A Comprehensive High-level Synthesis System for Control-Flow Intensive Behaviors*

W. Wang†, T. K. Tan†, J. Luo†, Y. Fei†, L. Shang†, K. S. Vallerio†, L. Zhong†, A. Raghunathan‡ and N. K. Jha†

† Dept. of Electrical Eng., Princeton University, NJ 08544

‡ NEC, C&C Research Labs, Princeton, NJ 08540

ABSTRACT

In this paper, we describe a comprehensive high-level synthesis system for control-flow intensive as well as data-dominated behaviors. We propose a new control-data flow graph model to preserve the parallelism inherent in the application, as well as to facilitate high-level synthesis. Our algorithm, which is based on an iterative improvement strategy, performs clock selection, scheduling, module selection, resource allocation and assignment simultaneously to fully derive the benefits of design space exploration at the behavior level. The system can be used to optimize area, power or energy, by selecting the cost function accordingly. Experimental results show that for *energy-optimized* designs, energy is reduced by up to 79.4% (an average of 42.2%), with an average of 24.8% area overhead, compared to *area-optimized* designs. For *power-optimized* designs, power is reduced by up to 70.8% (an average of 56.7%), with an average of 25.2% area overhead, compared to *area-optimized* designs. No V_{dd} scaling is performed to obtain the above results.

Categories & Subject Descriptors

B.5.1: [**Design**]: Control design, Data-path design

General Terms

Design, Performance

Keywords

High-level Synthesis, Low Power Design, Control-flow Intensive Behaviors

1. INTRODUCTION

High-level synthesis converts a behavioral description into an optimized register-transfer level (RTL) description. Typical tasks in high-level synthesis include scheduling, resource allocation, module binding, module selection, register binding and clock selection. The fact that these tasks interact with each other makes it necessary that their effects be considered simultaneously, in order to fully explore the benefits of the design space at the behavior level.

1.1 Related Work

Most previous work on high-level synthesis tackles data-dominated behaviors [1, 2], normally found in digital signal processing and image processing applications. These behaviors are characterized by a predominance of arithmetic operations and an absence of control flow. Control-flow intensive behaviors, which may have a large number of nested loops and conditionals, are frequently encountered in network-centric systems. Therefore, a high-level synthesis system that can handle control-flow intensive behaviors is also required. In [3] and [4], timing analysis and loop-directed scheduling are presented to drive high-level synthesis of control-flow intensive circuits. A number of efficient algorithms are presented in [5] for such behaviors. However, these techniques are limited to performance and area optimization, without consideration to power and energy. One work for reducing power consumption in control-flow intensive applications is given in [6], where a profile-driven high-level synthesis technique for low power is presented. This method, however, is limited to simple conditional constructs and cannot handle multiple nested loops and branches. In [7], a high-level synthesis algorithm for control-flow intensive behaviors is presented. However, it does not tackle high-level synthesis tasks such as clock selection and memory binding.

1.2 Paper Overview and Contributions

In this paper, we propose a comprehensive high-level synthesis system to perform the high-level synthesis tasks concurrently to better optimize energy, power, or area, by selecting the cost function appropriately. We take a control-data flow graph (CDFG) as an input to our high-level synthesis system. Loop optimizations, such as loop unrolling, may be performed on the CDFG. The scheduling algorithm, which supports concurrent loop optimization and multicycling under resource constraints, is able to generate a schedule that best preserves the parallelism in the input behavior. We obtain switched capacitance matrices [1], which are used later to efficiently estimate the power/energy consumption of the datapath by simulating the state transition graph (STG) representing the schedule. Starting with an initial architecture, the synthesis algorithm iteratively improves the architecture by performing various high-level synthesis tasks concurrently. The iterative improvement scheme is imple-

*This work was supported in part by Alternative System Concepts under an SBIR contract from Army CECOM and in part by DARPA under contract no. DAAB07-00-C-L516.

GLVLSI'03, April 28–29, 2003, Washington, DC, USA.

mented by generating and applying multiple sequences of moves to the initial architecture. The moves employed in our high-level synthesis system include module selection, module sharing and register sharing. The most efficient moves at any given point in synthesis are selected and implemented. After synthesis, `Verilog` descriptions of the most efficient datapath and controller are generated as the output of our system, which can be fed to lower level tools, such as a logic synthesis tool, to further optimize the design.

