An Iterative Improvement Algorithm for Low Power Data Path Synthesis *

Anand Raghunathan and **Niraj K. Jha**

Department of Electrical Engineering
Princeton University, Princeton, NJ 08544

ABSTRACT

We address the problem of minimizing power consumption in behavioral synthesis of data-dominated circuits. The complex nature of power as a cost function implies that the effects of several behavioral synthesis tasks like module selection, clock selection, scheduling, and resource sharing on supply voltage and switched capacitance need to be considered simultaneously to fully derive the benefits of design space exploration at the behavior level. Recent work has established the importance of behavioral synthesis in low power VLSI *design. However, most of the algorithms that have been proposed separate these tasks and perform them sequentially, and are hence not able to explore the tradeoffs possible due to their interaction. We present an efficient algorithm for performing scheduling, clock selection, module selection, and resource allocation and assignment simultaneously with an aim of reducing the power consumption in the synthesized data path. The algorithm, which is based on an iterative improvement strategy, is capable of escaping local minima in its search for a low power solution. The algorithm considers diverse module libraries and complex scheduling constructs such as multicycling, chaining, and structural pipelining. We describe supply voltage and clock pruning strategies that significantly improve the efficiency of our algorithm by cutting down on the computational effort involved in exploring candidate supply voltages and clock periods that are unlikely to lead to the best solution. Experimental results are reported to demonstrate the effectiveness of the algorithm. Our techniques can be combined with other known methods of behavioral power optimization like data path replication and transformations, to result in a complete data path synthesis system for low power applications.*

1. INTRODUCTION

Low power consumption has been established as an important metric for VLSI design. Recent work [1, 2, 3] has shown that the most savings in power consumption are often obtained at the higher levels of the design hierarchy. In this paper, we concentrate on the behavioral synthesis process, that takes as its input the behavioral description of a design, and produces a register-transfer level (RTL) circuit that implements the specified behavior. Behavioral synthesis can be sub-divided into several tasks including module selection, clock selection, scheduling, allocation and assignment. It is important to note that these tasks interact, and solving each one separately is likely to compromise the quality of the design.

Pioneering work in architectural power optimization was presented in [1], which used data path replication and pipelining to enable supply voltage scaling for power reduction. A methodology that used a variety of architectural transformations to reduce power consumption was presented in [2]. Module selection [4], allocation and assignment [5, 6] methods have also been proposed to reduce power consumption. While all the above methods perform some subset of the behavioral synthesis tasks to reduce power consumption by reducing the supply voltage or reducing the switched capacitance, few explore the tradeoffs involved in considering the interaction of the various tasks.

*Acknowledgments: This work was supported by NSF under Grant No. MIP-9319269.

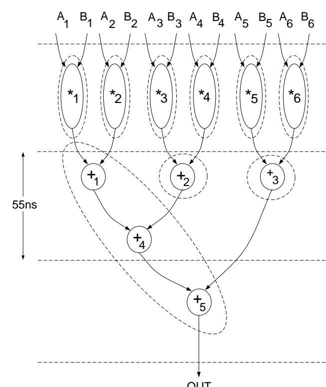


Figure 1: CDFG for dot product and a schedule/assignment

In this paper, we present an iterative improvement algorithm for low power data path synthesis that performs scheduling, clock selection, module selection, and resource allocation and assignment with an objective of minimizing power consumption. A key feature of our algorithm is that it performs these tasks simultaneously, making it possible to *explore the tradeoffs* that result from the *interdependence* of these tasks.

2. BACKGROUND

We consider behavioral descriptions that have been compiled into a *control-data flow graph* (CDFG), which is a directed graph whose vertices consist of arithmetic, logical and comparison operations, delay operators, and special branch, merge, loop entry, and loop exit vertices that represent control flow constructs. The CDFG contains data (control) flow edges that represent data (control) dependencies between operations. An example CDFG shown in Figure 1 represents the computation of the dot product of two vectors.

In this work, we consider data-dominated behavioral descriptions, as are common in digital signal and image processing applications. Two important characteristics of such descriptions are: (i) they consist mainly of arithmetic operations like addition, multiplication, and delay operators, *etc.*, and (ii) there is a constraint on the input sampling period, *i.e.*, the inputs arrive at a fixed rate. It is necessary to be able to process an input sample before the next one arrives. However, it does not pay to process input samples any faster than the required rate [1].

The average switching power, which accounts for the dominant part of power consumption in CMOS technology, of a gate is given by $\frac{1}{2} C_L {V_{dd}}^2 \frac{N}{T}$, where C_L is the gate output capacitance, V_{dd} is the supply voltage, and N is the number of transitions at the gate output during the period of operation T. The equation for power consumption implies that the supply voltage, V_{dd}, has a strong effect on power consumption due to its quadratic contribution. An unfortunate side-effect of decreasing V_{dd}, however, is that the delay of the circuit increases. The delay of a CMOS gate can be shown to be $k \frac{C_L V_{dd}}{(V_{dd} - V_{th})^2}$, where V_{th} is the device threshold voltage, and k is a constant that depends on the technology and the size of transistors

in the gate [7]. Hence, V_{dd} scaling is only performed when the delay degradation does not cause the delay to exceed the specified constraint, or when other means are used to combat the delay degradation. The product of the physical capacitance, C_L, and the transition activity, N, is called the *switched capacitance*. The effect of the switched capacitance term, though not as drastic as the supply voltage term, can also be used to reduce power consumption.

