IMPACT: A High-Level Synthesis System for Low Power Control-Flow Intensive Circuits*

Kamal S. Khouri, Ganesh Lakshminarayana, and Niraj K. Jha

Department of Electrical Engineering
Princeton University, Princeton, NJ 08544

Abstract

In this paper, we present a comprehensive high-level synthesis system that is geared towards reducing power consumption in control-flow intensive circuits. An iterative improvement algorithm is at the heart of the system. The algorithm searches the design space by handling scheduling, module selection, resource sharing and multiplexer network restructuring simultaneously. The scheduler performs concurrent loop optimization and implicit loop unrolling. It minimizes the expected number of cycles of the schedule without compromising on the minimum and maximum schedule lengths. A fast simulation technique based on trace manipulation aids power estimation in driving synthesis in the right direction. Experimental results demonstrate power reduction of up to 85% with minimal overhead in area over area-optimized designs operating at 5V.

1 Introduction

The increase in demand for portable electronics has resulted in a proliferation of research initiatives for reducing power consumption at different levels of the design hierarchy. We present a comprehensive high-level synthesis system, with a low-power directive, for applications comprised of both control and data processing.

It is shown in [1] that significant savings in power may be achieved at higher levels in the design hierarchy. High-level synthesis applications may be categorized as data-dominated, control-flow intensive (CFI), or control-dominated. Most high-level synthesis techniques for low power have targeted data-dominated designs, namely for digital signal processing and image processing applications [2]-[6]. However, today's network-centric systems are more likely to require both control and data processing, and may contain a large number of nested loops and conditionals. These CFI circuits include network protocol handlers, modern ATM network switches and graphics controllers. Control-dominated circuits, such as sequencers, consist almost exclusively of control-flow. A high-level synthesis system for such circuits is given in [7]. Past work in high-level synthesis for CFI designs has mostly targeted performance and area, with the exception of [8], where a profile-driven technique for low power is presented. In [9, 10], timing analysis and loop-directed scheduling drive high-level synthesis of CFI circuits. A number of efficient algorithms are applied to each high-level synthesis task in the system presented in [11]. The systems described in [9]-[11] do not target low power consumption. However, some general estimation techniques have been proposed in [12] where a user-defined cost function drives synthesis.

In this paper, we present the first comprehensive high-level synthesis system geared towards achieving low power designs for CFI applications. Our algorithm can efficiently handle data-dominated applications as well. The algorithm is based on iterative improvement and allows an exploration of the design space by interleaving synthesis tasks such as scheduling, allocation, resource sharing and module selection and, hence, exploits their interdependency to reach a solution. In addition, the algorithm specifically targets power reduction in multiplexer networks, which has been shown to account for more than 40% of power consumption in CFI circuits [13].

*Acknowledgments: This work was supported in part by Alternative System Concepts under an SBIR contract from Air Force Rome Laboratories and in part by NSF under grant No. MIP-9319269.

2 Preliminaries

High-level synthesis is the process of translating a circuit's behavioral description into an appropriate register-transfer (RT) level design. Properties such as unbounded loops, and variable path delays make high-level synthesis tasks such as scheduling, allocation and assignment, and clock and module selection very challenging for CFI circuits. This section describes a model for representing the behavior, and discusses scheduling and power estimation techniques for CFI specifications. The remaining high-level synthesis tasks are covered in Section 3.

2.1 The Control-Data Flow Graph Model

In high-level synthesis, directed graph structures called data-flow graphs (DFGs), control-flow graphs (CFGs) and control-data flow graphs (CDFGs) have been used as the intermediate representation for a behavioral description. DFGs are usually associated with data-dominated designs, while CFGs are associated with CFI designs. In [14], it is argued that while these two representations simplify tasks, such as scheduling, each has its disadvantages: DFGs cannot represent control structures, and CFGs cannot represent parallel processes. The problem is partially solved by using both DFGs and CFGs, where the DFGs represent basic blocks of code. This solution produces artificial boundaries, however, between basic blocks and prevents optimizations that may result from migrating operations between the basic blocks. To overcome these obstacles, DFGs and CFGs are combined into one model called a CDFG [11, 15, 16]. Our CDFG model is a variation of that introduced in [11], as described below.

We start with an input specification described in a hardware description language that has been compiled into a CDFG. Figure 1 illustrates an example of a CDFG (called Loops), with one conditional branch, and three loops. The two rightmost loops execute concurrently if condition *c* is false. The CDFG nodes are comprised of arithmetic, logical and comparison operations, and special data merge and loop exit nodes. The edges in the graph connect various nodes and, hence, describe the data and control inter-node dependencies. A node n is *data-dependent* on node n' if n uses the result from n' to evaluate its output. Node n is *control-dependent* on n' if it uses the result from n' to decide whether it should execute or not.

