and mapping

Architecture allocation and mapping produce design decisions, which are the input for model refinement. These decisions include: types and numbers of components, behavior mapping information, i.e. which behavior is mapped to which component, variable mapping information, i.e. where to store global variables, and execution schedule of behaviors on each component. The decisions can be either made by designers manually or generated with other architecture exploration tools.

4 Model Refinement

The model refinement process can be divided into three relatively independent steps, namely, *behavior refinement*, *variable refinement* and *scheduling refinement*, which can be further divided into sub-steps.

4.1 Behavior refinement

The behavior refinement step modifies the model to reflect the system component allocation and behavior partitioning decisions. This is by far the most important and time consuming part of architecture refinement.

4.1.1 Insert synchronization

The model refinement reflecting behavior partitioning is not as simple as merely grouping together behaviors mapped to same components. In architecture model components run concurrently (like in a multi-processor system), therefore explicit synchronization needs to be added to preserve the execution semantics of the specification model. To add this synchronization, we need a transition graph which defines the data dependency across the behaviors. However, we are given a hierarchy tree of the design and we need to derive a transition graph from it.

Note that after the partition, the leaf behaviors are allocated to specific components. In this case the black leaf nodes represent behavior instances mapped to hardware and the white leaf behavior instances are mapped to processor specific software. First we need to preprocess this graph by pruning it. All nodes with children of the same color (ie. the same component allocation) are assigned the color of the children and their children are removed from the graph. In this particular example we can see the effect of pruning on node X in Figure 4. Note that both the children of X (ie. D and E) were assigned to software, hence they are removed and X is colored white.

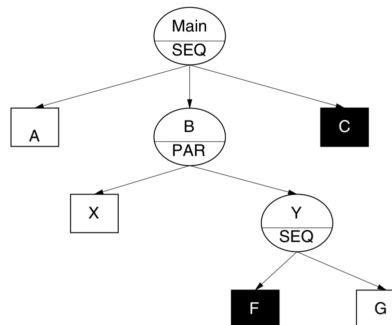

Figure 4. The Pruned Hierarchy Tree

The Algorithm to derive the transition graph is now run on this pruned hierarchy tree. Note that each non-leaf node in the hierarchy tree has an annotation SEQ or PAR. This means that its children are composed either sequentially or parallely, respectively. It is assumed that any sequentiality would mean data dependency. Deriving the transition graph would make use of this property of the hierarchy tree. The pseudo code for the algorithm is shown in Figure 5.

For better understanding, we will walk through our example to show the generation of the transition graph. The algorithm is essentially a DFS on the pruned hierarchy tree. We start with the root node Main. As we can see, its children (A, B and C) are sequentially composed, therefore we add directed edges (A,B) and (B,C) as shown in Figure 6(a). Next, we look at node A. Since it is a leaf node, we have nothing more to do. Node B is observed next. It has two children X and Y composed in parallel. This means that we can enter behavior B with either X or Y. Hence, both X and Y have possible dependencies on all predecessors of B. In this case the only predecessor of B is A, hence we add edges (A,X) and (A,Y). Similarly, B can exit with either X or Y so both can be predecessors of all successors of B, in this case C. Hence we add edges (X,C) and (Y,C). The intermediate transition graph is shown in Figure 6(b). X is a leaf node, so we explore Y. Y is a sequential composition of F and G, hence the edge (F,G) needs to be added. Also, schedule order dependencies would result in edges (A,F), (A,G), (F,C) and (G,C) to be added to the transition graph. Since, there are no more nodes to explore, we are done and the generated transition graph is shown in Figure 6(c).

