Efficient Switching Activity Computation during High-Level Synthesis of Control-Dominated Designs

A. Bogliolo L. Benini B. Riccó G. De Micheli†

DEIS - Universitá di Bologna
Bologna, ITALY 40136

† CSL - Stanford University
Stanford, CA 94305

Abstract

In this work we propose an exact technique for efficient computation of signal statistics during high-level synthesis for low-power of general control-dominated designs. Our approach does not require iterative simulation: simulation is performed once for all to collect boundary information that will be repeatedly exploited for computing signal statistics for alternative implementations.

1 Introduction

Advanced RTL power estimation techniques rely on activity-based macro-models [2, 3, 4] whose evaluation requires the computation of input-output switching activity and signal probability (hereafter called *boundary statistics*, for brevity) for all leaf cells in a design. Boundary statistics are usually computed by performing RTL simulation, that is the most time-consuming step in the estimation process.

Even though RTL estimators may provide power dissipation estimates early and fast enough for designs specified directly at the register-transfer level, this approach is not fully satisfactory for design styles based on high-level synthesis. High-level synthesis can be seen as the process of automatically generating an optimized (for speed, area, power) RTL description of a design specified at the *behavioral level*. The main steps in high-level synthesis are resource allocation, scheduling and binding [1]. If we want to use a RTL estimator to drive high-level synthesis for low power, we need to follow an iterative process that generates several alternative RTL implementations from a given behavioral specification, estimates power consumption for each one of them, and chooses the best. Clearly, this process is inefficient, because it requires iterated RTL simulation to collect boundary statistics for each implementation.

In this paper we propose a technique for reducing the number of simulations required during high-level synthesis for low power. We perform simulation of the design after scheduling and *before* resource binding. Simulation data are collected in a data structure that allows us to to compute boundary statistics for *every possible binding* without re-simulation. No approximations are made: boundary statistics returned by our computation are exactly the same provided by cycle-accurate RTL simulation performed after binding. This is the main contribution of the paper, that differentiates our approach from previously proposed high-level estimation techniques for control-dominated circuits.

ISLPED99, San Diego, CA, USA

2 Background

The starting point for high-level synthesis is a HDL description compiled into a data structure known as *control-data flow graph* [1]. The nodes in the CDFG represent operations, and the edges represent control and data dependencies. We use the notation adopted in [11]: data dependencies are represented by solid arcs, while control dependencies are represented by dashed arcs. Additional nodes (colored in gray in our representation) are used to model control flow (end of a computation, loop or conditional closing). Examples of CDFGs are shown in Figures 1 (pure data flow graph) and 2 (control-data flow graphs).

We assume a high-level synthesis flow that first performs *resource allocation*, then *scheduling* and finally *resource binding*. During resource allocation, the number and the type of functional macros that can be used to implement the operations in the CDFG is set. Scheduling creates a partial order among operations. The CDFG is levelized and every node is assigned to one level. Levels correspond to clock cycles when operations are executed. Two operations assigned to the same level will be executed concurrently in the final implementation. Finally, resource binding associates every node in a CDFG to a functional macro. Resource binding is sometimes called *sharing* because several operations can share the same macro. After scheduling and sharing, we know when each operation will be executed and by which functional unit, and we can generate a RTL description of the circuit.

One of the main challenges in high-level synthesis for low power is how to compute the cost metric that drives optimization. Some early approaches [5, 6, 7] adopted a simple constant power model, which assumes that every time a new input pattern is supplied to a functional macro $\mathcal{M}$, a constant amount of energy E_M is dissipated. If a functional macro is shared among several operations $Op_1, \ldots, Op_N$, and each operation Op_I executes $N_{eval}(Op_I)$ times, then the total energy consumed by the macro over the set of patterns is $E_{\mathcal{M}}^{TOT} = E_{\mathcal{M}} \sum_{I=1}^{N} N_{eval}(Op_I)$. Since $N_{eval}(Op_I)$ can be computed by simulating the CDFG once for all before high-level synthesis, $E_{\mathcal{M}}^{TOT}$ can be statically computed for any binding alternative. Unfortunately, the energy consumption of a functional macro is not a constant, but it is strongly dependent on its boundary statistics. Several researchers have pointed out the lack of accuracy of the constant power model, and have proposed more accurate power models for functional macros [2, 3, 4]. Pattern-dependent macro-models have the general form $E(\beta)$, where E is a

function determined by characterization, and β is some compact representation of boundary statistics.