In Figure 1, some nodes represent operations that map directly from functions in the behavioral description. These include arithmetic functions such as ADD ($+_1$), MULTIPLY ($*_2$), comparison

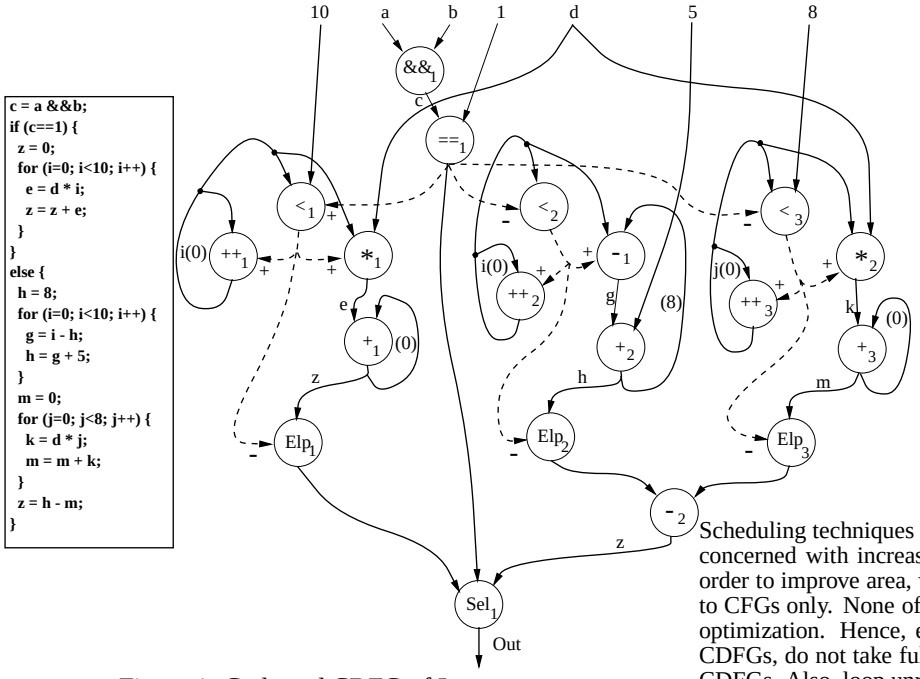

Figure 1: Code and CDFG of Loops

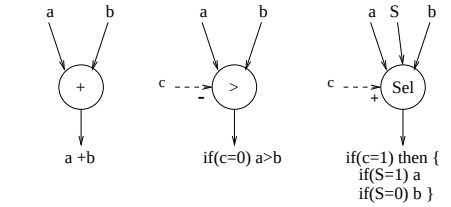

Figure 2: Control ports and their effect

functions such as LESS THAN ($<_3$), EQUAL TO ($==_1$), and Boolean functions such as AND ($\&\&_1$). Other nodes do not map directly from a function, but are used to help represent structures such as loops and branches. These *structural* nodes include select (*Sel*) and end-loop (*Elp*). *Sel* is similar to a 2-to-1 multiplexer and is used to merge branches from a conditional fork. One *Sel* node merges the two branches of condition c in Figure 1. *Elp* nodes terminate loops, and store any values that need to be passed on to nodes outside the loop body. In Figure 1, there are three loop structures terminated by *Elp* nodes.

In previous works that utilize CDFGs, it has been the convention to represent control dependencies by edges in the graph. We introduce the concept of *control ports* for this purpose. A control port is an abstraction that accepts an edge as an input, and evaluates the value on the edge independently of the operation performed by the node. Each node has exactly one control port, and each control port has an assigned *polarity* that describes the control condition required for its node to execute. There are three possible conditions: *active-high* (node executes on a true value, shown as a "+"), *active-low* (node executes on a false value, shown as a "−") or *null* (node is control-independent). Given that the data values have arrived at all its inputs, a node will execute its assigned operation when the edge entering its control port carries a value that matches its polarity. Figure 2 gives examples of the control-port representation. In Figure 1, there are seven nodes with positive polarities, five with negative polarities, and seven without any explicit control dependencies.

With the above convention in place, edges become only carriers of data values, and are no longer concerned with whether this data is for control or processing. However, for ease of understanding, edges that enter control ports are illustrated as dashed arrows, and others as solid arrows. An edge may carry a constant value (*e.g.* 10), or a variable (*e.g. a*) that may be modified throughout the CDFG. Edges may also carry an initial value, which is denoted by a number in braces. This is useful for loop iterators such as j in Figure 1, where the edge for j is initially set to 0.

2.2 Scheduling of CFI Behavioral Descriptions

Scheduling is the process of assigning nodes in the CDFG to states, and connecting the states via conditions to form a state transition graph (STG) [17]. Scheduling techniques for CDFGs given in [11, 16] are primarily concerned with increasing resource sharing among operators in order to improve area, while those given in [9, 17] are applicable to CFGs only. None of these techniques support concurrent loop optimization. Hence, even the techniques that are applicable to CDFGs, do not take full advantage of the inherent parallelism of CDFGs. Also, loop unrolling is only handled in [9].

We use a new scheduling algorithm called *Wavesched* [18]. *Wavesched* supports loop unrolling and concurrent loop optimization, and attempts to minimize the expected number of cycles (ENC) [9] of a schedule without compromising on the minimum and maximum schedule lengths. *Wavesched* has been shown to reduce the ENC by up to a factor of five over the schedule derived by the techniques presented in [9, 17]. Once an STG is available, both the datapath and controller may be optimized for low power dissipation.

2.3 Power Estimation

In high-level synthesis, it is computationally impractical to perform accurate gate-level simulations to estimate power at each step of the synthesis process. For this reason, we need to rely on a model at a higher level, that will be relatively accurate to drive synthesis in the right direction. In [19], a power estimation technique for CFI circuits, that takes glitches into account and uses signal statistics, is presented. The mean and standard deviation of the switching activities, and the spatial and temporal correlation of signals are evaluated for each functional unit, register, and multiplexer, to produce an accurate estimate of the overall power consumption.