2.1 Scheduling

The process of scheduling determines the cycle-by-cycle behavior of the CDFG, *i.e.*, it assigns each operation in the CDFG to one or more cycles or *control steps*. Figure 1 shows the schedule information for the example CDFG. The horizontal dotted lines labeled with numbers indicate the clock edges, *i.e.*, the boundaries between control steps. Note that $+_4$ is scheduled to be executed in the same control step as $+_1$ and $+_2$ because the clock period, which is $55ns$, is large enough to permit us to do so. This technique is called *chaining*. The term *multicycling* refers to the complementary situation where a single operation requires multiple control steps to execute. *Structural pipelining* refers to the use of pipelined execution units in the data path. Clearly, the choice of clock period affects the assignment of control steps to operations, as does the delay of each operation in the CDFG. These values are determined by the clock selection and module selection tasks, respectively, creating an interdependence among scheduling, module selection, and clock selection. Operations (variables) that are active in the same control step must be assigned to different functional units (registers). For example, operations $*_1$ through $*_6$ must all be performed by separate functional units. Since scheduling affects the rate at which input samples are processed, it also affects the possibilities for reducing V_{dd}. On the other hand, scheduling affects switched capacitance because it imposes constraints on the possibilities of resource sharing. The slack, if any, between the sample period constraint and the time taken by an implementation for processing input samples has been commonly exploited to reduce power consumption using V_{dd} scaling as illustrated below.

Example 1: Suppose the given sample period constraint for the example CDFG shown in Figure 1 is $200ns$. The clock period for the schedule shown in Figure 1 is $55ns$. Since the schedule has three control steps, processing each input sample requires $165ns$. Suppose the clock period was chosen based on delay numbers for $V_{dd} = 5V$. All multiplications in the CDFG of Figure 1 are assumed to be performed by functional unit instances of the template, *array_multiplier*, whereas all additions are assumed to be performed by functional unit instances of the type *ripple_carry_adder*. Dotted lines have been used to group operations that are performed by the same functional unit. Since the given schedule processes input samples faster than required, this surplus performance is exploited to reduce V_{dd} until the time required for one iteration of the CDFG becomes $200ns$, *i.e.*, the sample period constraint is just met. This is determined using a curve or equation that models the V_{dd}-delay relationship [2, 7]. In this case, it is possible to reduce V_{dd} to $4.0V$ and still meet the $200ns$ constraint. ∎

The extent of the slack that is available depends on the constraints imposed by the environment, as illustrated by the next example.

Example 2: Let us consider an image that has 288×360 pixels as per the CIF standard [8]. Consider the task of performing a discrete cosine transform (DCT) on the luminance information of each pixel. A commonly used approach is to divide up the image into blocks, say of 8×8 pixels, and perform a DCT on each block separately. Each 8×8 block thus obtained now requires a two-dimensional DCT, which can be further broken down into 16 one-dimensional 8-point DCT operations. The number of one-dimensional 8-point DCTs required to process one frame is thus calculated to be 25920. Thus, in order to process 30 $frames/sec$, we would need to perform each DCT in about $1286ns$. ∎

We define the term *laxity factor* of a data path that implements a given CDFG to be the ratio of the given sample period constraint

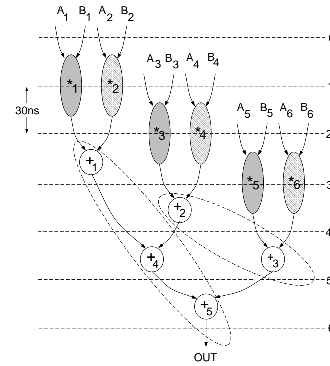


Figure 2: Dot product: Alternate schedule/assignment

to the actual execution time of the data path for one iteration. For the above example, if we assume that an implementation of an 8-point one-dimensional DCT takes $500ns$ to process each set of inputs, then the laxity factor available is 2.57. A higher laxity factor permits us to perform more V_{dd} scaling and hence results in greater power savings.

It is possible to use various methods to speed up the execution of the CDFG, and make use of the slack thus obtained to scale V_{dd} till the sample period constraint is just met. A consequence of these speedup techniques is that the switched capacitance per execution of the CDFG typically increases. This may be due to the use of faster functional units that contribute a higher switched capacitance per operation, or due to the constraints imposed by the tighter schedule on the possibilities for resource sharing. Thus, there exists a V_{dd} *vs* switched capacitance tradeoff that is illustrated by the example below.