2. BEHAVIORAL REPRESENTATION

Two different models have been proposed to describe control-flow intensive behaviors: *control-flow graph* (CFG) and CDFG. While the CFG model is well suited for capturing execution of instructions on a general-purpose uniprocessor, it has been shown to be inadequate in exploiting the parallelism inherent in typical control-flow intensive applications. Existing CDFG models [7, 8] are successful in preserving the parallelism in the behavior. However, the implementation complexity of using such a CDFG is much higher compared to that of using a CFG. Based on these observations, we propose our own CDFG model, which is a variation of the CDFG model proposed in [7] (it introduces some *special nodes*). Two types of special nodes are employed in our system:

- **SLP, EIF**: An *SLP* node represents the start of a loop. It selects between the initial value and the value calculated from the previous iteration of the loop. An *EIF* node represents the end of an *if-else* branch. It is used to select the correct value from the two branches, *if* and *else*. Note that one common feature of these two nodes is that they serve as a "selector" for the variables. In the datapath, both *SLP* and *EIF* nodes are implemented by multiplexer(s).

- **BLP, ELP**: A *BLP* node sends the values of variables computed in the loop, which will be used in the next iteration, back to the *SLP* node. An *ELP* node represents the end of the loop. All the loop variables, which are used outside the loop, have to pass through an *ELP* node, before they are fed to operations outside the loop. This ensures correct operation execution sequence related to the loop. Note that *BLP* and *ELP* nodes do not perform any functionality, and thus, in the datapath, they are implemented as wires.

Our CDFG model also supports memory operations, *e.g.*, *load (ld)* and *store (st)*. A special control edge, called *SC*, is added between *load* (*store*) and the corresponding *store* (*load*), if necessary, to ensure correct memory access sequence.

With the above special nodes, high-level synthesis tasks can be performed in a much more efficient way. For example, the tool can easily recognize the start/end of a loop, as well as the range of the loop, with help from *SLP, BLP* and *ELP* nodes. Without them, to perform the same set of tasks, complicated techniques like *pattern matching* may need to be employed.

3. HIGH-LEVEL SYNTHESIS

In this section, we present the overall methodology and algorithmic details of our high-level synthesis system.

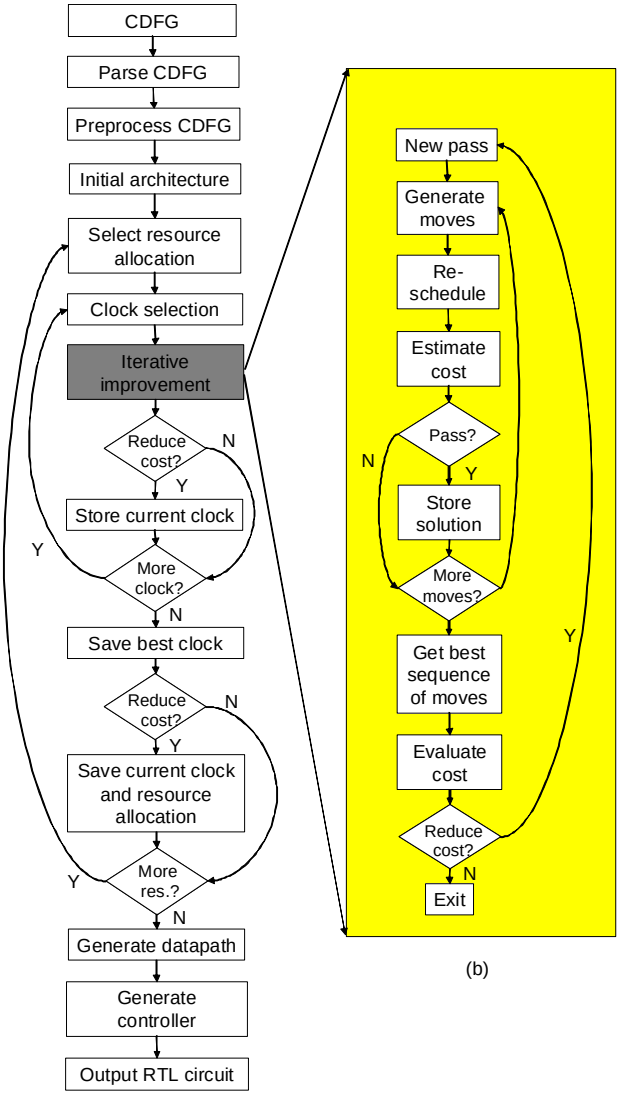

Figure 1: An overview of our high-level synthesis algorithm: (a) overall algorithm flow, and (b) iterative improvement algorithm

