FACT: A Framework for the Application of Throughput and Power Optimizing Transformations to Control-flow Intensive Behavioral Descriptions *

Ganesh Lakshminarayana and Niraj K. Jha

Department of Electrical Engineering, Princeton, NJ 08544

Abstract

In this paper, we present an algorithm for the application of a general class of transformations to control-flow intensive behavioral descriptions. Our algorithm is based on the observation that incorporation of scheduling information can help guide the selection and application of candidate transformations, and significantly enhance the quality of the synthesized solution. The efficacy of the selected throughput and power optimizing transformations is enhanced by the ability of our algorithm to transcend basic blocks in the behavioral description. This ability is imparted to our algorithm by a general technique we have devised. Our system currently supports associativity, commutativity, distributivity, constant propagation, code motion, and loop unrolling. It is integrated with a scheduler which performs implicit loop unrolling and functional pipelining, and has the ability to parallelize the execution of independent iterative constructs whose bodies can share resources. Other transformations can easily be incorporated within the framework. We demonstrate the efficacy of our algorithm by applying it to several commonly available benchmarks. Upon synthesis, behaviors transformed by the application of our algorithm showed up to 6-fold improvement in throughput over an existing transformation algorithm, and up to 4.5-fold improvement in power over designs produced without the benefit of our algorithm.

1 Introduction

The explosive growth in the embedded systems market has fuelled interest in the automated synthesis of digital systems from functional specifications. These systems often need to be compact, and satisfy throughput and power constraints. Transformations which can improve these design metrics have recently gained prominence. Transformational techniques replace a behavioral description by a structurally dissimilar, but functionally equivalent, description. The new description usually results in designs which are better than those synthesized from the original description, with respect to specific design metrics.

Research in compiling techniques has identified a large number of program transformations which enhance performance and reduce size of programs on general-purpose computers. A comprehensive survey of these techniques is provided in [1]. These include algebraic transformations such as constant propagation, commutativity, associativity, distributivity, strength reduction, and common sub-expression elimination [2], loop-optimizing transformations such as loop un-switching, loop distribution, loop fusion, loop coalescing, loop peeling, and loop unrolling, software pipelining, speculative execution, code motion, redundant code elimination, and interprocedural optimizations such as inlining, cloning, *etc.* The idea of using program execution profiles to guide performance-optimizing transformations has been proposed in [3] and [4]. In most of these approaches, scheduling succeeds transformation application. We demonstrate that, in high-level synthesis, scheduling information can guide the application of transformations and enhance their effectiveness. In addition, we support power-optimizing transformations, unlike compiler-related optimization techniques for which speed is the primary objective.

In high-level synthesis, transformations for throughput and power optimization for data-flow intensive (DFI) behavioral descriptions [5], [6], as well as control-flow intensive (CFI) behaviors [7], [8] have been presented. None of these transformation techniques for CFI behaviors targets power optimization. With the exception of [8], these systems do not incorporate scheduling information into the transformation application process.

In this paper, we present a framework for transforming CFI behavioral descriptions to synthesize throughput- or power-optimized designs. At the heart of our approach is a procedure we have devised for recognizing candidates for transformation application across the scope of basic blocks, and applying the transformation while preserving functionality. This procedure drives the transformation process, which is closely integrated with scheduling. Our algorithm begins with a scheduled behavioral description, specified in the form of a state transition graph (STG). Profiling information is used to partition the STG into blocks, which are separately optimized by transformation application. This enables our algorithm to direct its focus on the critical sections of the behavior. Since the identified blocks can have arbitrary control flows within them, our algorithm has a complete picture of the trade-offs involved, unlike many other algorithms that only target simple structures such as straight-line code, or single-entry loops. Each block is transformed by the repeated application of data- and control-flow transformations, to optimize for the desired objective (throughput or power). Within a block, scheduling is interleaved with transformation application. The scheduling information, available at each step, allows effective identification of bottlenecks and the selection of appropriate transformations to eliminate them. We currently support the application of commutativity, constant propagation, associativity, distributivity, code motion, and loop unrolling. The scheduling algorithm we use performs implicit loop unrolling, and functional pipelining, and parallelizes the execution of independent loops. The framework we have developed can, however, easily be customized by the addition of user-specified transformations.

2 Preliminaries

In this section, we discuss control-data flow graphs (CDFGs), which describe the input behavior, and STGs, which describe the schedule. We then outline a high-level power estimation technique which obtains a fast estimate of the power consumption of a design, using an STG, user-specified input traces, and a set of library mod-

*This work was supported in part by NSF under Grant No. 9319269 and in part by Alternative System Concepts, Inc. under an SBIR contract from Air Force Rome Laboratories.