Example 3: The CDFG of Figure 1, with a different schedule, clock selection, module selection, and resource assignment is shown in Figure 2, where multiplications are performed by 2-stage pipelined multipliers and the schedule is elongated in order to reduce the number of required multipliers to two. The multiplication operations have been shaded differently to indicate the multiplier that each is assigned to. Two functional units of type *ripple_carry_adder* are used to perform the additions. The clock period is changed to $30ns$ in order to match the cycle time of the pipelined multiplier. Since the schedule has been extended in order to use fewer functional units, processing each input sample now requires $180ns$. As a result, V_{dd} can only be scaled to $4.5V$. In terms of V_{dd}, the architecture implied by Figure 1 is better. In order to compare the actual power dissipation, however, switched capacitance for the two architectures was also measured as explained below. Layouts were first generated for the two candidate architectures, netlists annotated with resistances and capacitances were then extracted from the layouts, and a switch-level simulator was used to simulate the two netlists for the same input sequence (details of our experimental methodology are given in Section 4). The switched capacitance per sample period obtained for the implementations of Figures 1 and 2 were $2912.9pf$ and $2100.6pf$, respectively. From the switched capacitance and V_{dd} numbers, the energy per sample period was calculated to be $23303pJ$ and $21269pJ$, respectively (the power dissipation can be obtained by dividing these numbers by the sample period of $200ns$). Therefore, the architecture derived from Figure 2 has a lower power consumption than the one derived from Figure 1. ∎

Thus, it is important to consider the effects of the different behavioral synthesis tasks on both V_{dd} and switched capacitance in order to truly minimize power consumption.

2.2 Module Selection

Module selection refers to the process of selecting, for each operation in the CDFG, the type of functional unit that will perform it. In order to fully explore the design space, it is necessary to have a diverse library of functional

unit templates where multiple templates exist that are capable of performing each operation (*e.g.* *ripple_carry_adder*, *carry_lookahead_adder*, *carry_select_adder* for addition, *array_multiplier*, *wallace_tree_multiplier*, *pipelined_multiplier* for multiplication, *etc.*).

It is possible to perform area, delay, and power tradeoffs using module selection. The faster modules that perform an operation are typically more expensive in terms of area and switched capacitance. However, using faster modules can result in a faster execution time for the CDFG, thus enabling V_{dd} scaling [4]. Module selection interacts with clock selection, scheduling, and resource sharing. In the example of Figure 1, a clock period of $55ns$ was chosen based on the delay of the multiplication operations, that were assigned to library template, *array_multiplier*. In Figure 2, since the module selection was changed, the clock period was also changed to $30ns$, based on the cycle time of the template, *two_stage_pipelined_multiplier*. Operations that have been assigned to different functional unit templates during module selection cannot share the same resource. This situation is referred to as a *type conflict*. Our algorithm considers the effect of these interactions while synthesizing the data path.

2.3 Clock Selection

Clock selection refers to the process of choosing a suitable clock period for the controller/data path circuit. Given the clock period, T_{clk}, we can divide the execution time of the CDFG, which is equal to the input sample period, T_{sample}, into a number of control steps equal to $\left\lfloor \frac{T_{sample}}{T_{clk}} \right\rfloor$, where $\lfloor x \rfloor$ denotes the largest integer smaller than or equal to x. The choice of the clock period is known to have a significant effect on both area and performance [9]. However, its impact on power consumption was pointed out only recently in [3]. Once a clock period is chosen, we can calculate the delay of each functional unit template in the library in terms of control steps. Since this calculation involves the upward rounding of a fraction, a *slack* is introduced between the time at which a functional unit finishes executing and the clock edge at which its output is actually used. For example, for the CDFG of Figure 2, the clock period is $30ns$. Assuming each addition operation requires $25ns$, including estimates for register, multiplexer and interconnect delays, a slack of $5ns$ is introduced at every addition operation.

The slack introduced due to the clock granularity can result in less-than-complete utilization of the functional units, and could also result in an increase in the time required for the execution of the CDFG. In the context of minimizing power dissipation, slacks can cause two undesirable effects. First, it may not be possible to meet the sample period constraint for the CDFG for some values of V_{dd}. Second, slacks can result in a data path with a higher switched capacitance (this can happen either because faster functional units were used in order to meet the sample period, or because resource sharing was inhibited due to the increased life times of operations in the CDFG). Thus, reducing slacks is beneficial even from the power consumption point of view. It might at first appear that since slacks are caused by a granularity in the clock period, having a very small clock period would minimize the slack and is hence advantageous. However, having a very small clock period tends to significantly increase the switched capacitance in the data path registers (since they are clocked a greater number of times per execution period), the clock distribution network (since it needs to be switched a greater number of times), and the controller (since the number of states in the controller increases with the number of control steps). Due to these complicating factors, methods that solely target slack minimization are not directly applicable when minimizing power consumption is the objective. For reducing power consumption, slacks need to be minimized without choosing too small a clock period.