Once we have the transition graph, we need to add synchronization between components at appropriate points. Figure 7 shows a partition decision, where behaviors C and

Proceedings of the 15th International Conference on VLSI Design (VLSID'02)
0-7695-1441-3/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

```
Derive_Transition_Graph (G_H, G_T)
begin
   G_T = 0;
   Visit_Node (G_H.Root, G_H, G_T);
end

Visit_Node (node, G_H, G_T)
begin
   if (node.Composition == LEAF) return;
   child = node.firstChild;
   while (child != NULL)
      forall (x, node) ∈ G_T G_T = G_T ∪ (x, child);
      forall (node, y) ∈ G_T G_T = G_T ∪ (child, y);
      if (node.Composition == SEQ &&
         child.next != NULL)
         G_T = G_T ∪ (child, child.next);
      Visit_Node (child, G_H, G_T);
      child = child.next;
   forall (x, node) ∈ G_T G_T = G_T - (x, child);
   forall (node, y) ∈ G_T G_T = G_T - (child, y);
end
```

Figure 5. Pseudo Code for Generating Transition Graph

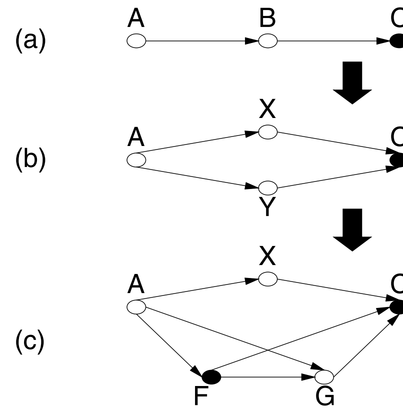

Figure 6. Construction of Transition Graph

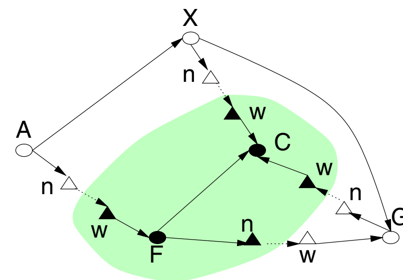

Figure 7. Synchronization across Partition

F are mapped to hardware component and the rest to software. The behaviors mapped to the same component run in the preassigned schedule in a single thread of execution. Therefore, they do not need any global synchronization amongst them. However, control/data dependencies across components must be resolved by adding appropriate synchronization. Our model of synchronization uses events. The dependent behavior must wait for an event notification to proceed. Hence we see the NOTIFY and WAIT modules added along cross component dependencies in our transition graph (Figure 7). These modules get translated to synchronization behaviors within the component.

4.1.2 Group behaviors and hoist commnication

In this step, component behaviors representing allocated components are constructed. Behaviors in the hierarchy will be grouped under these component behaviors by looking at the behavior partitioning information. The structural and behavioral hierarchy of the specification should be preserved in the parts mapped to each component after the grouping. We follow a simple method to create the partition. All components initially make a copy of the original hierarchy. We then travel through each component's hierarchy and remove behavior instances that are not mapped to it. We then create a top level behavior that instantiates all the global components and composes them in parallel.

The behavior usually has local variables and channels for communication among its sub-behaviors. If its sub-behaviors are mapped onto different components, the local communication variables and channels must be moved to the top level of the hierarchy as top-level variables and channels to be visible to all components that need to access them. For this purpose, we keep a list of behaviors for each variable. Behaviors in this list either read or write the variable. If any two of the behaviors is mapped to different component, the variable is hoisted as a global variable. In case of name conflicts amongst the hoisted global variables, the name is changed and all affected references are modified appropriately.

4.2 Variable refinement

In the architecture model, the global variables used in the specification model will be bound to physical storage. These variables can be mapped either to local memory of each component or to a dedicated shared memory component. Variable refinement changes the model to reflect variable mapping decisions. To automate this step, data dependency should be analyzed.

If variable is mapped to a shared memory component,

Proceedings of the 15th International Conference on VLSI Design (VLSID'02)
0-7695-1441-3/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

declare such variable in the memory behavior and remove it from the top level of the design. A top-level channel is introduced to connect the memory component with all other components that access the variable and accesses to it are replaced with READ and WRITE methods implemented by associated channel.