The necessary statistics may be obtained by performing behavior level or RT level simulations of the design using typical input sequences. It might be desirable to perform the simulations at each step of the synthesis process. However, by doing so, the synthesis system will incur a large runtime performance hit. To overcome this obstacle, an RT level simulation technique based on *trace*

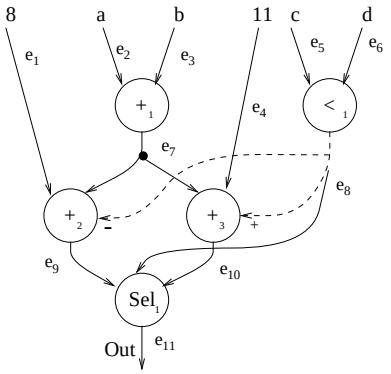

Figure 3: 3-addition example CDFG

manipulation is presented here. This technique records signal traces of the inputs and outputs of each functional unit, register, and multiplexer, and transforms signals appropriately when a synthesis task (resource sharing, module selection) is executed, without the need for re-simulation. A trace is a sequence of vectors which consists of signal values that appear at the input and output of an RT level unit. For example, consider an adder with input vectors $i_1 = [0101]$ and $i_2 = [0001]$. The output vector for the adder, as a result of these inputs, is $[0110]$. Therefore, the adder's trace for this particular input set is $[0101, 0001|0110]$. Values at the inputs and output of the adder are evaluated via simulation, and an accurate simulation requires hundreds of input vectors. Hence, the trace for the adder is the collection of all the different input and corresponding output vectors, vertically ordered in time. Considering there are hundreds of vectors for each RT level unit, it is clear why repeated simulations are not desirable.

To explain trace manipulation, consider the CDFG given in Figure 3. An initial behavioral simulation will produce traces for each of the edges e_1 through e_{11} in the CDFG. The symbolic traces (*i.e.* in terms of e_1 through e_{11}) for the three addition operations are:

Trace	In_1	In_2	Out
$TR(+_1)$	e_2	e_3	e_7
$TR(+_2)$	e_1	e_7	e_9
$TR(+_3)$	e_7	e_4	e_{10}

For the fully parallel architecture (one-to-one mapping of the CDFG), given in Figure 4, the trace for each adder is the same as the trace for the corresponding addition operation, or:
$TR(+_1) = TR(A_1), TR(+_2) = TR(A_2), TR(+_3) = TR(A_3)$.

Next, assume that there is only one adder to use. The resulting RT level implementation of the CDFG is shown in Figure 5. This implementation may be re-simulated to evaluate the traces. However, simply merging the previous traces will produce the same result faster. The trace for the adder becomes:

Trace	In_1	In_2	Out
$TR(A_1\|e_8)$	e_2	e_3	e_7
	e_7	e_4	e_{10}
$TR(A_1\|\bar{e_8})$	e_2	e_3	e_7
	e_7	e_1	e_9

The trace is dependent on condition e_8. Suppose that for a set of four input passes, the condition evaluates as e_8 = [T, T, F, T]. It may be seen from the STG shown in Figure 6 that the traces of $(+_1, +_3)$, $(+_1, +_3)$, $(+_1, +_2)$, and $(+_1, +_3)$ need to be merged. The resulting trace becomes:

Trace	In_1	In_2	Out
$TR(A_1)$	e_2	e_3	e_7
	e_7	e_4	e_{10}
	e_2	e_3	e_7
	e_7	e_4	e_{10}
	e_2	e_3	e_7
	e_7	e_1	e_9
	e_2	e_3	e_7
	e_7	e_4	e_{10}

It is clear from the above example that, in addition to knowing the traces, the STG of the CDFG must be available. The STG determines (1) the order and types of operations, and (2) the conditions under which they execute. In general, for each functional unit D_u there is a matrix $TR(op_i)$, for each operation op_i that is mapped to D_u. $TR(op_i)$ is a $[(e + o) \times 1]$ matrix, where e is the number of input edges to the operation and o is the number of output edges (typically 1). For a path P through the STG, the traces of each operation op_i encountered on P, $TR(op_i), T(op_{i+1})...T(op_{i+n})$, are merged to form the trace of D_u, given by $TR(D_u)$. A similar approach is used for registers and multiplexers, where the traces for CDFG variables and edges are merged. These traces are then used to obtain the statistics required for the CFI power estimator presented in [19], that enable it to compute a power number for the given synthesis step.

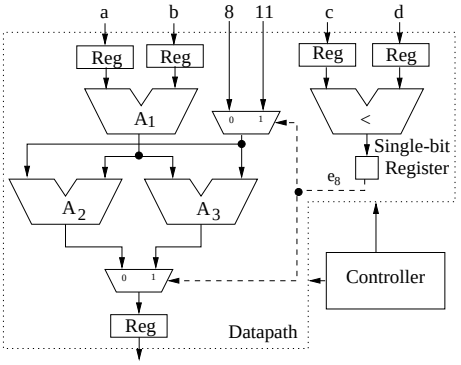

Figure 4: Parallel RT architecture for 3-addition example

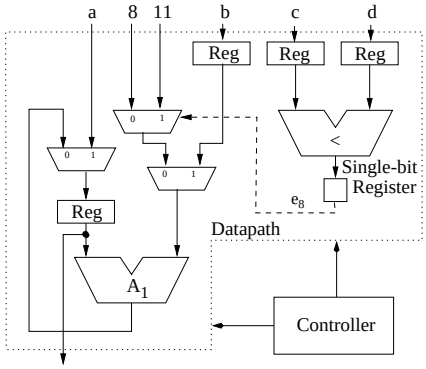

Figure 5: Shared RT architecture for 3-addition example

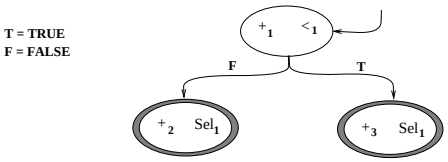

Figure 6: STG for the trace manipulation example

A synthesis process may proceed entirely on one initial behavioral simulation as long as certain types of moves are not made in the synthesis process. The moves are those that introduce paths in the CDFG that may not have been exercised in the initial simulation. Re-simulation is done on an as-needed basis, hence amortizing the cost of one simulation over multiple instances of trace manipulation.