2.4 Resource Sharing

Resource sharing refers to the use of the same hardware resource (functional unit or register) to perform different operations or store more than one variable. The behavioral synthesis tasks that perform resource sharing are hardware allocation and assignment. These processes decide how many resources of each type to use and which operations or variables to assign to each unit, respectively. Resource sharing significantly affects both the physical capacitance and switching activity in the data path. Heavy resource sharing tends to reduce the physical capacitance, but increase the average switching activity in the data path. Sparsely shared architectures have lower average switching activity, but higher physical capacitance. A detailed analysis of the effect of resource sharing on switched capacitance taking signal statistics into account was described in [6]. We use a similar model here, which we briefly describe below for the sake of completeness.

The functional units in the library are assumed to have some model for switched capacitance, so that, given a pair of input vectors, the capacitance switched in the functional unit on the application of the given input vector pair can be calculated. This model is abstracted into a procedure, SW_CAP(), which can be implemented using either a stochastic power analysis model [10], or instead, could invoke a gate- or switch-level simulator on an appropriate netlist if one is available for the given module, to return the exact capacitance switched. A functional simulation of the CDFG is performed, with an input sequence that is either provided by the user or generated based on known input characteristics. As the functional simulation is performed, a data structure called the *switched capacitance matrix* is updated using the values taken by variables in the CDFG and procedure SW_CAP(). Switched capacitance matrices associate a switched capacitance cost to each pair of operations that could be mapped to the same resource. A separate switched capacitance matrix is created for each functional unit template that exists in the library. For example, consider the functional unit template, *ripple_carry_adder*, and addition operations, $+_i$ and $+_j$. If $+_i$ and $+_j$ share the same ripple carry adder in such a way that the adder performs $+_i$ immediately followed by $+_j$, the operands to $+_i$ and $+_j$ effectively form an input vector pair to the adder. This input vector pair is used to update the switched capacitance matrix entry.

At the end of this data collection process, we have a switched capacitance matrix for each functional unit template t, having entries that indicate, for each pair of operations, the cost in terms of switched capacitance if the operations are both mapped to the same instance of template t. Similarly, switched capacitance matrices are also used to estimate switched capacitance in registers and interconnection units. Different candidate architectures can be evaluated with respect to their switched capacitance [6] by using the entries in the switched capacitance matrices.

3. ALGORITHM

As the discussions in the previous sections have shown, scheduling, clock selection, module selection, and resource sharing interact in a complex way to determine the power consumption of the data path. Since the computational complexity of the power minimization problem forbids an exact or optimal solution, we have developed an efficient heuristic method for performing the above tasks for minimizing power consumption. Our method targets both V_{dd} scaling and switched capacitance reduction.

The pseudo-code in Figure 3 gives an overview of our method, called SCALP. First, the procedure ESTIMATE_MIN_VOLTAGE() is called to estimate the minimum voltage, V_{min}, at which the given CDFG can be implemented. The voltage interval between V_{min} and V_{max} (5V) is discretized in steps of a suitable increment, ΔV, that could be specified by the user as a parameter. The techniques explained in Section 3.1 are used to prune the V_{dd} space significantly. For supply voltages that cannot be pruned, we move on to examine various values for the number of control steps, $csteps$ (or equivalently, various values for the system clock period). Again, it turns out that several candidate clock periods can be easily pruned, using the method explained in Section 3.2. For those combinations of V_{dd} and $csteps$ that cannot be pruned, an initial implementation is generated that satisfies the sample period constraint, which is then improved by calling procedure ITERATIVE_IMPROVEMENT. Since we are now attempting data path synthesis for a fixed value of V_{dd} and $csteps$, the objective is to synthesize a data path that

```
SCALP (CDFG G, Sample Period T_s, Library L) {
   V_min ← ESTIMATE_MIN_VOLTAGE(G, T_s, L);
   V_max ← 5Volts;
   Best_DP ← Φ;
   Cur_DP ← Φ;
   for (V_dd ← V_min; V_dd ≤ V_max; V_dd ← V_dd + ΔV) {
      if (VDD_PRUNE(G, Cur_DP, V_dd) ) {
         continue; (*move on to next V_dd*)
      }
      for (csteps ← MAX_CSTEPS;
            csteps ≥ MIN_CSTEPS; csteps ← csteps − 1) {
         if (CLK_PRUNE(G, L, csteps)) {
            continue; (*move on to next clock*)
         }
         (*minimum switched capacitance data path*)
         (*for the current V_dd and csteps*)
         Cur_DP ← INITIAL_SOLUTION(G, L, V_dd, csteps);
         ITERATIVE_IMPROVEMENT(G, L, Cur_DP);
         if (POWER_EST(Cur_DP) < POWER_EST(Best_DP)) {
            Best_DP ← Cur_DP;
         }
      }
   }
}
```

Figure 3: Overview of our low power synthesis method

satisfies the sample period constraint at the current V_{dd} and clock period, and has minimal switched capacitance. At any time, the best solution ($Best_DP$) seen thus far is stored. After all the candidate supply voltages and clock periods have been either pruned or explored, $Best_DP$ contains the final solution.