DAC 98, San Francisco, California

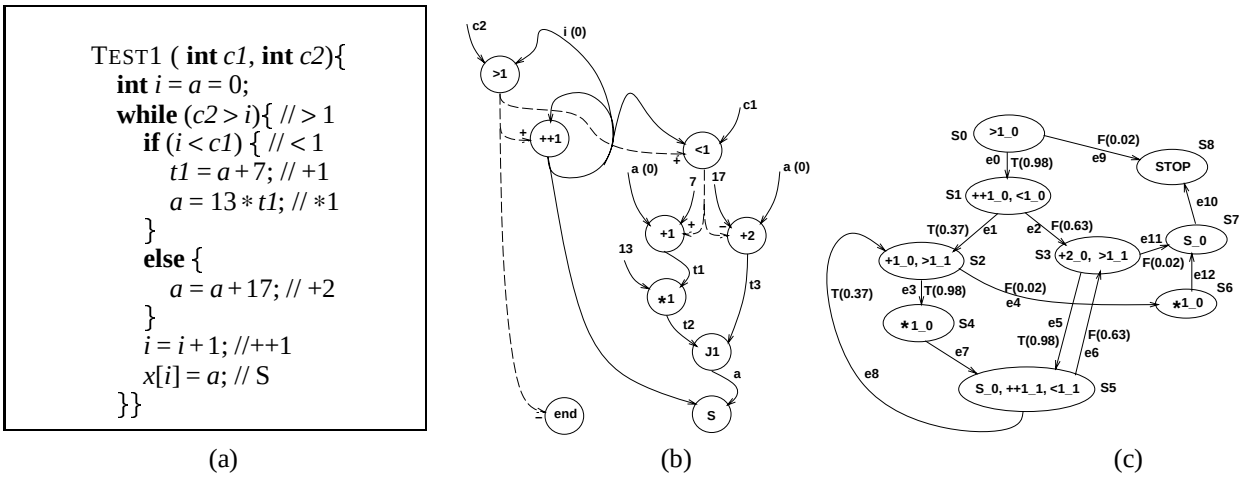


Figure 1: (a) A fragment of code called TEST1 written in a high-level language, (b) its CDFG representation, and (c) an STG representation of its schedule

ules characterized for delay and power.

2.1 CDFG and STG models

A CDFG is a directed graph, whose nodes represent operations, and edges represent dependencies of two types: data and control. An edge represents a data dependency if the source node of the edge produces data that the sink consumes. Existence of a control dependency between nodes implies that the execution of the sink node depends on the outcome of the execution of the source node. Our CDFG model uses a token passing semantic [9]. An operation consumes the tokens available at its inputs and generates a token at its output. The execution of an operation is contingent upon the arrival of tokens at its inputs.

Figures 1(a) and 1(b) show a behavioral description written in a high-level language and the CDFG corresponding to the description, respectively. Statements in the behavioral description are shown annotated with the operation(s) in the CDFG they represent. Data dependencies in the CDFG are indicated by continuous arcs, and control dependencies, by broken arcs. An operation, whose execution depends on a condition evaluating to *true* (*false*), is marked with a + (–). A *join* operation assigns to its output, the value at either of its input edges. Such an operation executes when a token is available at either input. Also included in our CDFG representation is a *select* operation with three inputs, *l*, *r*, and *s*, and an output *o*, which assigns to *o*, the value at its *l* (*r*) input ports, if the value at *s* is *true* (*false*).

On completion, the scheduler outputs the STG describing the schedule. Figure 1(c) represents the STG obtained by scheduling the CDFG shown in Figure 1(b). The STG is a directed graph whose nodes represent states and edges represent transitions between states. Nodes in the STG have information about the operations executed in the corresponding state, and edges capture the conditions under which a state transition takes place. Many schedulers support *loop unrolling*, a performance optimization technique, which allows operations from different iterations of a loop to be performed concurrently. In such cases, operations in a state are annotated with the iteration of the loop they represent. For example, the *zero*th iteration of operation $x[i] = a$ (**S_0**), and the first iteration of operations $i = i + 1$ (++1_1) and $i < c1$ (<1_1) are executed in state **S5**. The number in parentheses placed adjacent to an edge represents the probability that the corresponding edge is taken.

2.2 High-level power estimation

Application of transformations targeted towards reducing power consumption requires a high-level power consumption model whose predictions are correlated with the final power consumption of the design, and which is fast enough to be applied repeatedly in the inner loop of the transformation algorithm. In this section, we illustrate our power estimation technique with an example. Note that though this procedure is used to estimate the power consumption of designs during the transformation process, the final power measurements, used for our experimental results, are obtained from actual layouts.

