Common-Case Computation: A High-Level Technique for Power and Performance Optimization *

Ganesh Lakshminarayana †, Anand Raghunathan †,
Kamal S. Khouri ‡, Niraj K. Jha‡, and Sujit Dey§

† CCRL-NEC USA, ‡ Dept. of Electrical Engg., Princeton University
§ Dept. of Electrical Engg., Univ. of California, San Diego

Abstract

This paper presents a design methodology, called *common-case computation* (CCC), and new design automation algorithms for optimizing power consumption or performance. The proposed techniques are applicable in conjunction with any high-level design methodology where a structural register-transfer level (RTL) description and its corresponding scheduled behavioral (cycle-accurate functional RTL) description are available. It is a well-known fact that in behavioral descriptions of hardware (also in software), a small set of computations (CCCs) often accounts for most of the computational complexity. However, in hardware implementations (structural RTL or lower level), CCCs and the remaining computations are typically treated alike. This paper shows that identifying and exploiting CCCs during the design process can lead to implementations that are much more efficient in terms of power consumption or performance. We propose a CCC-based high-level design methodology with the following steps: extraction of common-case behaviors and execution conditions from the scheduled description, simplification of the common-case behaviors in a stand-alone manner, synthesis of common-case detection and execution circuits from the common-case behaviors, and composing the original design with the common-case circuits, resulting in a CCC-optimized design. We demonstrate that CCC-optimized designs reduce power consumption by up to 91.5%, or improve performance by up to 76.6% compared to designs derived without special regard for CCCs.

1 Introduction

In this paper, we present a design methodology and new computer-aided design algorithms for optimizing power consumption or performance. Our techniques can be applied to pre-designed RTL circuits, or in conjunction with traditional high-level synthesis optimizations. They exploit the well-known fact that in several applications, a small part of the behavior is likely to dominate the overall computational effort. This paper shows that identifying such frequently occurring, or common-case computations (CCC), and exploiting them appropriately, can lead to large improvements in performance or average power (energy) consumption.

Starting with a cycle-accurate functional RTL or scheduled behavioral description, along with its structural RTL implementation, we present techniques to identify CCCs from the schedule. In an implementation derived without particular attention to the common case, the delay and power expended in executing the CCCs may be significantly higher than necessary due to one or more of the following factors:

- Various synthesis optimizations, which may not be applicable in the context of the complete design, are applicable when only the CCCs are considered. For example, a CCC typically consists of only one or a few (conditional) threads of execution from the original behavior. Thus, a lot of control-flow constructs, which are known to be bottlenecks for various high-level optimizations [1, 2], are eliminated by considering the CCC alone.
- In conventional implementations, sharing of CCC operations with non-CCC operations may result in a significant amount of additional circuitry and parasitics being associated with the execution of CCCs (*e.g.*, additional multiplexers and control circuitry, and larger clock networks and global buses). A separate implementation of the CCC alone would avoid these above problems.
- Since the CCCs result in a much smaller sub-circuit than the complete circuit, sub-optimal (heuristic) synthesis algorithms often tend to perform better on them than when they are given large monolithic designs. Conversely, more aggressive and computationally intensive synthesis/optimization algorithms may be used to optimize CCCs.

CCCs have been exploited in various related areas of research. The observation that often under 10% of a program's instructions accounts for over 90% of its execution time has been exploited in the context of high-performance processor and compiler design [3, 4]. As a popular example, one of the arguments driving the evolution of reduced instruction set computer (RISC) architectures was that they allowed for simplified implementations of frequently occurring instructions [3]. Trace scheduling [4] exploits CCCs by compacting frequently occurring program threads using code motion. Another related logic-level power reduction technique, called precomputation [5], optimizes an embedded combinational circuit block by adding significantly simpler circuits (called predictor circuits), which compute the output and disable the original circuit for a subset of input conditions. In the context of logic synthesis, the principle of optimizing for the common case has been exploited for performance optimization in [6, 7]. Past work in the area of high-level power optimization has addressed scheduling, allocation, binding, power management, and behavioral transformations [8], but has not paid attention to analyzing, detecting, and simplifying common cases.

2 Common-Case Computation based Design

In this section, we present the basic ideas, as well as detailed tradeoffs involved in optimizing circuits for CCCs using illustrative examples. Section 2.1 illustrates the basic steps involved in CCC-based design. Section 2.2 illustrates the complex issues and tradeoffs involved in some of these steps. These ideas are later formalized into algorithms for power or performance optimization in Section 3.