Pattern-dependent macro-models are much harder to use during high-level synthesis than the constant model. In fact, resource binding can drastically change boundary statistics, thus making it impossible to compute total power as a weighted sum of constant contributions. One obvious solution to this problem is to perform simulation of the input pattern set for every candidate solution generated by high-level synthesis after binding. Simulation has complexity $O(K \times N_{op})$. Hence, evaluation of many candidate solutions may be too slow if K is large (and it should be large to obtain meaningful boundary statistics). Several techniques have been proposed to bypass iterative simulation [8, 9, 10, 11, 12]. All these techniques are based on the same basic idea: simulation is performed before binding, but transition activities and signal probabilities are stored in a data structure that contains information on how boundary statistics change with different bindings. Whenever a new binding is generated by synthesis, boundary statistics are efficiently extracted from the data structure, without the need of time-consuming re-simulation.

The main limitation of all previously proposed techniques is that they can accurately compute boundary statistics without iterated simulation only for data-dominated designs, with simple conditionals and no loops. For control-dominated designs, boundary statistics computation is either impossible or approximate. The technique described in the next section fully overcomes this limitation, and it allows us to compute *exact* signal and switching probabilities for every possible binding without re-simulation.

3 Computing boundary statistics

Our starting point is a CDFG representation of the behavioral specification after scheduling and before resource binding. We also assume the existence of a pattern set $\mathcal{S}_K$ of K consecutive input vectors, which represent typical input stimuli for the target design. Boundary statistics computation is based on a single simulation of the CDFG with $\mathcal{S}_K$ input vectors. Simulation of the CDFG is performed by propagating input stimuli through the nodes (representing operations), following edges dependencies. During simulation, nodes in the CDFG are processed in a total order which is compatible with the partial order enforced by scheduling. We call "iteration" the process of propagating an input vector though the nodes of a CDFG. Thus, each input vector in $\mathcal{S}_K$ corresponds to one iteration of the CDFG. Iterations are identified by a unique *iteration index* $i = 1, 2, \ldots, K$. An iteration terminates when propagation reaches the sink node of the CDFG. Notice that a new iteration can begin before the previous one is finished, in this case, the CDFG is functionally pipelined. Our task is to collect data during CDFG simulation in such a way that it will be possible to estimate boundary statistics for every functional macro that can be associated to a set of CDFG nodes after resource binding *without repeating the simulation* and *for any legal binding.*

Operations that can be performed by the same *shared* resource (i.e., by the same instance of a functional macro) are said to be *compatible.* For the sake of clarity, we will restrict our analysis to a single class $\mathcal{G}$ of compatible operations with cardinality Γ. Multiple classes are processed independently, using the same technique used for the single class case. For a given pattern set $\mathcal{S}_K$, we denote by $\beta(Op_1, ..., Op_N)$ the boundary statistics of a shared resource performing N operations Op_1, ..., Op_N in the order they are written ($N \leq \Gamma$).

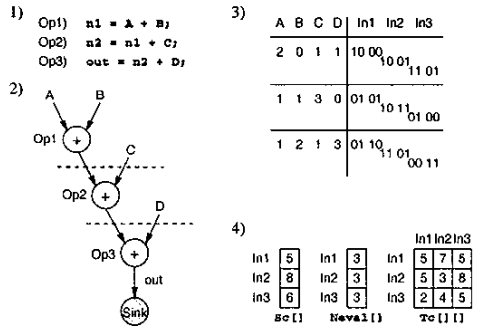


Figure 1: Example of a data flow graph.

We represent boundary statistics by means of average signal and transition probabilities at the inputs and outputs of a resource: *Pin*, *Din*, *Pout* and *Dout*. In this section we present a static approach to compute the exact values of $Pin(Op_1, ..., Op_N)$, $Din(Op_1, ..., Op_N)$, $Pout(Op_1, ..., Op_N)$ and $Dout(Op_1, ..., Op_N)$ for any ordered set of compatible operations. Without loss of generality, we refer only to input statistics *Pin* and *Din*. Output statistics are of the same nature and can be computed by applying (with no modification) the same methods described for input statistics.