Table 1: Functional unit selection, allocation, and component information

FU type	*Ops.*	#	$\frac{E}{V_{dd}^2}$	*Delay*	*Area*
comp1	>1, <1	2	1.1	12	1.3
cla1	+1, +2	2	1.3	10	1.5
incr1	++1	1	0.7	13	1.1
w_mult1	* 1	1	2.3	23	3.9
reg1		n/a	0.3	3	1.0
mem1	S_0	1	1.9	15	8.1

Problem statement: Given (a) a scheduled behavioral description in the form of an STG, (b) functional unit selection information (information about the type of functional unit which performs an operation), (c) a library of functional units and storage elements (registers, memories) characterized for area, delay, and power, (d) the clock period of the design, and (e) a set of typical input traces, estimate the power consumption of the final design.

We extend the method proposed in [5] to handle CFI designs. The average power consumption of the circuit is computed as the average energy consumption divided by the *average schedule length*. The average schedule length is defined as the average time, in cycles, to complete one execution of the behavior, and is computed from the STG and the input traces, using the method presented in [10]. The total energy consumed by the circuit is computed as the sum of the energies consumed by the functional units, registers, interconnect, and the controller. The energy, E, consumed by a functional unit or register is expressed as $E = C_{type} \times V_{dd}^2 \times N_{ops}$, where C_{type} is a constant which depends on the functional unit or register type, V_{dd} is the supply voltage, and N_{ops}, for a functional unit (register) is the number of operations executed (number of read/write accesses). Further details about the power model can be found in [5]. The following example illustrates our power estimation procedure.

Example 1: Consider the problem of estimating the power consumption of a design based on the STG shown in Figure 1(c). The library elements available, the energy dissipation, delay, and area associated with each element, functional unit selection, and allocation information (numbers and types of functional units allowed for synthesis) are summarized in Table 1. The clock period con-

straint is 25*ns.* Simulation of the behavioral description with the input traces yields the following branch probabilities: the while loop "**while** (*c2* > *i*)" closes with a probability of 0.98, and the branch "**if** (*i* < *c*1)" is taken with a probability of 0.37. Note that once the branch probabilities are determined, they can be *repeatedly used*. Therefore, simulation is done only once during an execution of the algorithm.

The state probabilities, evaluated using the method presented in [10], are as follows P_{S0} = 0.008, P_{S1} = 0.008, P_{S2} = 0.153, P_{S3} = 0.259, P_{S4} = 0.149, P_{S5} = 0.404, and P_{S6} = 0.003, P_{S7} = 0.008, and P_{S8} = 0.008, where P_{Si} is the probability of being in state *Si*. The average schedule length can be shown to be 119.11 cycles. Note that scheduling is performed assuming the supply voltage to be *5V*. If the supply voltage were different, functional unit delays would change, thus impacting the average schedule length.

Using the above probabilities, one can derive the average energy consumption per iteration of the design. To do this, we need to find the average number of operations executed by each type of functional unit, and the average number of variable accesses, per iteration of the design. These numbers are derived as the sums of per-state numbers of operations executed by different functional unit types, and variables accessed, weighted by the state probabilities. For example, the number of operations executed by functional units of type *incr1* is given by 119.11 × (P_{S1} × 1 + P_{S5} × 1) = 48.95. Since the energy consumed by the incrementer while performing one operation is 0.7 × V_{dd}^2 units (see Table 1), the average energy consumption, per iteration, by the incrementer is 34.27 × V_{dd}^2 units. The energy consumption due to the comparators, adders, multipliers, registers, and memory, and can be found to be 108.75 × V_{dd}^2, 63.64 × V_{dd}^2, 41.70 × V_{dd}^2, 99.38 × V_{dd}^2 and 93.10 × V_{dd}^2 units, respectively, using the same technique. The total energy consumption is $665.58V_{dd}^2$ units, after accounting for the contribution due to the interconnect and controller. We first compute the average power consumption by assuming the supply voltage to be *5V*. This is done by dividing the average energy consumption by the average time taken (119.11 cycles at *5V*), and evaluates to $\frac{665.58 \times V_{dd}^2}{119.11 \times cycle\_time}$ *units*.

The quadratic dependence of power consumption on supply voltage implies that significant reductions in power consumption can result if the supply voltage is scaled down. We estimate the scaled supply voltage in the following manner. As mentioned in Section 1, our algorithm starts with a scheduled behavior. Transformations and scheduling are then applied in an interleaved fashion. The initial schedule given to us is regarded as the "base" case, and all comparisons are made with respect to it. When we optimize for power, we scale the supply voltage until the average schedule length drops to that of the untransformed design. For example, if the STG shown in Figure 1(c) is derived from the transformed behavior, and the schedule derived from the initial (untransformed) behavior has an average schedule length of 151.30 cycles, the supply voltage is scaled according to the equation [1]

$$\frac{V_{dd}\_initial/(V_{dd}\_initial - V_t)^2}{V_{dd}\_new/(V_{dd}\_new - V_t)^2} = 119.11/151.30$$