Otherwise, variable is bound to local memories of components. A message passing channel is introduced to exchange updated values of the variable among components. For each global variable, a local copy is declared inside each component that needs access to the variable and we replace access to global variables with access to local copy and insert SEND and RECV methods implemented by associated message-passing channel at synchronization points to exchange updated values. For a read operation, we traverse the thread of execution backwards and insert an update of the variable after the first encountered synchronization point. Similarly, for a write operation, we traverse the thread of execution forward and insert an update of the variable before the first encountered synchronization point. Note this is needed to ensure coherence amongst loal copies.

Figure 8 shows how a specification is transformed after partitioning and variable refinement. As shown in the example, 'x' is a global variable accessed by behaviors A and B. The global variable is removed and local copies are made as A and B are mapped to different components. A message passing channel 'Cx' is introduced that is used for maintaining validity of local copies of 'x'.

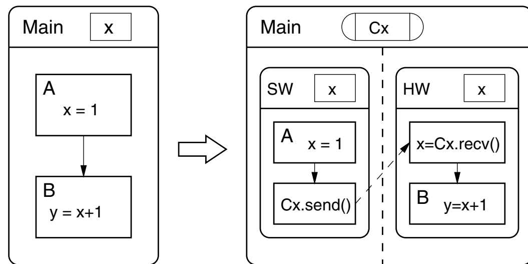

Figure 8. Variable Refinement

4.3 Scheduling refinement

On each single-processor component, the real execution of behavior instances is purely sequential. As mentioned earlier, after architecture allocation and mapping, the schedule on each component is determined. Based on the given schedule, scheduling refinement then transforms the model by replacing all concurrent (parallel, pipeline) constructs with sequential constructs. It is possible that after the scheduling of behaviors some of the earlier added synchronization behaviors become unnecessary therefore they must be identified and removed from the model for optimization purposes.

The final architecture model after scheduling refinement is shown in Figure 9.

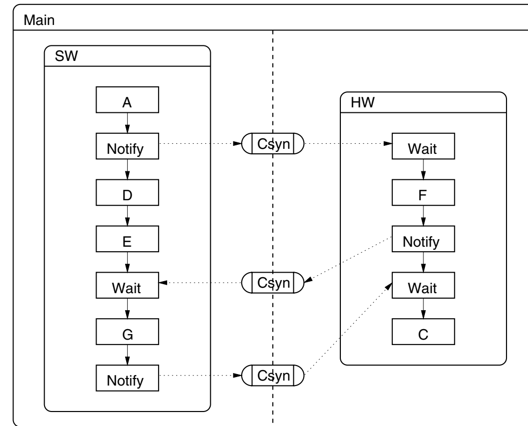

Figure 9. The Final Architecture Model

5 Experimental Results

Based on the refinement rules defined in previous section, we implemented a tool in C++, which can automatically refine a specification model into corresponding architecture model. We chose the SpecC design language for our modeling purposes.

The system design example is a JPEG encoder whose specification is illustrated in Figure 10. The top level encoder module is divided into four blocks loosely based on functionality. The first block, the *HandleData Block*, reads the inputs H,W and pixel stream through the input ports, then groups the pixel stream into 8X8 pixel matrices (called MCUs) for later processing. The *DCT block* reads each MCU passed from *HandleData Block*, preshits the MCU and performs DCT on it, producing a transformed 8X8 matrix. The *Quantization Block*, uses a quantization table to quantize each element of the MCU from the DCT block. The last block, *HuffmanEncode Block* performs Huffman Entropy encoding and run-length encoding (RLE) on successive bytes from the MCU. All these four blocks run in a pipelined fashion so we can think of them as a parallel composition of Behaviors.

Once we have the Specification model, we now need to partition it so as to map the individual blocks onto available components. For this example we chose the Motorola DSP56600 [6] as target for software implementation and an ASIC for hardware implementation. The partitioning decision is essentially mapping each block in the Specification Model to either the processor or ASIC. An extensive exploration of all kinds of partitionings is affordable with our

Proceedings of the 15th International Conference on VLSI Design (VLSID'02)
0-7695-1441-3/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

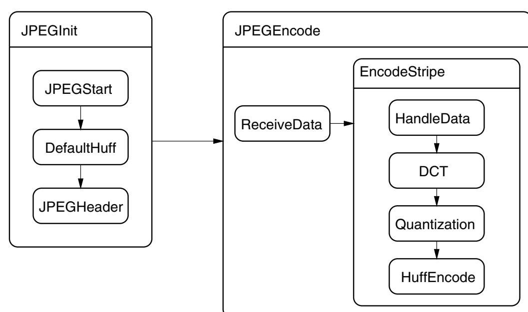

Figure 10. JPEG Specification Model

automatic model refinement. Two possible partitions are illustrated in Figure 11 and Figure 12.