3.1 Supply Voltage Pruning

The purpose of V_{dd} pruning is to identify candidate supply voltages that will *not* lead to a data path with the lowest power. Our V_{dd} pruning method is based on obtaining a lower bound on the switched capacitance for the current V_{dd}. A module selection is performed by mapping each operation in the CDFG to the functional unit template that has the lowest switched capacitance (this is determined using switched capacitance matrices [6]). Even though such an implementation may violate the sample period constraint, we ignore this fact since we need a pessimistic estimate. A parallel architecture (no sharing of functional units or registers) is then chosen to implement the data path. A parallel architecture is typically close to the lowest switched capacitance architecture due to the high temporal correlations of signals characteristic of the digital signal and image processing domains [6]. The switched capacitance of this implementation, multiplied by a pessimism factor, λ ($0 \le \lambda \le 1$), is used to lower bound the power consumed by a data path at the current V_{dd} (for our experiments, $\lambda = 0.8$ was used based on our experience in [6]). If the bound thus calculated is greater than the best solution seen, then the current V_{dd} can be pruned.

3.2 Clock Period Pruning

We use the following observation to prune the clock period space: *Given a desired sampling period T_s, it is sufficient to consider those clock periods T_{clk} that satisfy $T_{clk} * i = T_s$ for some integer i* (any other clock period would result in some part of T_s being unused). The practical lower bound on the clock period coupled with this observation itself restricts the set of candidate clock periods to a very limited set. This set can be further pruned as follows. Consider two candidate clock periods, T_{clk1} and T_{clk2}, such that $T_{clk1} < T_{clk2}$. For each functional unit template t in the data path library, let $rrdelay_t$ represent its register-to-register transfer delay. If

$$\left\lceil \frac{rrdelay_t}{T_{clk1}} \right\rceil = \left\lceil \frac{rrdelay_t}{T_{clk2}} \right\rceil \quad \forall \; templates \; t \tag{1}$$

```
Procedure ITERATIVE_IMPROVEMENT(CDFG G, Library L,
                                 Datapath DP) {
   do {
      for (i = 1; i ≤ MAX_MOVES; i = i + 1) {
         Gain_i ← GENERATE_MOVE(G, L, DP);
         Append Gain_i to Gain_List;
      }
      Find subsequence, Gain_1 ... Gain_k in Gain_List so
      that G = Σ_{i=1}^{k} Gain_i is maximized;
      if (G > 0) {
         Accept moves 1 ... k;
      }
   } until(G ≤ 0);
}
```

Figure 4: Procedure ITERATIVE_IMPROVEMENT()

holds, then it is sufficient to consider only T_{clk1} while searching for the minimum switched capacitance data path at the current V_{dd} (because any data path synthesized to operate at T_{clk2} will also operate at T_{clk1}, whereas T_{clk1} could allow us to synthesize data paths that would not satisfy the sample period constraint at T_{clk2}).

If operation chaining is employed with a maximum chaining factor of k (*i.e.* at most k operations can be chained together in a clock cycle), the condition of Equation (1) is checked not just for all functional unit templates in the library, but also for all chained combinations of upto k functional unit templates (note that the delay of chained configurations can be significantly less than the sum of delays of the chained components and should be measured separately for the various chained configurations possible).

3.3 Iterative Improvement Algorithm

Our iterative improvement procedure is based on a general search strategy for optimization problems called *variable depth search* [11]. Given an initial solution, we attempt to find a *sequence* of incremental moves (rather than a single move, as in the case of local search) that maximizes the *cumulative* improvement in the solution, also called the *gain*. This process is iterated until no such sequence can be found. Since we consider sequences that have a cumulative positive gain even though individual moves may have a negative gain, this class of algorithms is capable of hill-climbing to escape from local minima. At any point, the next move we make is chosen based on the steepest descent heuristic [11]. Figure 4 shows the pseudo-code for procedure ITERATIVE_IMPROVEMENT(). The cost function we use is the switched capacitance in the data path, that is estimated using switched capacitance matrices [6]. We define moves so as to explore the scheduling, module selection, allocation, and assignment choices available.

3.4 Moves used in the Iterative Improvement Procedure

The following key observation allows us to restrict the number of distinct types of moves we need to consider: *Moves that affect the schedule alone (called re-scheduling moves), without causing any change in the module selection or resource sharing cannot by themselves affect the switched capacitance in the data path.* However, such moves cannot be completely eliminated since they enable the application of other moves that change the module selection or resource sharing. Hence, we integrate the enabling re-scheduling moves with other moves that they enable. Thus, each composite move consists of a change in the module selection or resource sharing, preceded by an enabling re-scheduling move, if necessary.