If V_{dd}*_initial* is assumed to be 5*V* and the threshold voltage V_t is assumed to be 1*V*, then V_{dd}*_new* evaluates to 4.29*V*, which yields a power estimate of 665.58 × 4.29^2/(151.30 × *cycle_time*) = 80.96/*cycle_time* units for the design. Note that the term representing the average schedule length changes from 119.11 to 151.30 to reflect the change in the supply voltage. This is because an average schedule length of 119.11 *cycles* at 5*V* is equivalent to an average schedule length of 151.30 at 4.29*V*. Thus, our aim is to keep the performance of the untransformed and transformed scheduled behaviors the same while reducing power. ∎

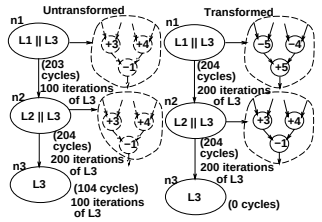
[1] $Delay = k \times V_{dd}/(V_{dd} - V_t)^2$ [11].

3 Motivational Examples

In this section, we motivate the key features of our algorithm with examples. Our algorithm is based on the following key observations:

- Considering scheduling information during transformation application can significantly enhance the quality of the chosen transformations. This is illustrated through Example 3.
- Applying transformations across basic block boundaries is critical in producing high-quality solutions, and maintaining correctness in the process can be complex and non-trivial. This is illustrated through Example 3.

```
TEST2 (int k){
L1   for i = 0 to 99{ //< 1
     t1 = y[i] + k; // +1
     k = k + t1; // +2 }
L2   for j = 0 to 99{ // < 2
     t2 = x[j] – k; // –2
     k = k – t2; // –3}
L3   for m = 0 to 399{ // < 3
     t3 = y1[m] + y2[m]; // +3
     t4 = y3[m] + y4[m]; // +4
     t5 = t3 – t4; // –1
     x1[m] = t5; }
     return;}
```



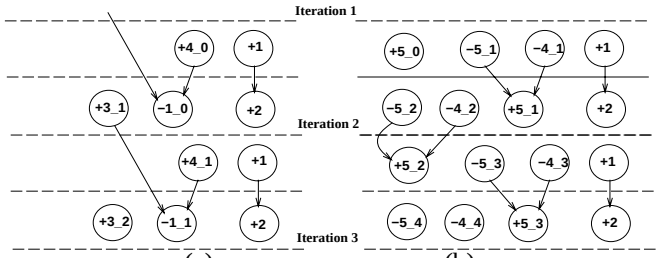
(a) (b) (c)

Figure 2: (a) A behavior, TEST2, (b) a schedule for TEST2, and (c) a schedule derived after transforming TEST2

Example 2: Consider the problem of synthesizing the behavior shown in Figure 2(a) to optimize for throughput. Suppose the resources available to implement this behavior are two adders, two subtracters, two comparators, and two incrementers, all of which execute in one cycle. Arrays *x*, *x*1, *y*, *y*1, *y*2, *y*3, and *y*4 are assumed to be mapped to separate memories, and can be simultaneously accessed. Figures 2(b) and 2(c) describe schedules for the untransformed and transformed behavior, respectively. Since the STGs for the actual schedules are rather involved, the given figures provide the basic features of the schedule, without giving a state-by-state description. Nodes in Figure 2(b) correspond to loops in the behavior. A node is annotated with its name (placed outside the ellipse representing it), and the loops it corresponds to (placed inside the ellipse). In this schedule *L1* and *L2* first execute in parallel, then *L2* and *L3*, and finally *L3* executes alone.

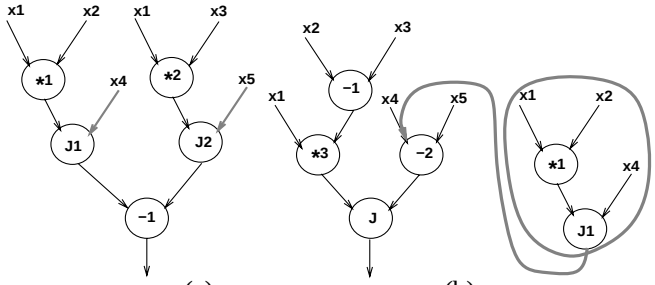
Nodes *n1* and *n2* are also annotated with the number of iterations of *L3* they represent. While *L3* is able to execute 200 iterations in node *n2*, it can only execute 100 iterations in node *n1*. An inspection of Figure 3(a) reveals the reason for this bottleneck. This figure shows concurrent execution of loops *L1* and *L3* over two iterations. Since contention for the adders and subtracters is the reason for the bottleneck, we only show the schedules for the addition and subtraction operations which compete for resources.