Figure 1(a) presents the overall flow of our high-level synthesis system. The first few steps are quite straightforward. The input behavioral description (CDFG) is parsed and read into the memory. The CDFG is preprocessed by unrolling the loops in the CDFG, whenever necessary. An initial architecture is then obtained, assuming infinite number of resources for the behavior. The following steps present the outline of our synthesis algorithm, which is described in more detail in Section 3.2. The outside loop is for resource allocation. For each resource allocation, multiple clock periods are selected and evaluated. Therefore, various promising combinations of resource allocation and clock periods are selected and used in the iterative improvement step. If the resource allocation is fixed in the form of a resource constraint, as may be the case for control-flow intensive behaviors, the algorithm just works within this constraint. The cost function is evaluated and the combination of resource allocation and clock period with the best cost is saved. Finally, the datapath as well as the controller are generated and the RTL circuit output in *Verilog*.

3.1 Scheduling

The schedule is represented as an STG. The scheduler takes a CDFG as an input and generates an STG as the scheduling result. An STG is a directed cyclic graph consisting of nodes and arcs representing states and transitions between states, respectively. Note that multicycling and resource constraints are supported in our scheduling algorithm. Figure 2 gives its pseudo-code. The inputs to the algorithm are a CDFG, G, and the resource constraint, RES. The scheduling result is stored in the STG. A queue, $State\_queue$, is employed to hold the states that need to be processed. An array, $indegrees$, is used to represent the number of input dependencies for each operation. For example, suppose operation, OP, has two input edges, $in1$ and $in2$, and no control edges. Initially, its indegree is 2. If the operation, whose output edge is $in1$, has been scheduled, implying availability of $in1$, OP's indegree is reduced to 1. If both $in1$ and $in2$ are available, its indegree is reduced to 0, indicating that OP is ready to be scheduled. In our scheduling algorithm, we first identify the *initial operations*, whose initial indegrees are 0. The *initial operations* are scheduled in the first state, $S0$, which is put into $State\_queue$, as well as the STG. Next, we dequeue $State\_queue$ into $state$. The potential branches out of $state$ are explored to find the corresponding successors. Under each branch, we mark the output edges of the operations, which finish their execution in $state$, as ready, and update $indegrees$. FIND_SCHED_OP() selects the schedulable operations, whose input edges are available and control conditions are satisfied under the branch, and puts them in $Oplist$. Note that in function FIND_SCHED_OP(), multicycling and the resource constraint are also taken into consideration. The schedulable operations are selected in a way that the resource constraint is not violated, and the continuity of multicycle operations is ensured. If $Oplist$ is not empty, *i.e.*, there are successors in this branch, a new_state and corresponding arcs are generated. This process is repeated until the $State\_queue$ is empty.

```
GEN_STG_FROM_CDFG (G, RES) {
  STG . INITIALIZE();
  State_queue . INITIALIZE ();
  indegrees . INITIALIZE ();
  Oplist ← FIND_SCHED_OP (G, indegrees, RES);
  S0 ← GENERATE_NEW_STATE (Oplist);
  State_queue . ENQUEUE (S0);
  STG . APPEND (S0);
  while (State_queue . NOT_EMPTY()) {
    state ← (State_queue . DEQUEUE());
    branches ← FIND_BRANCHES(state);
    FOR_EACH_branch(branches) {
      indegrees . UPDATE_INDEGREES (G, state, branch);
      Oplist ← FIND_SCHED_OP (G, indegrees, RES);
      if (Oplist . IS_EMPTY()) {
        continue; (*no successor for this branch*)
      }
      new_state ← FIND_STATE (Oplist);
      if (new_state) { (*existing state*)
      STG .GENERATE_NEW_ARC (state, new_state);
      } else { (*new_state is NULL, gen. a new state*)
        new_state ← GENERATE_NEW_STATE(Oplist);
        STG . GENERATE_NEW_ARC (state, new_state);
        State_queue . ENQUEUE (new_state);
      }
    }
  }
  return STG;
}
```