2.1 Fundamentals

We now illustrate the CCC idea using the greatest common divisor (`GCD`) example, whose structure and schedule are shown in Figures 1(a) and 1(b), respectively [1]. The first step involved in the CCC-based design methodology is to identify one or a few candidate state sequence patterns from the original design's simulation traces, using which we later derive CCC circuits. From the STG of Figure 1(b), it

DAC 99, New Orleans, Louisiana

[1] The schedule is represented in the form of a state transition graph (STG) whose edges are annotated with state transition probabilities and nodes with state probabilities. The probabilities are recorded during simulation with a typical testbench.

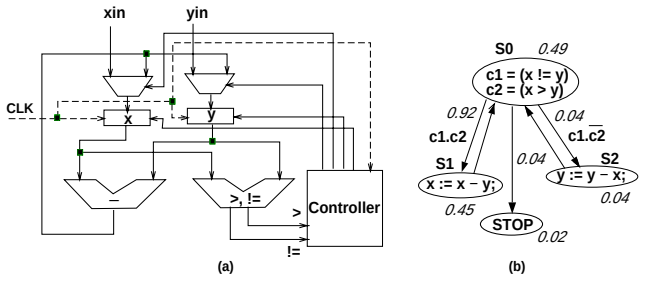

Figure 1: (a) Structural RTL implementation, and (b) scheduled description for the GCD circuit

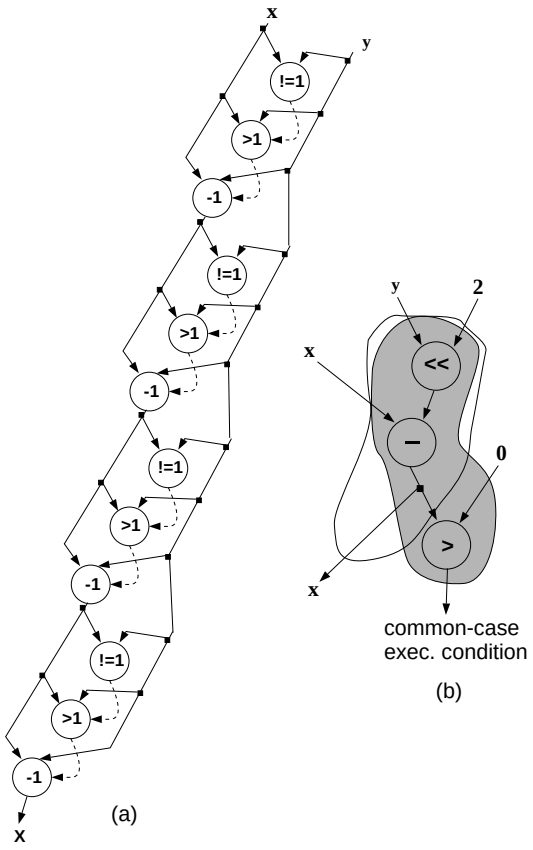

Figure 2: (a) The extracted common-case GCD behavior, and (b) a simplified common-case behavior

is clear that the state probabilities of states $S0$ and $S1$ are high, as is the probability of a transition between them. Upon performing a further automatic analysis of the GCD design and its execution traces during simulation, we found that $S0, S1, S0, S1, S0, S1, S0, S1$ was a promising state sequence pattern to derive a CCC circuit.

Given a candidate state sequence pattern, the next step in our CCC-based design flow is to extract the behavior induced by it and its execution condition, as explained below. We define the *behavior induced by a state sequence pattern* in a schedule (STG) as the set of operations that are executed when the given pattern is traversed in the STG. The behavior induced by the pattern $S0, S1, S0, S1, S0, S1, S0, S1$ in the schedule of Figure 1(b) is shown as a control-data flow graph in Figure 2(a). Similarly, we define the *execution condition of a state sequence pattern* in a schedule as the set of conditions that need to be satisfied in order to traverse the

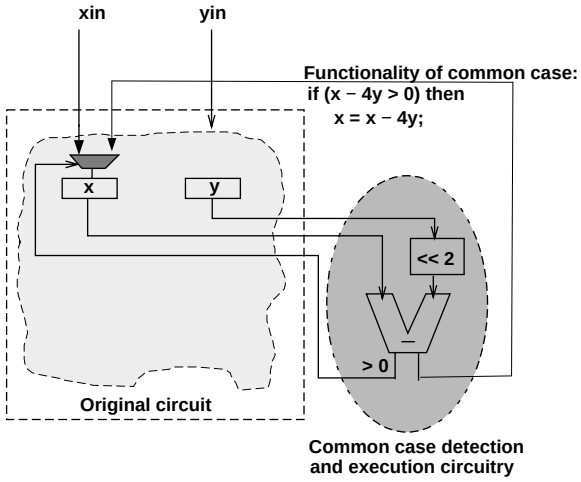

Figure 3: Optimized GCD design including CCC circuitry

given sequence of states, given that the STG is first initialized to the first state of the sequence. The execution condition of the state sequence pattern $S0, S1, S0, S1, S0, S1, S0, S1$ in the schedule of Figure 1(b) is the conjunction of the outputs of all eight conditional operations shown in Figure 2(a).