3.1 Data-dominated designs

Consider a shared resource $\mathcal{M}$ performing operations Op_1 and Op_2 of the DFG of Fig. 1. We denote by In_1 and In_2 the streams of bit vectors at the inputs of the two operations. Each vector of In_1 is uniquely identified by an apex, called *evaluation index*, that represents its relative position in the stream. Since in a pure data flow graph each operation is evaluated once per iteration, the *evaluation index* is equal to the *iteration index*. Pattern In_1^i is the concatenation of the values of A and B at the i-th iteration, pattern In_2^i is the concatenation of the values of $n1$ and C at the same iteration. During data-path evaluation, patterns from In_1 and In_2 alternate at the inputs of $\mathcal{M}$, generating a new input stream with statistics $Pin(Op_1, Op_2)$ and $Din(Op_1, Op_2)$, that can be directly computed from the data stream during DFG simulation:

$$Pin(Op_1, Op_2) = \frac{1}{K} \sum_{i=1}^{K} \frac{||In_1^i||_H + ||In_2^i||_H}{2BW} \quad (1)$$

$$Din(Op_1, Op_2) = \frac{1}{2K-1} \left(\frac{d_H(In_1^K, In_2^K)}{BW} + \sum_{i=1}^{K-1} \frac{d_H(In_1^i, In_2^i) + d_H(In_2^i, In_1^{i+1})}{2BW} \right) \quad (2)$$

In the equations above, K is the number of primary input patterns, BW is the number of inputs of Op_1 and Op_2, $||x||_H$ is the Hamming measure of x (i.e., the number of ones in x), $d_H(x1, x2)$ is the Hamming distance between $x1$ and $x2$ (i.e., the number of bit differences), and apex i is the evaluation. We observe that Equation 1 can be rewritten as:

$$Pin(Op_1, Op_2) = \frac{Sc(In_1) + Sc(In_2)}{BW(N_{eval}(Op_1) + N_{eval}(Op_2))} \quad (3)$$

where $Sc(In_1)$, hereafter called *signal count* of In_1, is the total number of ones in stream In_1 (in symbols, $Sc(In_1) = \sum_{i=1}^{N_{eval}(Op_1)} ||In_1^i||_H$). Similarly, we can express *Din* as:

$$Din(Op_1, Op_2) = \frac{Tc(In_1, In_2) + Tc(In_2, In_1)}{BW(N_{eval}(Op_1) + N_{eval}(Op_2) - 1)} \quad (4)$$

where $Tc(In_1, In_2)$, hereafter called *toggle count* of $In_1 In_2$, is the total number of input transitions observed when switching from a pattern of In_1 to the subsequent pattern of In_2 ($Tc(In_1, In_2) = \sum_{i=1}^{K} d_H(In_1^i, In_2^i)$).

N_{eval} and Sc are *first-order counts*, since they are properties of single input streams, Tc is a *second-order count*, since it is a property of a pair of input streams. Equations 3 and 4 show that $Pin(Op_1, Op_2)$ and $Din(Op_1, Op_2)$ can be statically computed from the first and second-order counts of In_1 and In_2, that, in their turn, can be computed during simulation.

Example 1 *Fig. 1.3 shows a three-pattern simulation for the DFG of Fig. 1.2. According to the schedule shown in the DFG, simulation of each input pattern requires 3 c-steps. In Figure 1.3, rows represent c-steps and horizontal lines denote iteration boundaries. For the sake of simplicity, operations denoted by + are assumed to be module-4 additions (i.e., the data path has width 2). We want to compute input statistics for a shared resource $\mathcal{M}$ implementing Op_2 and Op_3. From the trace we obtain $Sc(In_2) = 8$, $Sc(In_3) = 6$, $Tc(In_2, In_3) = 8$ and $Tc(In_3, In_2) = 4$. Since all operations have $N_{eval} = K = 3$ and $BW = 4$, from Equations 3 and 4 we obtain: $Pin(Op_2, Op_3) = \frac{8+6}{4(3+3)} = 0.583$ and $Din(Op_2, Op_3) = \frac{8+4}{4(3+3-1)} = 0.6$.*

Equations 3 and 4 can be generalized to compute boundary statistics for any shared resource, based only on first and second order counts:

$$Pin(Op_1, ..., Op_N) = \frac{Sc(In_1)+...+Sc(In_N)}{BW(N_{eval}(Op_1)+...+N_{eval}(Op_N))} \quad (5)$$

$$Din(Op_1, ..., Op_N) = \frac{Tc(In_1,In_2)+...+Tc(In_{N-1},In_N)+Tc(In_N,In_1)}{BW(N_{eval}(Op_1)+...+N_{eval}(Op_N)-1)} \quad (6)$$

The rationale behind the two equations is intuitively clear. *Pin* is computed as the ratio between the total number of ones at the inputs of the shared resource, divided by the total number of bits in the composed input stream. *Din* is the ratio between the total number of switching bits (computed as the sum of pair-wise contributions) and the total number of transitions. Equations 5 and 6 actually provide a constructive proof of the following theorem.