Figure 2: Pseudo-code of our scheduling algorithm

3.2 Synthesis Algorithm

In this section, we present more details of the synthesis algorithm. In our high-level synthesis system, the cost function can be either area, power, or energy. The algorithm performs resource allocation, clock selection, module binding, register binding, *etc.* Iterative improvement, as shown in Figure 1(b), is at the core of our synthesis algorithm.

We iteratively improve the initial RTL architecture cost by generating moves during the synthesis process. The iterative improvement algorithm is executed in multiple *passes* until there is no improvement in the cost. In each *pass*, a sequence of moves is generated. Three types of moves have been implemented in our synthesis system: module selection, module sharing, and register sharing. After each move, we reschedule the behavior, if necessary, and estimate the cost. If it improves the cost the most, the algorithm saves this move. In each pass, it explores the best sequence of moves and evaluates the cost by applying the moves. A new pass begins whenever the cost is reduced in the current pass. This process is repeated until no improvement in the cost can be achieved. Note that individual moves in the sequence may actually degrade the cost, allowing better moves to be applicable later as a result. This helps it escape local minima. Here, the cost can be area, power, or energy or their combinations. The complexity of the algorithm is related to the number of *passes*, the number of moves implemented, as well as the number of different clock steps. With a larger number of *passes*, moves and clock steps, we can achieve better results. However, the total execution time will increase accordingly. Our experiments show that, with five passes, four moves per pass, and ten clock steps, we can achieve close to optimal results with acceptable CPU time.

4. EXPERIMENTAL RESULTS

The methods described in this paper were implemented in the C++ programming language in a modular fashion so that it is easy to incorporate new techniques into our system. The input to our high-level synthesis system is a CDFG and an RTL design library. The output is an RTL circuit described in *Verilog*. Though our high-level synthesis system is targeted at control-flow intensive behaviors, it is also applicable to data-dominated behaviors. We applied our system to several behavioral benchmarks, consisting of both types of behavior.

The results obtained are summarized in Tables 1 and 2. In these tables, major column *Example* contains the name of the behavior. Major columns *Area-optimized*, *Power-optimized* and *Energy-optimized* represent, respectively, area-optimized, power-optimized, and energy-optimized designs. Columns *A, P, T, E* represent area, power, circuit execution time, and energy, respectively. Column *A.O.* represents the area overhead incurred by our technique. Columns *E.R.*

Table 1: Comparison of energy-optimized and area-optimized circuits

Example	*Area-optimized*				*Energy-optimized*					
	A (μm^2)	*P* (*mW*)	*T* (μs)	*E* (*nJ*)	*A* (μm^2)	*P* (*mW*)	*T* (μs)	*E* (*nJ*)	*A.O.* (%)	*E.R.* (%)
`Paulin`	60,014	0.550	513.470	282.11	95,134	0.370	484.129	179.13	58.5	36.6
`Conloop`	101,158	0.044	125.497	5.52	101,454	0.024	147.126	3.53	0.3	32.5
`GCD`	51,948	0.023	165.309	3.80	53,724	0.007	194.481	1.36	3.4	64.2
`Elliptic`	127,768	0.120	34.845	4.18	166,100	0.041	20.902	0.86	30.0	79.4
`Forloop`	43,308	0.175	2444.850	427.85	59,500	0.100	3200.530	320.05	37.4	25.2
`Nestloop`	65,388	0.048	3717.080	178.42	77,792	0.023	6595.860	151.70	19.0	15.0