As mentioned in Section 1, several optimizations that are not applicable in the context of the original design may be applicable in the limited context of CCCs. Thus, an important step in the CCC-based design methodology is to further optimize the (relatively small) common-case behavior aggressively using known power and performance optimization techniques. Since the common-case operations extracted from the schedule are represented at the behavior level, a natural choice is to use behavioral transformations [9, 10, 13, 14] to simplify them. We use a powerful transformation framework to apply various transformations aimed at minimizing the number of operations, and the critical path, of the common-case behavior.

For the GCD example, the initial common-case behavior shown in Figure 2(a) is automatically transformed into the simplified behavior shown in Figure 2(b). The sequence of four $(-)$ operations has been reduced to one left shift $(<<)$ operation and one (-) operation. In addition, the common-case execution condition has been simplified from a sequence of four $(>)$ and four $(!=)$ operations to a single $(>)$ operation. Note that this optimization is not valid in the context of the original design, but only in the scenarios under which the common-case behavior is executed. Algorithms used for automatically optimizing the common-case behavior in our CCC-based design methodology are described in Section 3.

The final, optimized GCD RTL design that contains an implementation of the common-case circuitry is shown in Figure 3. The circuitry added or modified for implementation of the CCC circuitry is indicated using the darker shade of grey. In this example, the CCC and execution condition are both implemented using a single *subtracter* (the condition $x - 4y > 0$ has been implemented using the borrow and the zero-detect output of the subtracter).

Upon synthesis, the average energy per input for the original design was found to be $11.05nJ$, and for the CCC-optimized design was found to be $4.46nJ$, representing an energy savings factor of $2.48X$. In addition, the average amount of execution time (number of clock cycles $\times$ clock period) per input was found to be $4,285ns$ and $1,003ns$, respectively, for the original and CCC-optimized design, a performance improvement of $4.27X$. The energy savings factor becomes $12.08X$ if the performance improvement is traded off through supply voltage scaling. Section 4 details our experimental methodology.

2.2 Tradeoffs involved in selecting CCCs

This subsection shows that the selection of the common-case behavior has a significant bearing on the quality of results obtained. It also demonstrates that it is important to take data statistics (since

they influence the probability of executing the common-case behavior) into account during CCC selection.

In general, the following tradeoffs are involved when performing common-case behavior selection:

- **Coverage.** The coverage of a state sequence pattern represents the expected fraction of the original design's total processing time that will be spent in executing instances of the pattern. Very small state sequence patterns (that involve very few distinct states) may not be desirable since they may not exploit enough of the state space to result in a large coverage. On the other hand, state sequence patterns that are larger than necessary may have poor coverage since they may be too specialized, *i.e.*, not occur frequently enough. For example, consider the pattern $S0$ in the `GCD` schedule of Figure 1(b). The coverage of this pattern is equal to the state probability of $S0$, *i.e.*, 0.49. [2]

  The next example shows that the problem of selecting a state sequence pattern to maximize coverage is a non-trivial one.

  Example 1: Consider the schedule shown in Figure 4. We computed the optimal coverage achievable by state sequence patterns of length less than or equal to 150. The results are presented as a plot of coverage *vs.* pattern length in Figure 5. The achievable coverage initially increases with pattern length, but starts decreasing after a point. Note that the pattern length 13 in Figure 5 is only a local optimum, since it is always possible to have the entire simulation state trace as a trivial pattern of coverage 1.0. However, then the common-case behavior chosen will correspond to the entire design, which will not lead to any power or execution time savings. Thus, it is typically necessary to have an upper bound on the length of state sequence patterns. ∎

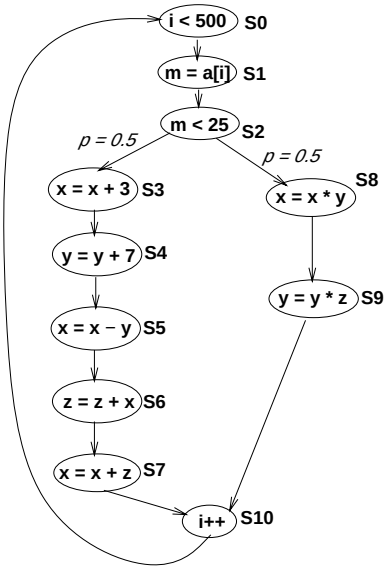

Figure 4: Schedule for example circuit `test1`

The following factors also need to be considered.