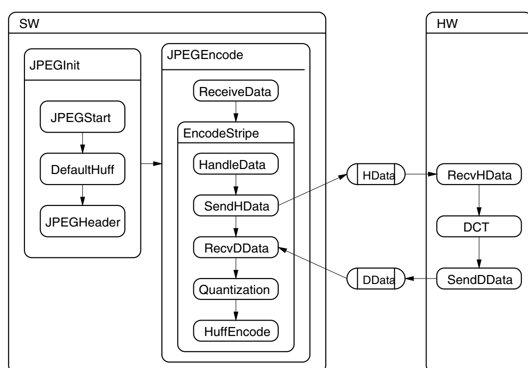

Figure 11. Candidate Architecture Model (1)

For the first phase of the experiment, we manually rewrote the two possible architecture models for respective partitions. In the second phase we input partitioning information to the tool and let it produce the architecture model automatically. Figure 13 compares the time used by each phase. As we can see, the automatic refinement reduces time from hours into minutes for each candidate architecture model. Typically, the exploration process has several iterations and hence the overall absolute gain can be considerably high.

6 Conclusion and future work

In this paper, We presented the refinement rules and algorithms for transforming a specification model into an architecture model in our design methodology. In the design flow, our contribution is primarily the automation of the architecture refinement process that facilitates rapid prototyping and evaluation of several design points. We developed a tool to automate the refinement based on the rules. Experiments were performed to show the feasibility and robustness of the refinement automation. The observed dramatic

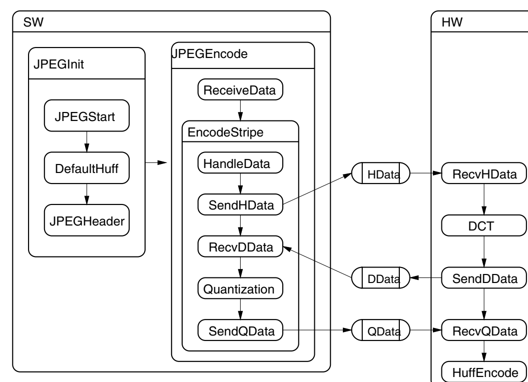

Figure 12. Candidate Architecture Model (2)

Task		Manual	Automatic
Architecture Model 1	Input	0	~5 minutes
	Refine	~5 hours	~1 minutes
	Total	**~5 hours**	**~6 minutes**
Architecture Model 2	Input	0	~5 minutes
	Refine	~3 hours	~1 minutes
	Total	**~3 hours**	**~6 minutes**

Figure 13. Time for JPEG Encoder Design

increase of productivity will relieve designers from tedious and error-prone task of rewriting models. For the future, we aim at refining communication between components and provide a suite of tools for going from a specification to an RTL implementation with our design methodology.

7 References

[1] D. D. Gajski et al. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[2] Coware Inc. N2C. *available at http://www.coware.com /cowareN2C.html*

[3] Synopsys Inc. SystemC, Version 2.0 *available at http://www.systemc.org*

[4] J. Buck, S. Ha, E. Lee, D. Messerschmitt. ”Ptolemy: a framework for simulating and prototyping heterogeneous systems”, *Int. Journal of Computer Simulation, vol. 4, pp.155-182, April 1994*

[5] A. Gerstlauer, *SpecC Modeling Guidelines*, University of California, Irvine, Technical Report ICS-TR-00-xx, September, 1998.

[6] Motorola Inc. DSP56600 family of DSPs. *available at http: //e-www.motorola.com*