Theorem 1 *Given a scheduled DFG and a workload, boundary statistics can be statically computed for any binding solution using only first-order counts of nodes in the DFG and second-order counts of pairs of compatible operations. First and second-order counts are computed during behavioral simulation of the DFG with the target workload.*

To understand the meaning of Theorem 1, remember that there are Γ compatible operations in class G. The number of possible sharing alternatives is $2^\Gamma - 1$, while the number of first-order counts to be collected during simulation is linear, and the number of second-order counts is quadratic in Γ. We store first-order counts in two integer arrays of size Γ (`Neval[]` and `Sc[]`), and second-order counts in a matrix of integers with size $\Gamma \times \Gamma$ (`Tc[][]`). Values in the data structures are incrementally computed during the behavioral simulation run performed before binding. This is a worst-case $O(\Gamma^2 \times K)$ operation whose complexity is greater than that of a single simulation run $O(\Gamma \times K)$, but is much smaller than that of repeated simulations for each possible binding, which is $O(2^\Gamma \times \Gamma \times K)$. It is also worth noting that data structure updates are usually much faster than simulation steps. Hence, data structures can be easily constructed during simulation.

Example 2 *Data structures* `Neval[]`, `Sc[]` *and* `Tc[][]` *are shown in Fig. 1.4 filled with the values computed from the simulation trace of Fig. 1.3. According to Equations 5 and 6, we use the pre-collected data to compute input statistics for a shared resource implementing all compatible operations Op_1, Op_2 and Op_3.*

$$Pin(Op_1, Op_2, Op_3) = \frac{Sc[1]+Sc[2]+Sc[3]}{BW(Neval[1]+Neval[2]+Neval[3])} = 0.528$$

$$Din(Op_1, Op_2, Op_3) = \frac{Tc[1][2]+Tc[2][3]+Tc[3][1]}{BW(Neval[1]+Neval[2]+Neval[3]-1)} = 0.531$$

Notice that only table lookup's and algebraic computation have been performed to compute the exact input statistics for the shared resource.

3.2 Control-dominated designs

Equation 5 of Section 3.1 is completely general: it can be applied without modification to arbitrary CDFGs to compute signal probabilities (*Pin*) at the inputs of shared resources. On the contrary, Equation 6 relies on the implicit assumption that the input stream of a shared resource can be obtained by interleaving the input streams of the operations it implements, taken in a fixed order. This assumption is always verified for DFGs without control, but it does not hold any longer for arbitrary CDFGs. In fact, if the execution of (some of) the compatible operations to be implemented by the same resource is conditioned to the run-time value of some input or internal signal, the way the input patterns of each operation alternate at the inputs of the shared resource may change dynamically.

Example 3 *Fig. 2.a shows a simple CDFG together with its behavioral specification, simulation trace and counts. Evaluation of Op_3 is conditioned to the values of primary input C and internal signal $n1$: only when $n1 > C$ operation Op_3 is evaluated. This is clearly illustrated by the simulation trace of Fig. 2.a.3: in iteration 2 Op_3 is not evaluated since $n1 = 2$, $C = 3$, $n1 < C$. The input stream for a shared resource $\mathcal{M}$ implementing Op_2 and Op_3 would be $In_2^1, In_3^1, In_2^2, In_3^2, In_3^3$. The missing pattern In_3^2 brakes the regular alternation between In_2 and In_3, giving rise to a direct transition from In_2^2 and In_2^3 at the inputs of $\mathcal{M}$.*

A further example is provided by the CDFG of Fig. 2.b, where Op_2 is within a data-dependent loop. The number of times Op_2 is evaluated at each iteration depends on the primary input values: it is evaluated once in iteration 1, twice in iteration 3, while it is not evaluated at all in iteration 2.

In presence of control statements the evaluation index is no longer equal to the iteration index and different operations may have different (and multiple) evaluation indexes at the same iteration. To deal with control-dominated designs we need, first, to generalize the definition of $Tc(In_1, In_2)$. The toggle count of $In_1 In_2$ is redefined as the sum of the Hamming distance between all pairs (In_1^i, In_2^j) such that: i) In_1^i precedes In_2^j in the simulation trace, and ii) there are no patterns of In_1 and In_2 between In_1^i and In_2^j in the trace.