Moves of class $\mathcal{A}$: Module selection with re-scheduling. Moves of class $\mathcal{A}$ transform the data path by replacing a functional unit fu_1 that is an instance of a library template, t_1, with another functional unit fu_2 that is an instance of a different library template t_2 (*e.g.*, an adder that is an instance of *carry_lookahead_adder* may be replaced with an instance of *ripple_carry_adder*). Note that a

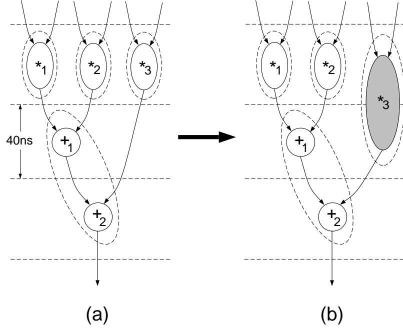


Figure 5: A move of class $\mathcal{A}$

re-scheduling might be needed because the delay of fu_2 (in terms of number of control steps) could in general be greater than that of fu_1. The re-scheduling is performed as follows. We process operations op_1, op_2, ..., op_n that were performed by fu_1, in that order, in the original schedule. For op_i, we first increment the death time of op_i to reflect the delay of fu_2. A breadth-first traversal of the CDFG is then performed starting at op_i, to update the scheduling information of the operations that are in the transitive fanout of op_i. After the above process has been performed for op_n, if all operations in the CDFG complete before the sample period, we are done. Otherwise (if the sample period constraint is violated), the move is not considered. It is easy to extend this method to allow the sample period constraint to be violated by intermediate solutions, provided the final solution of the iterative improvement phase meets it.

Example 4 : Consider the CDFG shown in Figure 5(a). All multiplications are performed by instances of the library template *wallace_tree_multiplier*, whereas all additions are performed by instances of *ripple_carry_adder*. A move of class $\mathcal{A}$ can be applied to result in a modified data path as indicated by Figure 5(b). In the modified data path, multiplication operation $*_3$ is performed by an *array_multiplier*, which requires two control steps. Since $*_3$ has a mobility of one control step, the total number of control steps in the schedule remains the same. ∎

Moves of class $\mathcal{A}$ directly help lower the switched capacitance when faster functional units that typically cause a large amount of switched capacitance are replaced by slower functional units that have a lower switched capacitance. Moves of class $\mathcal{A}$ can also help to indirectly lower switched capacitance when they are used to enable other moves, including those that perform resource sharing.

Moves of class $\mathcal{B}$: Resource sharing/splitting with re-scheduling. The purpose of moves of class $\mathcal{B}$ is to explore the resource sharing choices available. A move of class $\mathcal{B}$ can perform resource sharing by merging two functional units fu_1 and fu_2 into a single functional unit fu (fu performs the operations performed by fu_1 as well as fu_2). For such a move to be valid, fu_1 and fu_2 must be instances of the same library template. Moreover, no operation performed by fu_1 should have an overlapping lifetime with an operation performed by fu_2. If the second condition is not met, we attempt to find a re-scheduling using a method similar to the method described for moves of class $\mathcal{A}$.

Example 5 : Consider the CDFG shown in Figure 6(a). Each multiplication operation is performed by a separate multiplier, the two addition operations are mapped to one functional unit, and each variable is stored in a separate register. One possible move of class $\mathcal{B}$ can be applied to result in a new data path as indicated by Figure 6(b). Operation $*_3$ had to be re-scheduled from the first control step to the second control step in order to enable resource sharing. It is important to note that this move causes two additional multiplexers to be added at the inputs of the multiplier that performs operations $*_2$ and $*_3$ since it now has to select from different sources in the first and second control steps. Hence, switched capacitance estimates for these multiplexers must be taken into account while calculating the gain for this move. ∎