Since there are no inherent data dependencies that slow down the execution of *L3*, we can explore methods to transform its body in such a manner that its resource requirements are tailored to the environment in which it is embedded. One way to achieve this would be to rewrite (*y*1[*m*] + *y*2[*m*]) – (*y*3[*m*] + *y*4[*m*]) in the form (*y*1[*m*] – *y*3[*m*]) + (*y*2[*m*] – *y*4[*m*]), as shown in Figure 2(c) (see the CDFG enclosed in broken boundaries placed adjacent to node *n1*). In this figure, operation –5 represents *y*1[*m*] – *y*3[*m*], operation –4 represents *y*2[*m*] – *y*4[*m*], and operation +5 represents their addition. In this case, the body of loop *L3* would require two subtraction operations and one addition operation. Since *L1* consumes one adder per cycle, one adder and two subtracters are available. In this



(a) (b)

Figure 3: Transformations to improve resource utilization



(a) (b)

Figure 4: Transformations across basic blocks

scenario, one iteration of *L3* can begin every cycle, as illustrated in Figure 3(b). Note that this transformation is applied only to node *n1*, and node *n2* remains unaltered.

The schedule shown in Figure 2(c) has an expected length of 408 cycles, a 1.25-fold improvement over the schedule presented in Figure 2(b). This improvement is a direct consequence of considering scheduling information while applying transformations. Note that if this behavior were optimized for power, the improvement in throughput could be traded off for a 25% reduction in power consumption through V_{dd}-scaling. ∎

Example 3: Consider the problem of transforming the CDFG shown in Figure 4(a) to optimize for throughput under the following allocation constraints: one multiplier and two subtracters. All functional units are assumed to take one cycle to execute. The join operations embedded in the CDFG indicate that different threads of execution are possible, depending upon which of the *join* operation's inputs propagates to its output. In this case, since there are two distinct *join* operations, four threads of execution are possible. Out of the four possible threads, not all would occur in practice, because the occurrence of a token on one input might preclude the occurrence of a token on another, *i.e.*, some input pairs might be mutually exclusive. In this example, the pairs $\{x2,x5\}$ and $\{x3,x4\}$ are assumed to be mutually exclusive.

Transformational techniques typically identify specific structures in the CDFG which are good candidates for transformation application, and alter these structures in the manner dictated by the selected transformation. In this example, let us suppose we choose to utilize the distributivity of multiplication over subtraction to replace a structure, *Source*, of the form $a \times b - a \times c$ by a structure, *Target*, of the form $a \times (b - c)$. An inspection of the CDFG reveals that, if the *join* operations in the CDFG were configured to select their left inputs, *i.e.*, if *J1* selected the result of operation $*1$ and *J2* selected the result of operation $*2$, the data and control flows within the resultant CDFG would be identical to *Source*. This implies that under some conditions, the CDFG can be made to appear isomorphic to *Source*. Let C denote the condition under which this occurs. If *Source* were replaced by *Target* when C were to evaluate to *true*, then the resultant behavior would take only two cycles to execute as there is one subtraction, ($t1 = x2 - x3$), and one multiplication, $x1 \times t1$, as opposed to the three cycles it would otherwise need (two multiplications, occurring in sequence on one multiplier, with the results feeding a subtraction). If C were to occur with a high probability, the ability to perform this substitution would, clearly, be desirable.

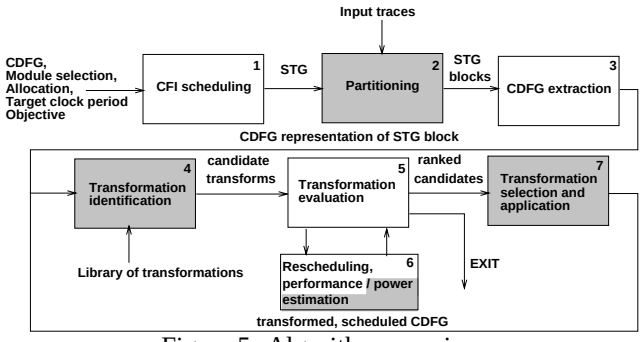


Figure 5: Algorithm overview

In performing this substitution, the following two issues need to be considered.

- The transformed CDFG should be functionally equivalent to the original CDFG for every thread of execution encountered.
- The transformed CDFG should be as compact as possible, *i.e.*, no redundant operations should be performed.

Figure 4(b) shows the CDFG obtained upon transforming the CDFG shown in Figure 4(a) (for now, the part of the CDFG shown in the grey enclosure, and the grey edge it sources can be neglected). It can be verified that, for every thread of execution encountered, the output of the transformed CDFG is functionally equivalent to the output of the original CDFG. Note that, if *x2* and *x5* were not mutually exclusive, the transformed behavior would not be functionally equivalent to the original one, as the original CDFG would compute $x1 \times x2 - x5$ upon the arrival of tokens at inputs *x1*, *x2*, and *x5*, and the transformed CDFG would not produce an output. To ensure correctness, in the case when *x2* and *x5* are not mutually exclusive, the left input to operation -2 should come from the grey edge. ∎