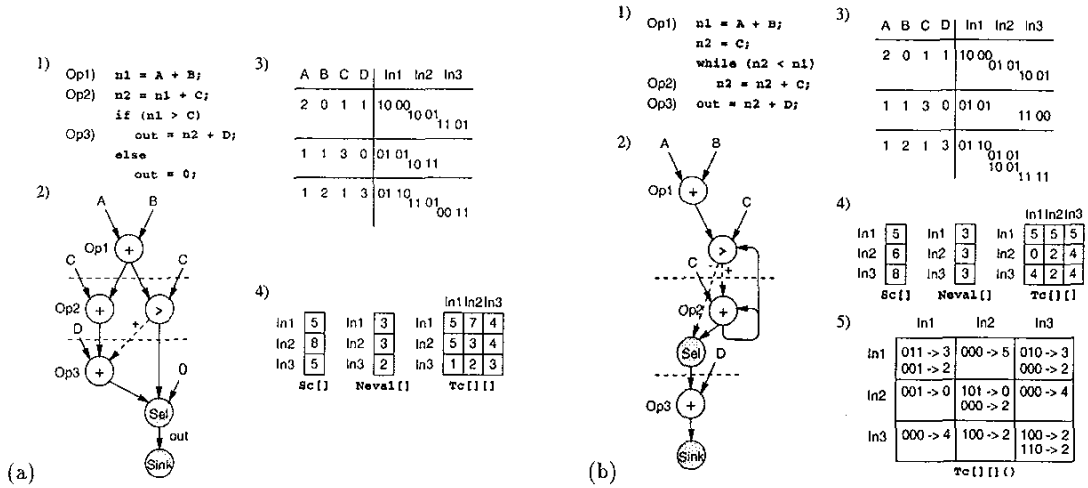


Figure 2: Example of CDFGs. (a) CDFG with a conditional statement. (b) CDFG with a data-dependent loop.

Example 4 *The extended definition of Tc allows us to compute toggle counts for the compatible operations of the CDFG of Fig. 2.a. For instance, $Tc(In_2, In_3)$ is the sum of only two contributions: $d_H(In_2^1, In_3^1)$ and $d_H(In_2^3, In_3^3)$. Similarly, $Tc(In_3, In_2) = d_H(In_3^1, In_2^2)$. Toggle counts for all compatible resource pairs are reported in matrix* Tc[][] *. If we use Equation 4 to compute input statistics for a resource $\mathcal{M}$ implementing operations Op_2 and Op_3, we obtain*

$$Din(Op_2, Op_3) = \frac{Tc[2][3]+Tc[3][2]}{BW(Neval[2]+Neval[3]-1)} = 0.375$$

However, if we compute $Din(Op_2, Op_3)$ directly from the trace we obtain a different result: $8/16 = 0.5 \neq 0.375$. The difference is due to the transition from In_2^2 and In_2^3, whose contribution (2 switching bits) is not accounted for by Equation 4. For a similar reason, 6 fails in computing $Din(Op_1, Op_2, Op_3)$: it provides 0.429 while the actual value is 0.5. The missing transitions are those from In_2^2 and In_1^3.

The reason why Equations 4 and 6 failed in computing input activities for the CDFG of Example 4 is two fold. First, some of the contribution to the overall toggle count do not appear in the equations. This is the case, for instance, of the contribution of $d_H(In_2^2, In_2^3)$ to $Din(Op_2, Op_3)$. Second, matrix Tc[][] does not retain sufficient information about the simulation trace. For instance, the only information about $d_H(In_2^2, In_2^3)$ contained in matrix Tc[][] is the intra-stream toggle count of In_2 (entry [2][2] of matrix Tc) that is the sum of $d_H(In_2^2, In_2^3)$ and $d_H(In_2^1, In_2^2)$. While the first term has to be accounted for when computing $Din(In_2, In_3)$, the second one has not. In fact, In_2^1 and In_2^2 are not consecutive input patterns for the shared resource, due to the presence of the intermediate pattern In_3^1.

To compute the exact value of $Din(Op_2, Op_3)$ we should split the cumulative count $Tc(In_2, In_2)$ into two partial counts: $Tc(In_2, In_2|0)$ and $Tc(In_2, In_2|1)$. $Tc(In_2, In_2|0)$ represents the total Hamming distance between pairs of patterns of In_2 having no patterns of In_3 in between, while $Tc(In_2, In_2|1)$ represents the total Hamming distance between pairs of patterns of In_2 having at least one pattern of In_3 in between. If we assume that such partial counts are available, Din can be computed as:

$$Din(Op_2, Op_3) = \frac{Tc(In_2,In_2|0)+Tc(In_2,In_3)+Tc(In_3,In_2)}{BW(Neval(Op_2)+Neval(Op_3)-1)} = 0.5$$

Though the previous equation has been obtained ad hoc for example 4, it contains all the elements to provide a general solution to the problem of computing boundary statistics for CDFGs. All we need is to split all pair-wise toggle counts to retain some information about intermediate input patterns. To this purpose, we define the *conditional toggle count* of In_1, In_2 subject to condition c (denoted by $Tc(In_1, In_m|c)$) as the partial toggle count $Tc(In_1, In_2)$ restricted to those pairs (In_1^i, In_2^j) satisfying condition c. The condition we are interested in is the presence of input patterns from other streams between In_1^i and In_2^j. For a set of Γ compatible operations, we express condition c as a string of Γ Boolean flags taking value "1" (present), "0" (absent). If the k-th Boolean flag in c has value "1" then the toggle count is restricted to pairs of In_1, In_2 patterns having at least one pattern of In_k in between. If the k-th Boolean flag in c have value "0", then we restrict toggle count to pairs In_1, In_2 patterns that do not have a pattern of In_k between them.