- **Scope for optimization.** Once the common-case behavior is derived from the chosen state sequence pattern, it is further optimized in order to minimize power consumption or execution time. Longer state sequence patterns typically lead to behaviors that offer more opportunities for optimization. However, if the state sequence pattern is too long, the coverage suffers.

[2] Note that the coverage of state sequence patterns of length greater than 2 cannot be computed directly from the state and state transition probabilities, which only indicate first order statistics and ignore higher order statistics such as the self and mutual correlations of state transition conditions [11].

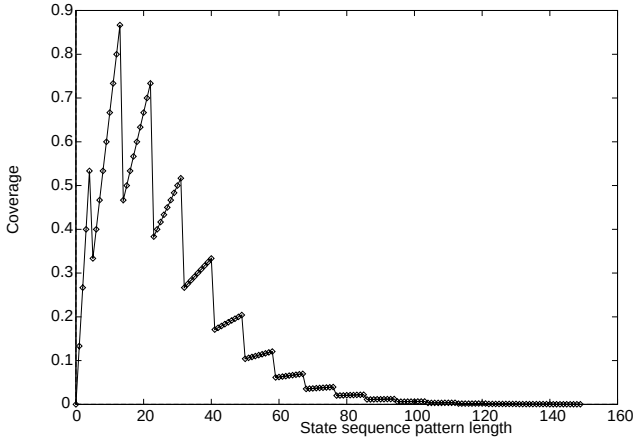

Figure 5: Plot of coverage *vs.* state sequence pattern length

- **Compactness of common-case circuitry.** One of the advantages of CCC-based design, as mentioned in Section 1, is that it eliminates a lot of the additional circuitry and parasitics (multiplexer and control circuitry, and clock network and global bus capacitance) activated during the execution of the common-case behavior in a non-CCC-based design. These effects rely on the requirement that the common-case circuitry is much smaller than the complete design. Due to the difficulty of estimating such low-level parasitics as the clock and interconnect capacitance at the behavior level, we do not directly model or target this factor in choosing a state sequence pattern for deriving the common-case behavior. Instead, our algorithms accept constraints on the number of resources of each type (functional units and registers) that are allowed in the implementation of the common-case behavior. This parameter provides a handle to effectively limit the size of the common-case circuitry.

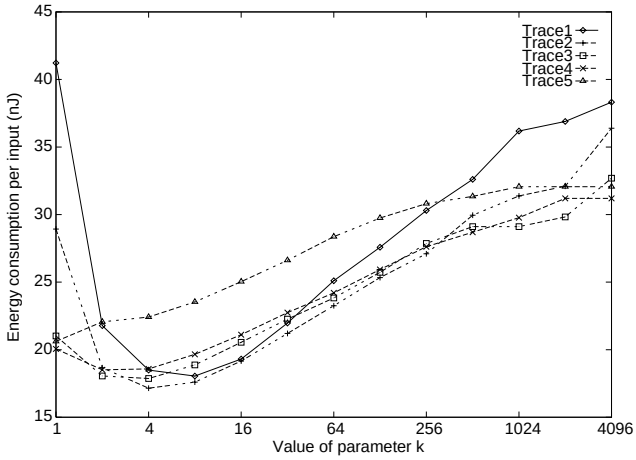

Figure 6: Energy consumption of CCC-based designs vs. state sequence pattern length for different input traces

Example 2: Consider again the `GCD` example shown in Figure 1. In order to show the tradeoffs involved in choosing an appropriate state sequence pattern, we obtained CCC-based designs for several candidate state sequence patterns of different lengths, and evaluated them for performance and energy consumption for different simulation testbenches having different input distributions. Since the loop involving states $S0$ and $S1$ accounts for most of the simulation time, we considered CCC-based designs that use k copies of the loop, *i.e.*,

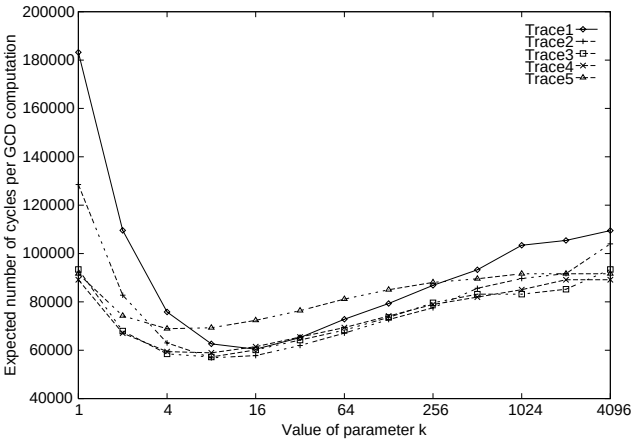

Figure 7: Expected execution time of CCC-based designs vs. state sequence pattern length for different input traces