3 The Synthesis Algorithm

For even moderately-sized designs, complex interaction among synthesis tasks makes it computationally infeasible to exhaustively search for an optimal solution. To obtain good results in a relatively short amount of time, we must resort to heuristic methods. The concepts presented in Section 2 go hand-in-hand with an iterative improvement algorithm to form the synthesis system. Our algorithm is called IMPACT (Iterative iMprovement, Power optimizing Algorithm for Control-flow inTensive designs). While developed primarily for CFI designs, the system can efficiently handle data-dominated designs as well.

3.1 Iterative Improvement

In [20], iterative improvement has been shown to be an efficient algorithm in terms of both execution time and quality of results, which are both desirable for high-level synthesis tasks. In [3], an iterative improvement algorithm, called *SCALP*, has been presented for high-level synthesis of low power data-dominated circuits. This algorithm has the ability to simultaneously handle scheduling, module selection, and resource sharing to arrive at efficient architectures. It is based on a *variable-depth search* strategy, where sequences of *moves* are applied to an initial RT level architecture and evaluated. The sequence that produces the best improvement or *gain*, for the specified cost function, is chosen. The algorithm has the ability to escape local minima by allowing individual moves in a sequence to have negative gain. IMPACT retains the positive features of *SCALP*, while significantly generalizing its capabilities to handle CFI behavioral descriptions, not just data-dominated descriptions.

Figure 7 is a block diagram illustrating our high-level synthesis system, IMPACT. The initial steps of the algorithm are straightforward. First, a behavioral simulation is performed to obtain the traces and statistics necessary for power estimation. The initial RT level architecture is a parallel architecture, in which each node is assigned to a separate functional unit, each functional unit is chosen to be the fastest module available in the library, and each variable is assigned to a separate register. The CDFG is scheduled using a clock period defined by the designer. The signal trace statistics are then plugged into the power estimator to produce a power number. Together with the parallel RT level architecture, the power number serves as an initial solution to the iterative improvement algorithm. The algorithm exits once further power reduction is no longer possible.

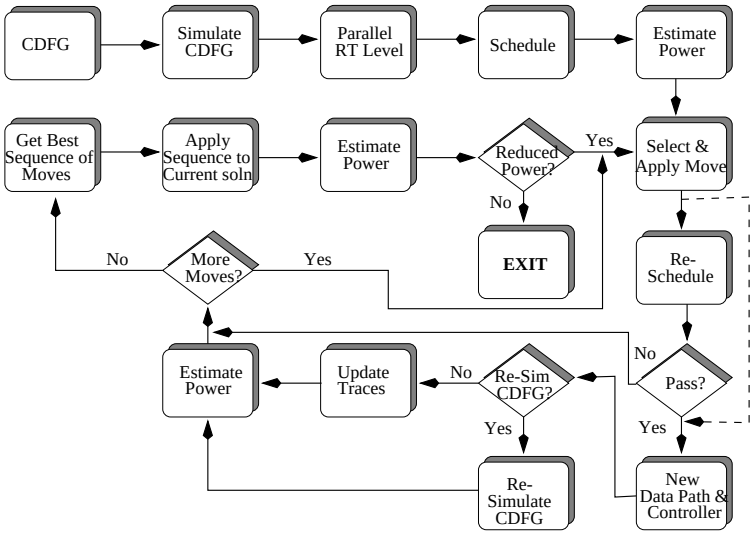

Figure 7: Block diagram for IMPACT

3.2 Moves for Iterative Improvement

In the subsequent subsections, we describe the different iterative improvement moves that are applied during the synthesis process. First, we introduce a new RT level transformation technique that specifically targets power reduction in multiplexer networks. We then proceed to describe the module selection and resource sharing moves.

3.2.1 Multiplexer Tree Restructuring

Interconnect in the form of multiplexer networks may consume more than 40% of the total power of a CFI circuit [13]. We introduce a new RT level technique that targets multiplexer trees and restructures them to reduce power consumption. Multiplexer decomposition has also been used at the logic level to

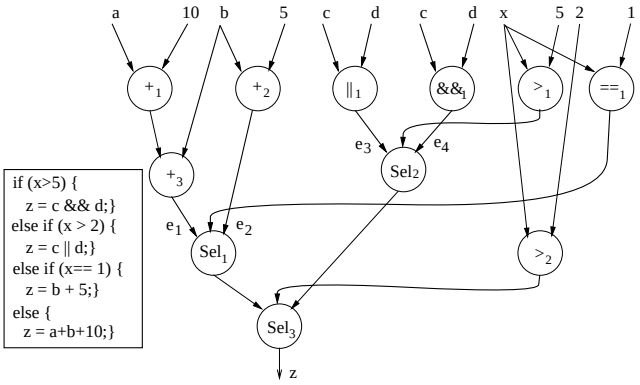

Figure 8: CDFG for the multiplexer example

reduce power consumption [21]. To motivate this move, consider the CDFG given in Figure 8. The following information is given: the clock period is $15ns$, the delay of an adder is $10ns$, a chained adder incurs 10% delay overhead, and the delay of a multiplexer is $3ns$. Trace analysis provides the activity measures for each RT level unit and the probabilities for each branch being taken. Suppose the relative switching activity (branch probabilities) are found to be: e_1 =0.6(0.7), e_2 = 0.1(0.2), e_3 = 0.2(0.05) and e_4 = 0.1(0.05).