Example 5 *Consider the simulation trace of Figure 2.b.3. The conditional toggle count of In_1, In_3 subject to $c = 010$ is*

$$Tc(In_1, In_3|010) = d_H(In_1^1, In_3^1) + d_H(In_1^3, In_3^3) = 3$$

that is "the total number of transitions between pattern pairs In_1, In_3 that have one or more In_2 patterns in between". Notice that we conventionally assign value 0 to the Boolean flags in c corresponding to In_1 and In_3 (i.e., the streams for which conditional counts are computed).

In the following, we use the shorthand notation "–" (don't care) to compactly represent *sets of conditions*. For instance, condition $c = 0 - 0$ indicates the set of pattern pairs of In_1, In_3 that either have or do not have a pattern of In_2 in between.

In principle, all conditional toggle counts could be stored in a three-dimensional matrix of size $\Gamma \times \Gamma \times \Gamma_c$, where Γ_c is the number of different conditions, i.e., the number of possible configurations of c, that is exponential in Γ. Data structures of exponential size are usually of little practical interest. Fortunately, the matrix is extremely sparse in the third dimension since most conditions are either impossible or they are never verified in the actual simulation trace. What we need is an efficient representation of conditional toggle counts, allowing us to represent and compute only significant values.

To efficiently represent conditional toggle counts we use a $\Gamma \times \Gamma$ matrix of hash tables, `Tc[i][j](c)`. Entry (i,j) is a hash table containing all significant conditional toggle counts of In_i, In_j. Condition c is the hash key. This hybrid representation has three main advantages: (i) we do not need to construct the entire matrix beforehand (the data structure is initialized as a square matrix of empty hash tables); (ii) only significant entries are represented and computed; (iii) conditional toggle counts are incrementally computed during simulation. In practice, `Tc[i][j](c)` contains the minimum amount of information required to compute boundary statistics without repeating simulation. Most important, the construction of `Tc[i][j](c)` has the same worst-case complexity of the construction of the unconditioned matrix used for DFG with no control, namely $O(\Gamma^2 \times K)$.

Example 6 *Figure 2.b.5 shows a schematic representation of a matrix of hash tables containing the conditional toggle counts computed from the simulation trace reported on Figure 2.b.3. Notice that* Tc *has 13 entries that has been computed from a simulation trace consisting of three iterations (9 evaluations). The construction of* Tc *required 27 steps. Now suppose that there are* $\Gamma = 10$ *compatible resources in the CDFG, but only* Op_1, Op_2 *and* Op_3 *are evaluated in the three iterations that represent our workload. Though* Tc *will be a matrix of 100 hash tables, it will still have 13 significant entries computed in 27 steps. Incidentally, we remark that the sum of the entries in each hash table is the unconditioned toggle count of the corresponding pair of streams.*

The last issue to be addressed is how to use conditional toggle counts to compute boundary statistics. This is done by the generalized form of Equation 6, expressing the switching activity for a shared resources implementing N operations from a class G of Γ compatible operations:

$$Din(Op_1,..,Op_N) = \frac{\sum_{i=1}^{N}\sum_{j=i}^{N} Tc(In_i,In_j|c_{ij})}{BW(N_{eval}(Op_1)+...+N_{eval}(Op_N)-1)} \quad (7)$$

where c_{ij} is a string of Γ symbols, one for each compatible operation, whose L-th symbol is "0" if and only if Op_L is one of the N operations implemented by the resource, "$-$" otherwise. The rationale behind this equation is that in CDFG any pair of operations i and j sharing the same resource contributes to its input activity whenever a pattern from In_i and a pattern from In_j are to be evaluated in sequence. This happens if there are no input patterns of other shared operations in between.