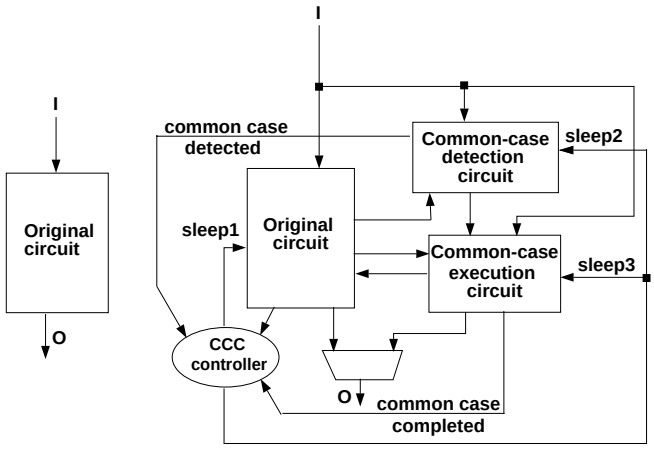

Figure 8: Original and CCC-optimized circuits

$(S0, S1)^k$ as the chosen state sequence pattern, for various values of k ranging from 1 (representing the original design itself) to 4,096. Each of these designs was evaluated for energy consumed per GCD computation and performance (time consumed per GCD computation) for five different input traces labeled $Trace1, \ldots, Trace5$. The input traces were generated as follows. Each input trace corresponds to a fixed number (500) of GCD computations, where each GCD computation may take a different number of cycles depending on the values of inputs x and y. The values of x for all five traces were generated using a uniform distribution between 0 and $2^{20} - 1$. The values of y for $Trace1, \ldots, Trace5$ were generated using uniform distributions between 0 and $2^{16} - 1, 2^{17} - 1, \ldots, 2^{20} - 1$, respectively.

The energy results from the above experiments are plotted in Figure 6. The results indicate whether energy savings can be obtained using the CCC-based design, and if so, which value of k leads to the best energy design. This depends in a complex manner on the input traces. Note that, on the one hand, for $Trace5$, $k = 1$ leads to the least energy consumption, *i.e.*, CCC-based design with other values of k do not result in any energy savings. On the other hand, for $Trace1$, a large energy savings (about $2.3X$) is possible compared to the original design. Also, note that for the curves which attain least energy at $k > 1$ (*i.e.*, at least one CCC-based design is better than the original design), the following observations hold: (i) the best value of k varies depending on the input trace, and (ii) the smallest and largest considered values of k never lead to the best design.

Figure 7 shows results for performance (number of clock cycles) for the same experiments. Again, it is clear that judicious selection of the value of k is necessary to realize the full potential of CCC-based design, and the best value of k varies depending on the input data statistics. Another important point illustrated by Figures 6 and 7 is that energy and performance optimization are sometimes divergent goals. For example, CCC-based designs derived for all values of parameter k result in performance improvements over the original (non-CCC-based) designs. However, some values of k and input distributions result in CCC-based designs that consume more energy than the original designs. ∎

The above example illustrates two key concepts:

- There are several (possibly conflicting) factors involved in choosing a common-case behavior that leads to maximal energy (or power) and execution time savings.
- Input data statistics play an important role in determining the best common-case behavior.

The algorithms presented in Section 3 quantitatively explore these factors in choosing the best common-case behavior.

3 The Algorithm for CCC-based Design

In this section, we present the algorithmic details of our power or performance optimization technique. The inputs to the algorithm are an STG representing the schedule, a complete RTL description of the circuit to be optimized, and a set of typical input traces. The output is the RTL circuit augmented with hardware that detects and executes some common cases in a power- or performance-efficient manner. Another way to employ our technique is to use it as a plug-in to an existing high-level synthesis tool (note that we do not assume anything about the algorithms employed in high-level synthesis). In such a scenario, the high-level synthesis tool would be used to separately generate the RTL circuitry to implement the common-case behavior and the remaining parts of the behavior. The resulting circuits would be composed into a CCC-optimized implementation.