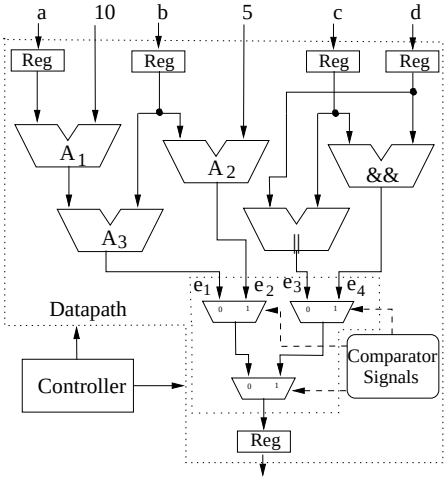

Figure 9: Balanced mux tree RT level design

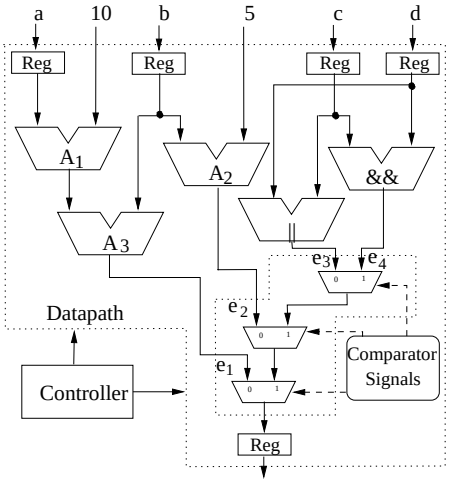

Figure 10: Unbalanced mux tree RT level design

Two RT level implementations of the CDFG of Figure 8 are shown in Figures 9 and 10. The first is implemented using a balanced multiplexer tree, while the second is not. By examining Figure 9, the following observations can be made: (1) the delay through the first branch (e_1), with two chained adders and two multiplexers, will be $> 15ns$ ($10ns + 1ns + 3ns + 3ns$), and hence require two cycles, and (2) the signals from the two most probable branches (e_1 and e_2) have to travel through two multiplexers to reach the output. We claim that the implementation in Figure 10 consumes less power. Removing a multiplexer from the most probable branch path: (1) reduces the delay of the path to $< 15ns$ ($10ns+1ns+3ns$), saving a cycle and hence enabling V_{dd} scaling, and (2) reduces the switching in the multiplexer tree by 34%. Switch-level simulations (based on a switch-level circuit extracted from the layout) verify our claim, and show that the implementation of Figure 9 consumes 10.1 mW of power, while that of Figure 10 consumes 6.0 mW of power.

An *n*-to-1 multiplexer is represented as a tree of 2-to-1 multiplexers. The intuition behind the multiplexer move is to restructure a tree such that signals with high activity-probability (*ap*) product are closer to the output. Doing so reduces the switched capacitance in the high activity paths and, hence, reduces power consumption. In the following analysis, a probabilistic approach is used and some well-known algorithms from the coding theory domain are applied.

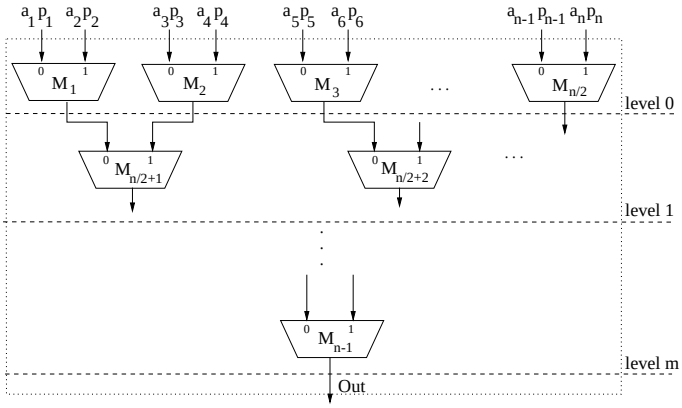

Figure 11: *n*-to-1 mux composed of 2-to-1 muxes

Examine the *n*-to-1 multiplexer shown in Figure 11. From the STG and trace simulation, two important statistics may be derived: (1) the transition activity number a_i, which represents the level of switching in input signal i, and (2) the probability of propagation p_i, which represents the probability that a signal i will appear at the output of the multiplexer tree (this is synonymous with the branch probability of a path). The sum of p_i's for a given multiplexer tree is 1 (*i.e.* $\sum_{i=1}^{n} p_i = 1$). The switching activity of the multiplexer tree is defined as:

$$A_{tree} = \sum_{k=1}^{n-1} A_k, \tag{1}$$

where A_k is the activity of individual 2-to-1 multiplexers, and the switching activity of an individual leaf (level-0) multiplexer, is given as the sum of the *ap*'s normalized by the respective probabilities or:

$$A_k = \frac{a_i p_i + a_{i+1} p_{i+1}}{p_i + p_{i+1}} \qquad 1 \le k \le \frac{n}{2} \tag{2}$$