4 The Algorithm

In this section, we outline the working of our algorithm. Details can be found in [12]. Figure 5 illustrates the basics of our algorithm. In this figure, the steps shown shaded have been enhanced by our algorithm. Other steps correspond to well-researched problems, and are not described. The inputs consist of a behavioral description, specified as a CDFG, functional unit selection and allocation information, target clock period, typical input traces, and synthesis objective (performance or power). The behavior is first scheduled, using an existing CFI scheduler (step **1** in Figure 5). The output of the scheduling process is an STG. The input traces provided are used to analyze the STG to determine the probabilities with which individual states are encountered. The STG is then partitioned into blocks (step **2**). Individual blocks (groups of states) in the partition are then transformed and rescheduled in an iterative fashion using the following technique. First, the functionality of the block is captured in the form of a CDFG (step **3**), *i.e.*, we identify the portion of the CDFG which corresponds to the STG block. We then determine multiple ways in which the CDFG can be transformed, using a library of transformations (step **4**). The key strength of this process lies in its ability to recognize and apply transformations across the scope of basic blocks. The efficacy of each considered transformation is then assessed (steps **5** and **6**). This is done by applying the considered transformation, scheduling the CDFG, and using the schedule information to estimate the throughput or power (see Section 2.2). A subset of candidates is then chosen for further exploration (step **7**).

4.1 Partitioning

In this section, we describe the procedure followed to partition the STG prior to the application of transformations. The starting point is an STG and a set of user-specified input traces. The output of the process is a set of *disjoint* "STG blocks". An STG block is a collection of states that satisfies the following properties: (a) any state, S, can form a block, and (b) if a state, S', is an immediate predecessor or an immediate successor of any state in a block, B, then $B \cup S'$ is a valid block.

The first step in partitioning is the derivation of transition probabilities on the edges of the STG, *i.e.*, the probability that an edge, e, is taken, given that the current state is $Source(e)$[2]. This is done by simulating the CDFG representing the input behavior with the input traces provided. The simulation yields the number of times each branch in the CDFG is encountered, from which the probability of a branch can be computed. Once the probability of each CDFG branch is known, the transition probabilities on the edges of the STG can be computed as the product of the probabilities of the CDFG branches which constitute the STG edge. These probabilities are used to evaluate the probability of being in a particular state, using the technique described in [10].

The derived state probabilities are used to rank transitions in decreasing order of *relative frequency*, as follows: the relative frequency of a transition along edge e in the STG is computed as the product of the probability of being in $Source(e)$ and the probability of e being taken, given that the current state is $Source(e)$. Edges, whose relative frequency exceeds a threshold are chosen for further processing.

From the chosen set of edges, STG blocks are formed in the following manner: the edge set is traversed in decreasing order of frequency. The source and sink nodes of the first edge are grouped into a block, B. If the source (n_{source}) and sink (n_{sink}) nodes of any subsequent edge, e_{next}, do not belong to any existing block, a new block with contents n_{source} and n_{sink} is created. If the source (sink) node of e_{next} belongs to an existing block, B', but the sink (source) node, n_{sink} (n_{source}), does not, B' is augmented with the addition of n_{sink} (n_{source}). If e_{next} has its source node in one block, and its sink node in another, the two blocks are fused into one. The created blocks are optimized by the application of transformations, as described in the following section.

4.2 Controlling the application of transformations

As mentioned in the previous section, the partitioned STG is modified by the application of transformations chosen from a library. In this section, we describe the procedure used to control the application of transformations to an STG block.

Figure 6 describes the part of our algorithm which corresponds to steps **3**, **4**, **5**, **6** and **7** in Figure 5. The inputs are the STG block, *B*, to be modified, functional unit selection information, allocation constraints, clock period, the transformation library, and the objective function (throughput or power). The approach we use combines some aspects of both simulated annealing and iterative improvement. At each stage of the algorithm, we maintain a set, *In_set*, of solutions which serve as starting points for future exploration. These solutions are CDFGs which are the result of the application of a sequence of transformations to the CDFG derived from the input STG block. The neighborhood of each element of *In_set* is explored to identify candidate transformations. The loop in statement **5** of the pseudocode in Figure 6 represents iteration through *In_set*, and statement **6** represents exploration of the neighborhood of an element, *g*, of *In_set* by identifying candidate transformations which can be applied to *g*. As shown in statement **7**, CDFGs obtained by the applications of the candidate transformations are grouped into a single set, *Behavior_set*. Statements **8**-**10** represent the assessment of the efficacy of different CDFGs in *Behavior_set*. Rescheduling is performed prior to estimation with the intent of enhancing the accuracy of the process. The elements of *Behavior_set* are then ranked in decreasing order of gain, *i.e.*, if there are N candidate CDFGs, the one assessed to be the "best" ("worst") will rank 0 ($N-1$). A fixed-size subset of these CDFGs, selected and stored in *In_set*, serves as the starting point for the next iteration of the loop represented by statement **3**. The selection process is explained next.