Figure 8 shows the structures of the original and CCC-optimized circuits. The optimized circuit has three major components: the original circuit, a *common-case detection* circuit, and a *common-case execution* circuit. The common-case detection circuit accepts as inputs, the primary inputs of the circuit, and the values of some internal variables in the original circuit. It detects the occurrence of a specific condition, referred to in the sequel as the *common case*. The inputs of the common case execution circuit could be any subset of the primary inputs and internal variables in the original circuit. When activated, it computes a subset of the primary outputs and the values of some internal variables in the original circuit. Each of the three components of the CCC-optimized circuits is designed to support power management (*sleep* mode) using a combination of clock gating and operand isolation [8, 12]. Clock gating ensures that the registers do not load new values and that the clock network does not dissipate power. Operand isolation uses transparent latches to freeze the non-registered primary inputs. When the sleep input to a circuit is asserted high, it does not dissipate any dynamic power. A circuit in the sleep mode can be restored to active mode in the next clock cycle by asserting the sleep input low. The sleep inputs to the various circuits are generated by a small global controller, using the controller state from the original RTL circuit, the common-case detection signal, and a completion signal generated by the common-case execution circuit.

Figure 9 illustrates the chronology of related events. Rectangles (a), (b), and (c) represent, respectively, the activity of the original circuit, the common-case detection circuit, and the common-case execution circuit over time. The shaded regions of the rectangles correspond to activity in the circuit, and the clear regions correspond to inactive or idle time slots. During the idle slots, the component is sent into sleep mode by the control circuitry. Initially, only the original circuit is active. At time t_p, the original circuit enters into a state which activates the common-case detection circuit. The common-case detection circuit uses the primary inputs and the values of internal variables in the original circuit to test for the occurrence of the common case. This process continues until time t_c. Note that, to avoid performance degradation, the original circuit continues its computation between times t_p and t_c. At time t_c, the common-case detection circuit confirms the occurrence of the common case, and activates the common-case execution circuit. The original circuit

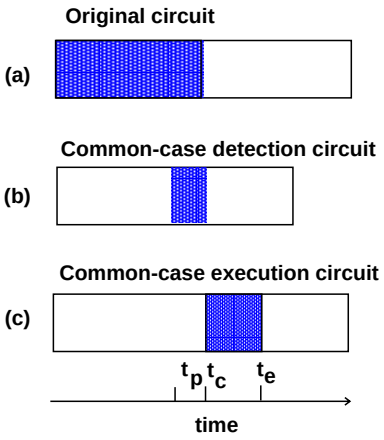

Figure 9: Activity of CCC-optimized circuit over time

and the common-case detection circuit are then de-activated. The common-case execution circuit completes at time t_e, and writes the appropriate values into the original circuit, which then resumes normal computation.

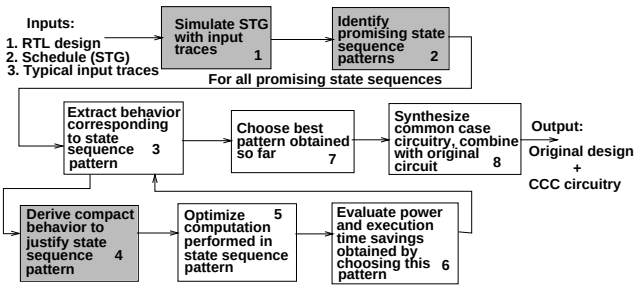

Figure 10: The CCC optimization algorithm

Figure 10 outlines our algorithm. We first simulate the STG with the input traces to obtain a sequence, ρ, of states. We then traverse the state sequence to identify frequently-encountered sub-sequences of states, which potentially constitute good common cases. The output of step **2** in the algorithm is a set of state sequence patterns which can potentially be synthesized into an efficient common-case circuitry. It is important to note that steps **1** and **2** cannot identify the *best* state sequence pattern, but only obtain some promising ones. This is because these steps do not use detailed synthesis information, which is needed to assess the ease of detection, and the simplicity of the common case. Therefore, these steps cannot rank closely-matched state sequence patterns. Rather, they serve as a filter that protect later stages from having to focus synthesis effort on obviously undesirable state sequence patterns. $Gain(\sigma)$ (Equation (1)) measures the desirability of the state sequence pattern σ as a CCC.