The activity at the output of the leaf multiplexers is also given by the above equation, while the output probability of the 2-to-1 multiplexer is the sum of its input probabilities. The input activity (probability) of a level-1 multiplexer (*e.g.* $M_{n/2+1}$) is the output activity (probability) of the multiplexers at its inputs. For $M_{n/2+1}$ the activity is given by :

$$A_{M_{n/2+1}} = \frac{A_{M_1}(p_1 + p_2) + A_{M_2}(p_3 + p_4)}{(p_1 + p_2) + (p_3 + p_4)} \text{ and,} \tag{3}$$

$$A_{M_1} = \frac{a_1 p_1 + a_2 p_2}{p_1 + p_2}, \tag{4}$$

$$A_{M_2} = \frac{a_3 p_3 + a_4 p_4}{p_3 + p_4} \tag{5}$$

Putting Equations (4) and (5) into Equation (3) yields

$$A_{M_{n/2+1}} = \frac{a_1 p_1 + a_2 p_2 + a_3 p_3 + a_4 p_4}{p_1 + p_2 + p_3 + p_4} \tag{6}$$

Therefore, the activity of a level-1 multiplexer is equivalent to having the four inputs of its predecessors directly feeding it (*i.e.* it appears to be a 4-to-1 multiplexer), and may be found using the same analysis as the leaf multiplexers. The total activity of the 4-to-1 multiplexer represented by M_1, M_2 and $M_{n/2+1}$ is the summation of Equations (4), (5), and (6). Performing this analysis recursively down the multiplexer tree results in a value for its switching activity. The equation will be of the form:

$$\begin{aligned} A_{tree} = {} & \frac{a_1 p_1 + a_2 p_2}{p_1 + p_2} + \frac{a_3 p_3 + a_4 p_4}{p_3 + p_4} + \ldots + \\ & \frac{a_{n-1} p_{n-1} + a_n p_n}{p_{n-1} + p_n} + \\ & \frac{a_1 p_1 + a_2 p_2 + a_3 p_3 + a_4 p_4}{p_1 + p_2 + p_3 + p_4} + \ldots + \\ & \frac{a_{n-3} p_{n-3} + a_{n-2} p_{n-2} + a_{n-1} p_{n-1} + a_n p_n}{p_{n-3} + p_{n-2} + p_{n-1} + p_n} + \\ & \vdots \\ & a_1 p_1 + a_2 p_2 + a_3 p_3 + \ldots + a_n p_n \end{aligned} \tag{7}$$

Returning to the example of Figure 8, applying these equations to the multiplexer tree in Figure 9, the activity is found to be 1.09. The activity in the multiplexer tree in Figure 10 is 0.72, a reduction of 34%.

```
RESTRUCTURE_MUX (Datapath D, Signals_List S) {
  Ordered Signals List S_ord = ORDER_SIGNALS_LIST(S);
  //Place signals in increasing ap order
  HUFFMAN_CONSTRUCT(S_ord);
}

HUFFMAN_CONSTRUCT (Signals_List S) {
  if (size_of(S) > 1) {
    ASSIGN_MULTIPLEXER(S[0], S[1]);
    // assigns a multiplexer to the two signals and
    // removes them from list
    Activity-Probability ap_new = (PROB(S[0]) + PROB(S[1]))×
                                  Σ_{k∈sub-tree} ACTIVITY(Mux(k));
    // sum up the activity of all multiplexers in the sub-tree
    New signal s = CREATE_NEW_SIGNAL(ap_new);
    S = APPEND(s);
    S_ord = ORDER_SIGNALS_LIST(S);
    HUFFMAN_CONSTRUCT(S_ord);
  }
  return;
}
```

Figure 12: Pseudo-code for multiplexer tree restructuring

It may be seen from Equation (7) that the term for the root multiplexer is a constant and independent of how the tree is ordered. The remaining terms, however, are sensitive to ordering. Simply exchanging two of the input terms (*e.g.* $a_1 p_1$ and $a_3 p_3$) will result in a different value for switching activity. Finding an

optimal ordering (both vertical and horizontal) to minimize A_{tree} is computationally intensive. A heuristic solution to this problem may be obtained by making the following observation: ranking the inputs in the order of increasing ap's, and ignoring the normalizing denominators in each term of Equation (7), transforms the problem to a source encoding problem from coding theory. For source encoding, the goal is to assign the source-words (signals) with high probability (ap), short code words (distance to output), and source-words with low probability, longer code words. The Huffman algorithm will find an optimal solution by minimizing $\sum_{i=1}^{n} a_i p_i l_i$, where l_i is the distance of signal i from the output. However, with the presence of the normalizing terms, the Huffman algorithm is a greedy algorithm and produces only an approximate solution. Still, we can benefit from the algorithm since it is fast, and may help reduce switching activity in the multiplexer tree. Figure 12 contains the pseudo-code for the algorithm used. The algorithm creates a list by placing the signals in the order of increasing ap's. It assigns a multiplexer to the two signals with the least ap. The two signals are removed from the list, and the output of the new multiplexer becomes a new signal. The ap of this new signal is evaluated using the equations presented, and it is placed in order with the remaining signals. The algorithm proceeds recursively until all signals have been assigned to a multiplexer. A multiplexer restructuring move is allowed to increase the delay of some signals only if this is compensated for during successive moves.

3.2.2 Module Selection/Substitution

There are many VLSI implementations for different functions, and it is important to capture the diversity of these implementations in the module library. Module substitution replaces a functional unit f in the data path with another functional unit f' that performs the same operation as f, but has different properties in terms of performance, area, and power (*e.g.* replace an *array multiplier* with a larger and faster *wallace_tree multiplier*). Moves of this type may or may not require re-scheduling of the CDFG. Rescheduling may be avoided if replacing f with f' does not violate the cycle time constraint in every state that f appears in (shown by the dashed arrow in Figure 7). If a violation occurs, then it becomes necessary to re-schedule the CDFG. Hence, replacing f with a faster instance f' does not require a re-scheduling. We can generalize this method by allowing intermediate solutions to violate the cycle time constraint, as long as a later move ensures that the constraint is met.