Example 7 *Fig. 3.2 shows how to apply Equation 7 to compute the switching activity at the inputs of a shared resource implementing operations* Op_1 *and* Op_2 *of the CDFG of Fig. 2.b. Formulas for computing the input activity of a shared resource implementing all compatible operations are also shown in Fig. 3.2. Equation 7 leads to exactly the same results obtained by simulating the input stream.*

```
addA)  n1 = a + b;
addB)  n2 = c + d;
       n3 = n1 * n2;
addC)  n4 = a + n1;
addD)  out = n4 + d;
```

a)

```
       data_valid = 0;
subA)  n1 = x - y;
subB)  n2 = y - x;
cmpC,D) if ((n1 != x) && (n2 != y))
cmpA)    while (x != y)
cmpB)      if (x < y)
subC)        y = y - x;
           else
subD)        x = x - y;
       out = x;
       data_valid = 1;
```

b)

Figure 4: Behavioral specification of two benchmark designs: (a) simple data-path computing (a+b)(c+d)+a+d, and (b) greatest common divisor (GCD) algorithm.

In summary, we showed that exact switching activities (and signal probabilities) for any resource sharing in general CDFGs can be computed without re-simulation by storing data from a single simulation run in a matrix of hash tables. Data-collection complexity is quadratic in the number of compatible operations Γ and linear in the input pattern set S. Switching activity estimation is performed by simply evaluating Equation 7.

4 Experimental results and conclusions

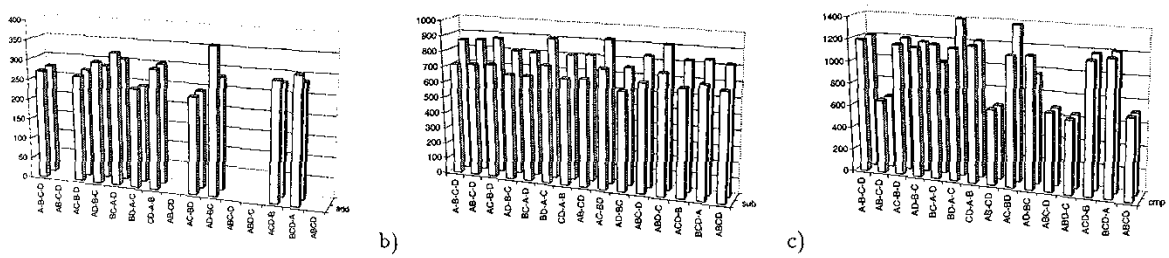
The approach described so far has been implemented in C and linked to *Monet*, a high-level synthesis and design exploration tool by *Mentor Graphics*. A cycle-accurate CDFG simulator has been developed to perform pre-binding simulation. The power estimation flow can be summarized as follows. After scheduling, CDFG simulation is performed only once with a given set of input patterns to construct the matrix of hash tables. The matrix is then exploited to compute signal statistics at the boundaries of the functional units for any alternative binding. Power dissipation in functional units is estimated using pre-characterized LUT power models [4]. In our experiments, we pre-characterized a set of functional macros taken from the *Monet*'s library and synthesized using *Synopsys's Design Compiler*. An accurate gate-level power simulator (namely, PPP [13]) has been used to provide reference power values for characterization and validation. We present results on two case studies: a data-path without control performing arithmetic computations, and the greatest common divisor (GCD) algorithm. Their behavioral specifications are shown in Fig. 4. For each design, allocation and scheduling have been repeatedly performed by *Monet* with different area and timing constraints. CDFGs have been simulated right after scheduling with biased sequences of 1000 input patterns to construct data structures `Neval[]`, `Sc[]` and `Tc[][]()` for each set of compatible operations.

Fig. 5.a shows the results obtained for the design of Fig. 4.a. Bars represent the average energy (in pJ) required to perform the four 8-bit additions in the data path according to a given binding. Labels on the x axis represent sharing alternatives (operations are denoted by A,B,C and D, operations bound to different resources are separated by a minus). Operations A and B were scheduled concurrently, hence, they were not compatible for sharing. The two series of bars represent results obtained with two library elements. The impact of sharing on power consumption is evident: the difference between two sharing alternatives may be greater than 30%. This is only due to the changes induced on boundary statistics. Finally, we remark that the minimum-power binding solution is AC-BD, that is also a minimum-area solution. The energy efficiency of this solution can be explained by observing that, at each iteration, operations A

$$Din(Op_1, Op_2) = \frac{\mathtt{Tc[1][1](00-)} + \mathtt{Tc[1][2](00-)} + \mathtt{Tc[2][1](00-)} + \mathtt{Tc[2][2](00-)}}{BW(\mathtt{Neval[1]} + \mathtt{Neval[2]} - 1)} = 0.45$$

$$Din(Op_1, Op_2, Op_3) = (\mathtt{Tc[1][1](000)} + \mathtt{Tc[1][2](000)} + \mathtt{T[1][3](000)} + \mathtt{Tc[2][1](000)} + \mathtt{Tc[2][2](0-0)} + \mathtt{Tc[2][3](000)}$$
$$\mathtt{Tc[3][1](000)} + \mathtt{Tc[3][2](000)} + \mathtt{Tc[3][3](000)}) \frac{1}{BW(\mathtt{Neval[1]} + \mathtt{Neval[2]} + \mathtt{Neval[3]} - 1)} = 0.531$$