$$Gain(\sigma) = Coverage(\sigma) \times |\sigma| \quad (1)$$

$$= N(\sigma) \times |\sigma|^2 \quad (2)$$

Coverage has been defined in Section 2, $|\sigma|$ is the length of σ, and $N(\sigma)$ represents the number of non-overlapping instances of σ in ρ. Note that $Gain(\sigma)$ can be easily computed without any knowledge of the behavior represented by the common case. We now justify our choice of this measure. Of two state sequence patterns with equal length, the one which occurs more often would clearly constitute a better choice, if behavioral information is unavailable. The *Gain* function is, therefore, proportional to $N(\sigma)$. Consider two different state sequence patterns, σ_1 and σ_2, which occur 10 and 50 times, respectively. Suppose σ_1 has a length of 5, and σ_2 has a length of 1. If we chose $Gain(\sigma)$ to be proportional to $Coverage(\sigma)$, then these two state sequence patterns would be considered equally good. However, a common case that consists of a longer state sequence pattern is likely to be easier to optimize. A behavior extracted from a single state would be extremely difficult to optimize because it has a very short critical path (of one cycle). Multiplying $Coverage(\sigma)$ by $|\sigma|$ takes into account the increased ability to optimize larger behaviors. Extremely long state sequences are, however, undesirable because they add to the complexity of the common-case detection and execution circuits, thus increasing the overall power consumption of the design. We, therefore, upper-bound the length of the common case state sequence pattern by a small, user-defined constant. For our experiments, a bound of 32 on the length of the common case yielded good results.

Steps **3**-**6** are performed for all promising state sequence patterns. Step **3** extracts the behavior corresponding to a state sequence pattern. Step **4** derives a compact *justification behavior* for the common case implied by the state sequence pattern, *i.e.*, it derives a set of conditions, which, if satisfied, guarantees the occurrence of the chosen state sequence pattern. This is done as follows: consider a state sequence pattern, $\sigma = \{S_1, S_2, \ldots, S_n\}$, whose occurrence needs to be detected. Let c_i represent the condition for a transition from state S_i to S_{i+1}. x is a Boolean variable which is *true* if and only if σ occurs. Clearly, $x = \bigwedge_{i=1}^{n-1} c_i$. When an instance of the common case is detected, the common case execution hardware is activated. Therefore, it is critical that the simplified behavior does not incorrectly report the occurrence of a common case. However, in the interest of ease of detection, the detection process might choose to *ignore* some hard-to-detect, infrequent, occurrences of the common case. Therefore, the output, x', of the detection process is required to be *true* only if x is *true*. It is hard to find a general simplifying procedure, which works for all behaviors. From our experiments, we identified some promising directions for simplification. Specifically, we noted the existence of implications between the c_i's for many of our benchmarks, *i.e.*, a *true* value on c_{i1} often guarantees a *true* value on c_{i2} ($i1 \neq i2$). In this case, we can remove c_{i2} from the detection process, and, therefore, also remove the operations which are responsible only for its generation, thus simplifying the behavior.

Step **5** derives the common-case execution circuit. In this step, optimizing transformations are applied to simplify the common-case behavior, prior to synthesis. Power-optimizing transformations have been extensively studied in the literature [10, 13, 14]. Performance-optimizing transformations can also be used at this point [9]. At the end of this step, the simplicity of the common-case circuit can be assessed. We also have sufficient information to estimate the power and execution time savings obtainable from the chosen pattern. Step **6** performs this estimate.

The process described in the previous paragraph is repeated for every state sequence pattern identified in steps **1** and **2**, and the most promising pattern is chosen as the common case. The common-case detection and execution circuits for this pattern are then combined with the original circuit in step **8** to produce a CCC-optimized circuit.

4 Experimental Results

While CCC optimization can target power or performance, we next present experimental results for several circuits where power optimization was performed. Scheduling and binding information was available for all circuits. The STGs representing the schedules of the example circuits were analyzed to detect common cases, and the most promising common case was chosen for synthesis. The original circuits were modified by adding common-case detection and execution circuitry. The original and power-optimized RTL descriptions were mapped to gate-level netlists using synthesis tools from the NEC CAD tool suite, OpenCAD [15]. The resulting gate-level circuits were compared with respect to the following metrics: area, performance, and power. The area, delay, and power consumption were extracted from technology-mapped gate-level circuits using static timing analysis tools and power estimation tools from the NEC OpenCAD suite [15]. The results obtained are summarized in Tables 1 and 2.

The power consumption of the original and the optimized designs are computed in the following manner when V_{dd}-scaling is not performed. For the original design, the energy, E_{orig}, consumed while

Table 1: Area and performance results

Circuit	*Area (# transistor pairs)*			*# cycles*		
	original	*optimized*	*A.O.*(%)	*original*	*optimized*	*P.I.* (%)
`GCD`	3,647	4,706	29.0	428,460	100,310	76.6
`Poly`	16,801	19,232	14.5	1,760,000	1,445,000	17.9
`Test1`	10,163	12,386	21.9	338,300	194,600	42.5
`Linegen`	3,340	4,126	23.5	718,000	406,800	43.3
`Graphics`	5,894	7,644	29.7	159,800	116,000	27.4