3.2.3 Resource Sharing/Splitting

Resource sharing is the process of using the same hardware to perform different operations (of the same type) in the CDFG, or to store values of different variables. Resource sharing may have both an adverse and positive effect on power consumption. Increased resource sharing tends to reduce physical capacitance, but also increases the amount of switching activity. Performing resource sharing (splitting) allows the system to examine the trade-offs between increasing (decreasing) switching activity and reducing (increasing) circuit capacitance. A resource split is the process of assigning separate functional units (registers) to operations (variables) that originally shared the same functional unit (register). For example, suppose two operations $+_2$ and $+_3$ share an adder α. A split will assign a new functional unit, adder β (of the same library type as α) to $+_3$.

Resource sharing may only occur between two similar operations, unless the library element performs multiple functions (*e.g.* an ALU). When a resource sharing move is performed, rescheduling is necessary since the operation assignment set is reduced in size by one functional unit. On the other hand, rescheduling is not needed after a resource split because the new assignment set is a superset of the previous one.

Resource sharing across mutually exclusive operations (operations that can never execute at the same time) can help reduce the number of states. This does not imply an improvement in performance, but it does usually imply a smaller controller, which in turn may be beneficial for area and power.

4 Experimental Results

Next, we present the results obtained by the IMPACT high-level synthesis system. The system is implemented in C++. The benchmarks used are our Loops example shown in Figure 1, Greatest Common Divisor (GCD) [22], the send process of the X.25 communications protocol [9], a Blackjack dealer process (Dealer) [10], the co-ordinate transformation algorithm Cordic [2] and Paulin [23]. The last benchmark is used to demonstrate the system's ability to efficiently handle data-dominated designs.

Each of the benchmarks is compiled into a CDFG and fed to IMPACT. The synthesis process attempts to reduce power consumption by using the algorithm described in Section 3. The MSU standard cell library with the logic synthesis tool *SIS* is used to transform the RT level design into a logic-level netlist. The *Octtools* suite is then used to perform layout and routing of the controller and datapath, followed by *IRSIM-CAP*, a switch-level simulator (which works on a switch-level description extracted from the layout), for computing the power consumption of the circuits. Since the measurement of power consumption is based on layouts, various components of power, such as glitching power, clock power, interconnect power, and controller power, which are difficult to estimate at the higher levels, are taken into account.

The results are presented in Figure 13 as plots of *normalized power and area* vs. *laxity factor*. The laxity factor is the ratio of the given ENC to the minimum ENC that can be obtained using the given library of components. For each benchmark, circuits are synthesized at laxity factors ranging from 1.0 to 3.0 under area-optimization mode and then power-optimization mode. The results are normalized with respect to the base area-optimized designs operating at 5V. We plot the power consumption for the area-optimized V_{dd}-scaled circuits (A-Power), and for the power-optimized circuits (I-Power). The ENC, *i.e.* the performance, of these two sets of circuits that are compared is equal. We also plot the area of each power-optimized circuit (I-Area) normalized with respect to the base area-optimized circuit, to indicate the area overhead incurred due to power optimization.

Our power-optimized circuits result in up to 6.7-fold power reduction over the base area-optimized circuits, and up to 2.6-fold power reduction over area-optimized circuits which are also V_{dd}-scaled. The price paid for power optimization is an increase in area of no more than 30%.

5 Conclusions

In this paper, we presented a high-level synthesis system that targets CFI designs. Such designs comprise a large portion of today's portable electronics. The system uses an iterative improvement algorithm to successfully take into account the interaction among the different synthesis tasks. The techniques used to search the design space are module selection, scheduling, resource sharing/splitting and multiplexer tree restructuring, all of which are performed simultaneously. Our system employs a new scheduling algorithm that performs concurrent loop optimization and implicit loop unrolling, and a fast simulator, based on trace manipulation, to produce accurate signal statistics used for power estimation. While no other comprehensive high-level synthesis system exists for optimizing power in CFI circuits that we could compare our results to, we have achieved results comparable to those reported by comprehensive data-dominated power optimizers by reducing power consumption with small area overhead.

References

[1] R. Mehra and J. Rabaey, "Behavioral level power estimation and exploration," in *Proc. Int. Wkshp. Low Power Design*, pp. 255-270, Jan. 1994.