The following principle guides our algorithm: better solutions are selected with a higher probability. In the initial phase of the algorithm, bad solutions get selected with relatively high probabilities, and as the algorithm proceeds, their probabilities of selection drop. We, therefore, generate a stream of unique random numbers whose size equals the size of the required subset. The ratio of the probabilities of selection of two solutions, which rank $n1$ and $n2$ is given by e^{-kn1}/e^{-kn2}. The parameter k is low at the beginning of the algorithm, which implies that poorer solutions have a higher probability of being selected. As the algorithm proceeds, the value of k increases, favoring better solutions. The algorithm terminates if an iteration of the loop represented by statement **3** does not improve the quality of solution. k is a linear function of the number of executions of the loop represented by statement **3**.

The rationale behind our algorithm is as follows: we try to simultaneously explore multiple regions of the solution space. If at any point, we have multiple possible solutions, simulated annealing or iterative improvement would choose one, based on a heuristic, while our algorithm would choose more solutions, thus increasing the probability of generating a better solution.

```
Apply_transforms (STG_BLOCK B, ALLOCATION_CONSTRAINT C,
 TRANSFORM_LIBRARY T_lib, CLOCK_PERIOD clk,
 FUNCTIONAL UNIT SELECTION fu_select, OBJECTIVE obj){
0  CDFG G = extract_CDFG(B);
1  SET <CDFG> In_set = {G};
2  while (stopping criterion not satisfied){
3     for(i = 0; i < MAX_MOVES; i++){
4        SET <CDFG> Behavior_set = Φ;
5        foreach element (g, In_set){
6           SET <CDFG> beh_set = Identify_and_apply
            _candidate_transformations(g, T_lib) ;
7           Behavior_set = Behavior_set ∪ beh_set;
         }
8        foreach element (b, Behavior_set) {
9           Reschedule CDFG b using fu_select;
10          Estimate objective function;//power or throughput
11          Store best solution seen so far;
         }
12       LIST <CDFG> Behavior_list = Sort (Behavior_set);
         //Sort candidate CDFGs in decreasing order of objective function
13       Select a fixed-size subset, s, of Behavior_set
         //based on a random variable distributed as c × k × e^(-kx)
         //where c is a constant, k is a monotonically increasing
         //function of i, and x is a uniform random variable.
14       In_set = s;
}}}
```

Figure 6: Controlling the application of transformations

5 Experimental Results

The techniques described in this paper were implemented in a program called *FACT*, written in C++. We evaluated this program by using it to transform several commonly available benchmarks, to optimize for throughput and power. We compare our method, *FACT*, with the technique presented in [7] (*Flamel*), and another technique, which will be called *M1* for the remainder of this discussion. *Flamel* applies the same transformation suite as our method does, and also has the ability to transcend basic blocks in its optimization, which makes it an ideal candidate for comparison. All

[2] $Source(e)$ represents the state that sources e.

Table 2: Throughput and power results

Circuit	*Clk*	*T-opt.*			*P-opt*	
		M1	*Fl*	*FACT*	*M1*	*FACT*
	ns	*T*	*T*	*T*	*P*	*P*
GCD	25	6.3	10.1	16.9	2.8	0.9
FIR	25	167	167	1000	7.6	1.7
Test2	25	2.0	2.0	2.5	11.3	8.4
SINTRAN	25	1.3	1.7	2.5	11.4	4.0
IGF	25	0.2	0.3	0.3	9.1	7.0
PPS	25	125	333	333	9.9	3.6

Table 3: Allocation constraints for the examples in Table 2

Circuit	*a1*	*sb1*	*mt1*	*cp1*	*e1*	*i1*	*n1*	*s1*
GCD	-	2	-	1	1	-	-	-
FIR	1	4	1	-	-	-	4	-
Test2	2	2	-	2	-	2	-	-
SINTRAN	4	4	5	1	-	1	2	-
IGF	1	1	2	1	-	1	-	1
PPS	5	-	-	-	-	-	-	-

methods have access to our in-house scheduling algorithm [13], which can perform the following transformations in an integrated fashion: loop unrolling, functional pipelining (even across **if** constructs), and *concurrent loop optimization* (the ability to parallelize the execution of independent loops, possibly with unknown bounds). Method *M1* just takes the input CDFG through behavioral synthesis, giving it access to only those transformations supported by our scheduling algorithm. Note that recent work on power-optimizing transformations, *e.g.* [5], has focussed on DFI descriptions, making them inapplicable to many of our benchmarks.

Comparisons were performed with respect to throughput and power in the following manner. The CDFGs were synthesized, placed, and routed, using our in-house synthesis tool. Throughput was estimated as the inverse of the expected execution time between consecutive iterations of the CDFG, using techniques presented in [10]. Power estimation was performed using a transistor-level netlist, extracted from the layout, using a switch-level simulator, IRSIM-CAP. The inputs to the power estimator were derived using a zero-mean Gaussian sequence, which was subsequently passed through an autoregressive filter to introduce the desired level of temporal correlation.