Table 2: Comparison of power-optimized and area-optimized circuits

Example	*Area-optimized*				*Power-optimized*					
	A (μm^2)	*P* (*mW*)	*T* (μs)	*E* (*nJ*)	*A* (μm^2)	*P* (*mW*)	*T* (μs)	*E* (*nJ*)	*A.O.* (%)	*P.R.* (%)
`Paulin`	60,014	0.550	513.470	282.11	95,134	0.350	513.470	179.71	58.5	36.7
`Conloop`	101,158	0.044	125.497	5.52	101,454	0.022	164.619	3.62	0.3	50.8
`GCD`	51,948	0.023	165.309	3.80	53,724	0.007	196.777	1.38	3.4	69.6
`Elliptic`	127,768	0.120	34.845	4.18	166,100	0.035	28.112	0.98	30.0	70.8
`Forloop`	43,308	0.175	2444.850	427.85	60,604	0.070	4534.090	317.39	40.0	60.0
`Nestloop`	65,388	0.048	3717.080	178.42	77,792	0.023	6699.550	154.10	19.0	52.1

and *P.R.* represent the reduction in energy and power, respectively. Of the benchmarks, four are control-flow intensive behaviors. `Forloop` represents a behavior with a *for loop* in it. `Conloop` represents a behavior which has concurrent loops. `Nestloop` describes a behavior which has nested loops. Greatest Common Divisor (`GCD`) is a well known control-flow intensive behavioral benchmark. `Elliptic` is a fifth-order Elliptic wave filter. `Paulin` is another data-dominated behavior from the literature.

Table 1 presents comparisons between area-optimized and energy-optimized designs. Energy consumption is calculated as the product of power consumption and circuit execution time. The exact number of cycles to execute the behavior can be obtained from STG simulation. After selecting the clock period, we can obtain the circuit execution time by multiplying the number of cycles with the clock period. Compared to *area-optimized* designs, *energy-optimized* designs consume up to 79.4% (average of 42.2%) less energy, at the expense of 24.8% average area overhead. Table 2 presents comparisons for *area-optimized* designs and *power-optimized* designs. Compared to *area-optimized* designs, *power-optimized* designs consume up to 70.8% (average of 56.7%) less power, at the expense of 25.2% average area overhead. The experiments were performed on a 733 MHz Pentium-III PC with 512MB of memory, and took several seconds to several minutes of CPU time.

5. CONCLUSIONS

We presented an efficient iterative-improvement based high-level synthesis algorithm to perform clock selection, scheduling, module selection, module sharing and register sharing for control-flow intensive as well as data-dominated behavioral descriptions in order to minimize power, energy or area. Unlike most previous work, we also consider the interaction among these tasks in order to better explore the design space. We have implemented the algorithm, and presented experimental results to demonstrate its effectiveness. Up to 79.4% (70.8%) reduction in energy (power) can been achieved by employing our high-level synthesis system, without the aid of any supply voltage scaling.

6. REFERENCES

[1] A. Raghunathan and N. K. Jha, "SCALP: An iterative improvement based low-power data path synthesis algorithm," *IEEE Trans. Computer-Aided Design*, vol. 16, pp. 1260–1277, Nov. 1997.

[2] J. M. Chang and M. Pedram, "Register allocation and binding for low power," in *Proc. Design Automation Conf.*, pp. 29–35, June 1995.

[3] G. Lakshminarayana, K. S. Khouri, and N. K. Jha, "Wavesched: A novel scheduling technique for control-flow intensive behavioral descriptions," *IEEE Trans. Computer-Aided Design*, vol. 18, May 1999.

[4] S. Bhattacharya, S. Dey, and F. Brglez, "Performance analysis and optimization of schedules for conditional and loop-intensive specifications," in *Proc. Design Automation Conf.*, pp. 491–496, June 1994.

[5] S. Amellal and B. Kaminska, "Functional synthesis of digital systems with TASS," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 537–552, May 1994.

[6] N. Kumar, S. Katkoori, L. Rader, and R. Vemuri, "Profile-driven behavioral synthesis for low-power VLSI systems," *IEEE Design & Test Comput. Mag.*, pp. 70–84, Sept. 1995.

[7] K. S. Khouri, G. Lakshminarayana, and N. K. Jha, "High-level synthesis of low-power control-flow intensive circuits," *IEEE Trans. Computer-Aided Design*, vol. 18, pp. 1715–1729, Dec. 1999.

[8] R. A. Bergamaschi, A. Raje, I. Nair, and L. Trevillyan, "Control-flow versus data-flow scheduling: Combining both approaches in an adaptive scheduling system," *IEEE Trans. VLSI Systems*, vol. 8, pp. 82–100, Mar. 1997.