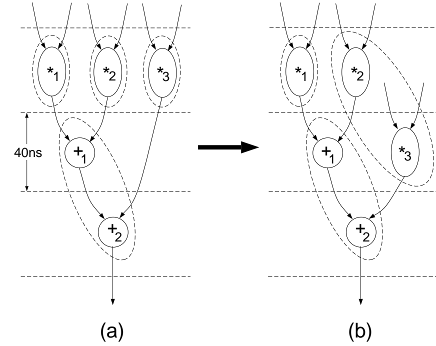


Figure 6: A move of class $\mathcal{B}$ that performs resource sharing

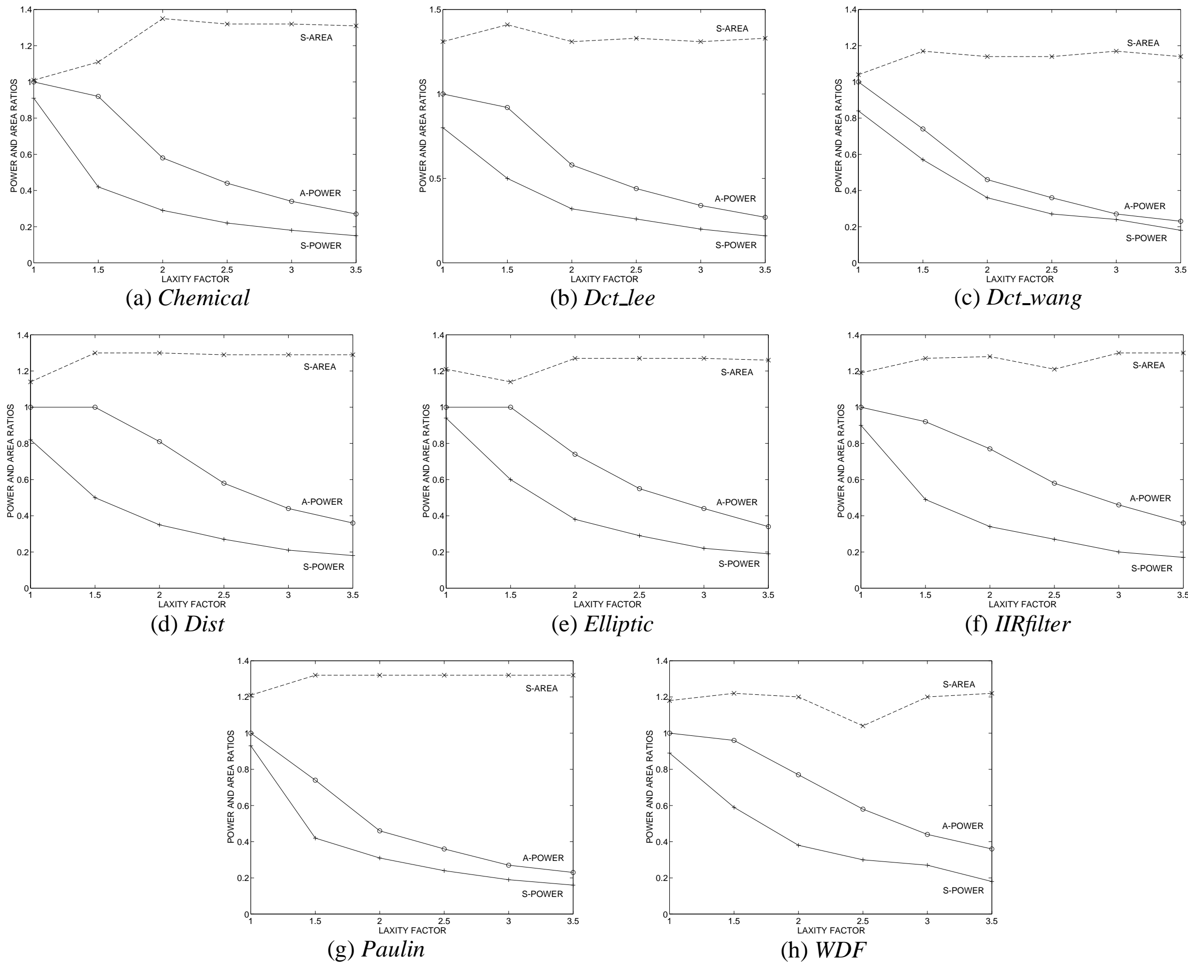
We also provide for moves of class $\mathcal{B}$ that perform resource *splitting*, *i.e.*, replace a single functional unit fu with two functional units, $fu1$ and $fu2$. Moves of class $\mathcal{B}$ that perform splitting do not require any re-scheduling transformations. Apart from potentially reducing switched capacitance, such moves also open up avenues for applying moves of class $\mathcal{A}$, or other resource sharing moves of class $\mathcal{B}$ that were not previously possible.

4. EXPERIMENTAL RESULTS

The methods described in this paper were implemented in the C++ programming language in a program called SCALP. SCALP was evaluated by using it to synthesize several benchmarks from the digital signal and image processing domains. The input to SCALP is a CDFG and a desired sample period. The output of SCALP is a data path and controller that together implement the behavior specified by the CDFG. A logic-level netlist is then obtained for the combined controller/data path, and mapped to the MSU standard cell library using the logic synthesis tool, SIS. Following this, standard cell layout and routing tools from the *Octtools* suite are used to obtain the layout of the combined controller/data path. A switch-level simulator, IRSIM-CAP, that records the switched capacitance during a simulation run is then used to perform a simulation on a switch-level netlist that is extracted from the layout and annotated with resistances and capacitances using MAGIC. Since we obtain a complete layout, several factors that are difficult to estimate at higher levels, like interconnect power, clock network power, controller power, glitching power, *etc.* are taken into account in our power measurements. Input sequences used for measuring power consumption using IRSIM-CAP are generated by first creating a sequence corresponding to a zero-mean, unit-variance Gaussian distribution and passing it through an autoregressive filter to introduce a temporal correlation [6]. The autocorrelation factor used for our experiments is 0.2.