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

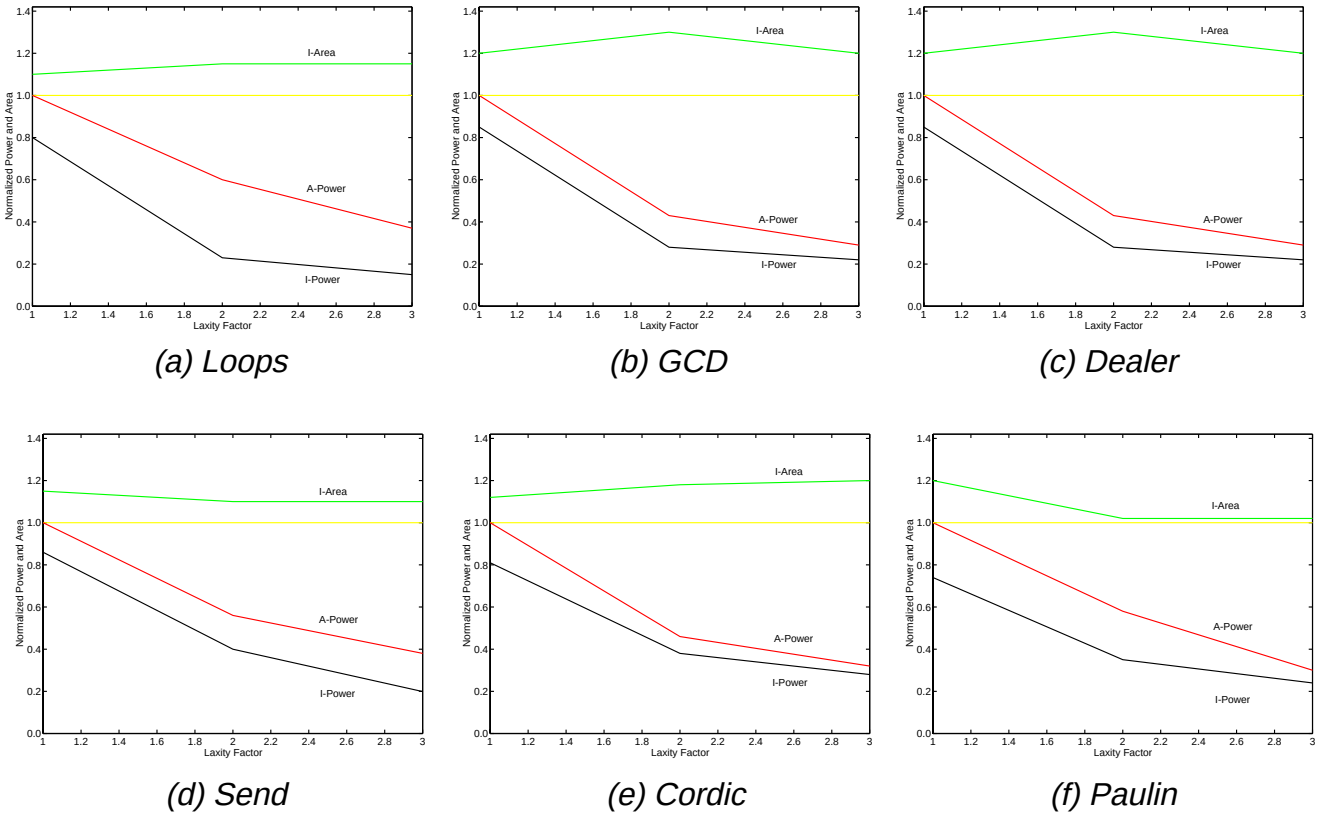

Figure 13: Normalized power and area vs. laxity factor

[3] A. Raghunathan and N. K. Jha, "An iterative improvement algorithm for low power data path synthesis," in *Proc. Int. Conf. Computer-Aided Design*, pp. 597-602, Nov. 1995.

[4] A. Chatterjee and R. K. Roy, "Synthesis of low power DSP circuits using activity metrics," in *Proc. Int. Wkshp. Low Power Design*, pp. 255-270, Jan. 1994.

[5] A. Raghunathan, N. K. Jha, and S. Dey, *High-level Power Analysis and Optimization*, Kluwer, Boston, MA, 1998.

[6] M. B. Srivastava and M. Potkonjak, "Energy efficient implementation of linear systems on programmable processors," in *Proc. European Design and Test Conf.*, pp. 267-271, Mar. 1994.

[7] I. Park, K. O'Brien, and A. A. Jerraya, "AMICAL: An interactive high-level synthesis environment," in *Proc. European Conf. Design Automation*, Feb. 1993.

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

[9] 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.

[10] S. Bhattacharya, S. Dey, and F. Brglez, "Clock period optimization during resource sharing and assignment," in *Proc. Design Automation Conf.*, pp. 195-200, June 1994.

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

[12] S. Y. Ohm, F. J. Kurdahi, N. Dutt, and M. Xu, "A comprehensive estimation technique for high-level synthesis," in *Proc. Int. Symp. System Synthesis*, pp. 122-127, Sept. 1995.

[13] A. Raghunathan, S. Dey, and N. K. Jha, "Glitch analysis and reduction in register-transfer level power optimization," in *Proc. Design Automation Conf.*, pp. 331-336, June 1996.

[14] 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. 5, pp. 82-100, Mar. 1997.

[15] K. Wakabayashi and T. Yoshimura, "A resource sharing and control synthesis method for conditional branches," in *Proc. Int. Conf. Computer-Aided Design*, pp. 62-65, Nov. 1989.

[16] T. Kim, N. Yonezawa, J. W. S. Liu, and C. L. Liu, "A scheduling algorithm for conditional resource sharing - a hierarchical reduction approach," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 425-438, Apr. 1994.

[17] R. Camposano, "Path-based scheduling for synthesis," *IEEE Trans. Computer-Aided Design*, vol. 10, pp. 85-93, Jan. 1991.

[18] 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.

[19] A. Raghunathan, S. Dey, and N. K. Jha, "Register-transfer level estimation techniques for switching activity and power consumption," in *Proc. Int. Conf. Computer-Aided Design*, pp. 158-165, Nov. 1996.

[20] I.-C. Park and C.-M. Kyung, "FAMOS: An efficient scheduling algorithm for high-level synthesis," *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 1437-1448, Oct. 1993.

[21] U. Narayanan, H. W. Leong, K.-S. Chung, and C. L. Liu, "Low power multiplexer decomposition," in *Proc. Int. Symp. Low Power Elec. Design*, pp. 269-274, Aug. 1997.

[22] P. R. Panda and N. D. Dutt, "1995 high-level synthesis design repository," in *Proc. Int. Symp. System Synthesis*, pp. 170-174, Sept. 1995.

[23] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 661-679, June 1989.