Table 2 summarizes the results obtained. *Clk* represents the clock period constraint. The columns labeled *T-opt* and *P-opt* represent, respectively, the CDFGs produced by the application of transformations aimed at throughput and power optimization. Minor column *M1* represents method *M1*, and minor columns *Fl* and *FACT* represent CDFGs which have been transformed by the application of techniques presented in [7] and this paper, respectively. Columns *T* and *P* represent the throughput (measured in $cycles^{-1} \times 1000$) and power (measured in *mW*), respectively, of the circuits synthesized from the corresponding CDFG, measured in the manner outlined before (*e.g.*, for *FIR*, *FACT* produces a throughput of 1 $cycle^{-1}$ in the throughput optimization mode). All CDFGs were synthesized under the same allocation constraints, to meet the same target clock period. We used a library of functional units which consists of (a) an adder, *a1*, with a delay of 10*ns*, (b) a subtracter, *sb1*, with a delay of 10*ns*, (c) a multiplier, *mt1*, with a delay of 23*ns*, (d) a less-than comparator, *cp1*, with a delay of 10*ns*, (e) an equality comparator, *e1*, with a delay of 5*ns*, (f) an incrementer, *i1*, with a delay of 5*ns*, (g) a multi-bit inverter, *n1*, with a delay of 2*ns*, and (h) a shifter, *s1*, with a delay of 10*ns*. The allocation constraints for an example can be found by looking up the entry corresponding to the example in Table 3. For example, the allocation constraint for *GCD* is two *sb1*, one *cp1*, and one *e1*.

Of our examples, *GCD* computes the greatest common divisor of two numbers, *FIR* is a finite impulse response filter, *SINTRAN* evaluates the sine transform, *IGF* evaluates the incomplete gamma function, *PPS* is a parallel prefix sum, and *Test2* is the example shown in Figure 2(a).

The results obtained indicate that the designs derived from *FACT*, in the throughput optimization mode, produced on an average a 2.7-fold improvement in throughput over designs produced using *M1* and 2.1-fold improvement over designs derived from *Flamel*. The power-optimized designs produced by *FACT* consumed on an average 62.1% less power than circuits synthesized from *M1*, for the same throughput (the throughput was made the same as *M1*'s throughput in the throughput-optimized case). We do not compare the results of *FACT* in the power optimization mode with *Flamel* since the latter does not have the ability to apply power-optimizing transformations.

6 Conclusions

In this paper, we presented a framework, *FACT*, for applying throughput and power-optimizing transformations to CFI behavioral descriptions. Our algorithm interleaves scheduling with transformation application, which allows accurate identification of performance or power bottlenecks, and the selection of transformations to eliminate them. The applied transformations can span several basic blocks, thus improving their optimizing capabilities. This ability is the result of a general technique we have devised for recognizing and applying a general class of transformations across the scope of basic blocks. The CDFGs produced by *FACT*, upon synthesis, have achieved upto 4.5-fold improvement in power and upto 6-fold improvement in throughput over designs synthesized without the benefit of our technique.

References

[1] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys*, vol. 26, pp. 346–420, Dec. 1994.

[2] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*. Narosa Publishing House, New Delhi, 1985.

[3] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using profile information to assist classic code optimizations," *Software Practice & Experience*, vol. 21, pp. 1301–1321, Dec. 1991.

[4] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proc. SIGPLAN Conf. Programming Language Design & Implementation*, pp. 16–27, June 1990.

[5] A. P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, and R. W. Brodersen, "Optimizing power using transformations," *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 12–51, Jan. 1995.

[6] M. B. Srivastava and M. Potkonjak, "Power optimization in programmable processors and ASIC implementations of linear systems: Transformation-based approach ," in *Proc. Design Automation Conf.*, pp. 343–348, June 1996.

[7] H. Trickey, "Flamel: A high-level hardware compiler," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 259–269, Mar. 1987.

[8] A. Nicolau and R. Potasman, "Incremental tree height reduction for high level synthesis," in *Proc. Design Automation Conf.*, pp. 770–774, June 1991.

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

[10] 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.

[11] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*. Addison-Wesley, Reading, MA, 1988.

[12] G. Lakshminarayana and N. K. Jha, "Throughput and power optimizing transformations for control-flow intensive behavioral descriptions," Tech. Rep. CE-J98-004, Princeton University, 1998.

[13] G. Lakshminarayana, K. S. Khouri, and N. K. Jha, "*Wavesched:* A novel scheduling technique for control-flow intensive behavioral descriptions," in *Proc. Int. Conf. Computer-Aided Design*, pp. 244–251, Nov. 1997.