We synthesized each benchmark corresponding to various possible values for the laxity factor ranging from 1.0 (which corresponds to a scenario where there is no laxity at all, *i.e.*, the sample period constraint is just satisfied by the fastest design at $5V$) to 3.5. For each benchmark and laxity factor we synthesized (i) power-optimized architectures generated by SCALP, and (ii) area-optimized architectures with V_{dd} scaling. Area optimization was performed by using a method similar to SCALP, but with area used as the cost function and the metric for evaluating moves. The supply voltage of each area-optimized architecture was then scaled as much as possible subject to the sample period constraint. Figures 7(a) through 7(h) provide plots of (i) the power consumed by the controller/data path circuits produced by SCALP (see curves marked S-POWER) and the area-optimized architectures after V_{dd} scaling (see curve marked A-POWER), and (ii) the area (obtained after layout) of the controller/data path circuits synthesized by SCALP (see curves marked S-AREA). All numbers were normalized with respect to an area-optimized architecture at a supply voltage of $5V$ (we refer to these numbers as the base case). The variable on the x-axis is the laxity factor.

The *Dct* examples perform the Discrete Cosine Transform using different algorithms and are named after their inventors. *Wdf* is



(a) *Chemical* (b) *Dct_lee* (c) *Dct_wang*

(d) *Dist* (e) *Elliptic* (f) *IIRfilter*

(g) *Paulin* (h) *WDF*

Figure 7: Experimental results: power and area ratios (normalized to area-optimized base) for various laxity factors

an FIR wave digital filter. *Elliptic* is the fifth order elliptic wave filter. *Dist*, *Chemical* and *IIRfilter* are different IIR filters used in the industry. The CDFG of *Paulin* has been borrowed from the literature. The area curves are plotted in dotted lines, while the power curves are plotted in solid lines. The curves show that (i) circuits synthesized by SCALP require upto 7 times less power than corresponding area-optimized circuits that operate at $5V$, (ii) SCALP circuits consume upto 2.3 times less power than area-optimized circuits that use V_{dd} scaling after synthesis, and (iii) the area overhead for SCALP synthesized circuits was less than 41% in all the cases. Note that the comparisons are between circuits that were synthesized to meet the same sample period constraints. On the average, SCALP circuits required 1.14, 1.99, 2.95, 3.80, 4.75, and 5.87 times lower power than the base case for laxity factors of 1.0, 1.5, 2.0, 2.5, 3.0, and 3.5 respectively, requiring corresponding area overheads of 16%, 24%, 27%, 24%, 27%, and 27% respectively. The CPU times taken by SCALP were less than 20 minutes in all the cases on a SPARCstation 20 with 128MB of memory. Further power reduction can be achieved by incorporating a suite of architectural transformations [2] and data path duplication [1] into our iterative improvement framework as well.

5. CONCLUSIONS

We presented an efficient iterative-improvement based algorithm to perform scheduling, module selection, clock selection, and hardware allocation and assignment for data-dominated behavioral descriptions in order to minimize power consumption. 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. Since it is possible to easily incorporate other techniques like transformations and data path replication into our framework, current and future work includes incorporating them into a complete data path synthesis tool for low power applications.

REFERENCES

[1] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-power CMOS digital design," *IEEE J. Solid-State Circuits*, pp. 473–484, Apr. 1992.

[2] A. P. Chandrakasan et al., "Optimizing power using transformations," *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 12–31, Jan. 1995.

[3] R. Mehra and J. Rabaey, "Behavioral level power estimation and exploration," in *Proc. Int. Wkshp. Low Power Design*, pp. 197–202, Apr. 1994.

[4] L. Goodby, A. Orailoglu, and P. M. Chau, "Microarchitectural synthesis of performance-constrained, low-power VLSI designs," in *Proc. Int. Conf. Computer Design*, pp. 323–326, Oct. 1994.

[5] A. Raghunathan and N. K. Jha, "Behavioral synthesis for low power," in *Proc. Int. Conf. Computer Design*, pp. 318–322, Oct. 1994.

[6] A. Raghunathan and N. K. Jha, "An ILP formulation for low power based on minimizing switched capacitance during datapath allocation," in *Proc. Int. Symp. Circuits & Systems*, May 1995.

[7] M. Horowitz, T. Indermaur, and R. Gonzalez, "Low-power digital design," in *Proc. Symp. Low Power Electronics*, pp. 8–11, Oct. 1994.

[8] M. Liou, "Overview of the px64 kbit/s video coding standard," *Communications ACM*, vol. 34, Apr. 1991.

[9] S. Chaudhuri, S. A. Blythe, and R. A. Walker, "An exact solution methodology for scheduling in a 3D design space," in *Proc. Int. Symp. System Level Synthesis*, Sept. 1995.

[10] P. Landman and J. M. Rabaey, "Architectural Power Analysis: The Dual Bit Type Method," *IEEE Trans. VLSI Systems*, vol. 3, June 1995.

[11] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, NJ, 1982.