Figure 3: Two examples of switching activity computation at the inputs of a shared adder.



a) b) c)

Figure 5: Average energy per iteration (in pJ) required to perform (a) the four additions in the data path of Fig. 4.a, (b) the four subtractions and (c) the four comparisons in the control/data path of Fig. 4.b.

and C (B and D) have the same left-hand operand, hence, sharing reduces the overall switching activity.

The behavioral specification of the GCD algorithm shows two sets of compatible operations: 4 subtractors and 4 comparators. Each operation was scheduled in a different c-step, thus making all sharing alternatives worth to be explored. Results obtained for subtractors and comparators are separately reported in Figg. 5.b and 5.c. While binding has only a marginal impact on the power consumed to perform subtractions, it may reduce by 50% the power spent in performing comparisons. Once again, this can be explained by looking at the design specification: at each iteration, comparisons A and B have the same inputs. If a shared comparator providing both "$\neq$" and "$<$" output signals is used, the overall cost is significantly reduced. Moreover, comparisons A and B are within the inner loop of the algorithm, that is repeatedly evaluated for each input pattern. That's why all binding solutions using the same macro to perform comparisons A and B achieve 50% power savings. The speed-up obtained by our method with respect to iterated simulation was of two orders of magnitude without accuracy loss. It is also worth noting that the speed-up would grow linearly with the simulation length and exponentially with the number of compatible resources.

4.1 Conclusions

We described a technique for computing input-output statistics during high-level synthesis of control-dominated specifications, without iterative simulations. Simulation is performed only once, after scheduling and before resource binding. Information on signal probabilities and switching activities is collected in a complex data structure. Whenever the high-level optimization algorithm computes a new binding, signal statistics can be estimated by performing a set of lookups in the data structure. Signal probability and switching activity computation based on our technique does not imply any accuracy loss with respect to iterated simulation and it is much more efficient.

Acknowledgments: This work is supported by a grant from Mentor Graphics Corporation (*CoDesign Consortium*).

References

[1] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

[2] P. Landman et al., "Architectural Power Analysis, the Dual Bit Type Method," *IEEE Trans. on VLSI*, vol. 3, no. 2, pp. 173–187, 1995.

[3] A. Raghunathan et al., "Register-Transfer Level Estimation Techniques for Switching Activity and Power Consumption," *ICCAD-96*, pp. 158–165, 1996.

[4] S. Gupta et al., "Power macromodeling for high-level power estimation," *DAC-97*, pp. 365–370, 1997.

[5] A. Chandrakasan et al., "Optimizing Power Using Transformations," *IEEE Trans. on CAD*, vol. 14, no. 1, pp. 12–31, Jan. 1995.

[6] N. Kumar et al., "Profile-Driven Behavioral Synthesis for Low-Power VLSI Systems," *IEEE Design and Test of Computers*, vol. 12, no. 3, pp. 70–84, Fall 1995.

[7] R. San Martin et al., "Optimizing Power in ASIC Behavioral Synthesis," *IEEE Design and Test of Computers*, vol. 13, no. 2, pp. 58–70, Summer 1996.

[8] A. Raghunathan et al., "SCALP: An Iterative-Improvement-Based Low-Power Data Path Synthesis System," *IEEE Trans. on CAD*, vol. 16, no. 11, pp. 1260–1277, Nov. 1997.

[9] J. Chang et al., "Module Assignment for Low Power," *EDAC-96*, pp. 16-376–81, 1996.

[10] J. E. Crenshaw et al., "Accurate High Level Datapath Power Estimation," *EDTC-97*, pp. 590–595, 1997.

[11] K. Khouri et al., "IMPACT, A High-Level Synthesis System for Low Power Control-Flow Intensive Circuits," *DATE-98*, pp. 848–854, 1998.

[12] K. Khouri et al., "Fast High-Level Power Estimation fo Control-Flow Intensive Designs," *ISLPED-98*, pp. 299–304, 1998.

[13] A. Bogliolo et al., "Power Estimation of Cell-Based CMOS Circuits," *DAC-96*, pp. 433 – 438, 1996.