Table 2: Power results

Circuit	*Power (mW)* (non-V_{dd}-scaled)			*Power (mW)* (V_{dd}-scaled)	
	original	*optimized*	*P.S.* (%)	*optimized*	*P.S.* (%)
`GCD`	2.59	1.04	59.8	0.22	91.5
`Poly`	55.57	23.23	58.2	17.98	67.6
`Test1`	23.74	12.20	48.6	6.17	74.0
`Linegen`	8.96	5.40	39.7	2.69	70.0
`Graphics`	18.48	16.19	12.4	10.77	41.7

executing the input trace is divided by the time, T_{orig}, in cycles, taken for executing the trace, to determine the power consumption. The power-optimized design is assumed to consume E_{opt} units of energy and operate for T_{opt} cycles, where T_{opt} is less than T_{orig}. In this case, the optimized design is assumed to operate for T_{orig} cycles, while being in an inactive (zero energy) state for $T_{orig} - T_{opt}$ cycles. Therefore, the power consumption is given by E_{opt}/T_{orig}. If V_{dd}-scaling is performed, the optimized design is assumed to take the same time as the original design. This enables us to use the following equation to scale the supply voltage [10]

$$\frac{V_{dd}initial}{(V_{dd}initial - V_t)^2} \times T_{orig} = \frac{V_{dd}new}{(V_{dd}new - V_t)^2} \times T_{opt}$$

where $V_{dd}initial$ is the initial supply voltage, $V_{dd}new$ is the new supply voltage, and V_t is the threshold voltage of the implementation. The power consumption is obtained using the new supply voltage.

In Table 1, major columns *circuit*, *area* and *# cycles* represent the name of the design, the area, and the expected number of clock cycles to process one input, respectively. Minor columns *original* and *optimized* represent, respectively, the original design and optimized design. Column *A.O.* represents the area overhead incurred by our technique, and column *P.I.* represents the improvement in performance. Similarly in Table 2, major columns *circuit*, *power* (non-V_{dd}-scaled), and *power* (V_{dd}-scaled) represent, respectively, the name of the design and the power consumption, without and with V_{dd}-scaling. Column *P.S.* represents the savings in power consumption.

Of our examples, `GCD` is a well-known benchmark. `Poly` represents the computation of a polynomial, and `Test1` represents the behavior shown in Figure 4. `Linegen` and `Graphics` are parts of an in-house graphics controller ASIC.

The results indicate that our power optimization procedure produces circuits which perform significantly faster or consume significantly lower power than the original designs. On an average, the circuits produced from our technique consumed 69.0% (43.7%) less power than the original circuits when V_{dd}-scaling was (was not) performed, at an average area overhead of 23.7%. These power reductions are only achieved when the performance of the original and optimized designs is made equal. The optimized circuits performed, on an average, 41.5% faster than the original circuits. However, note that for the faster circuits, we do not get the above power reductions. Thus, our technique can either optimize power or performance, but not necessarily both.

5 Conclusions

In this paper, we presented a technique that performs power or performance optimization by identifying and specially synthesizing frequently-encountered behavioral fragments, or common cases. We introduced a technique to identify promising common cases, and simplify the detection and execution of the chosen common case by using targeted behavioral transformations. We also proposed an architecture to implement our optimization technique. Experimental results, performed on several benchmarks, demonstrate significant power savings or performance improvements at reasonable area overheads.

References

[1] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin, *High-level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, Norwell, MA, 1992.

[2] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, New York, NY, 1994.

[3] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman Publishers, San Mateo, CA, 1989.

[4] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Computers*, vol. C-30, pp. 478–490, July 1981.

[5] M. Aldina, J. Monteiro, S. Devadas, A. Ghosh, and M. Papaefthymiou, "Precomputation-based sequential logic optimization for low power," *IEEE Trans. VLSI Systems*, vol. 2, pp. 426–436, Dec. 1994.

[6] L. Benini, E. Macii, M. Poncino, and G. De Micheli, "Telescopic units: A new paradigm for performance optimization of VLSI designs," *IEEE Trans. Computer-Aided Design*, vol. 17, pp. 220–232, Mar. 1998.

[7] S. K. Bommu, N. O'Neill, and M. Ciesielski, "Retiming based factorization for sequential logic optimization," *ACM Trans. Design Automation Electronic Systems*, to appear, 1998.

[8] A. Raghunathan, N. K. Jha, and S. Dey, *High-level Power Analysis and Optimization*, Kluwer Academic Publishers, Norwell, MA, 1998.

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

[10] A. P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, and R. Brodersen, "Optimizing power using transformations," *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 12–31, Jan. 1995.

[11] G. Casella and R. L. Berger, *Statistical Inference*, Duxbury Press, Belmont, CA, 1990.

[12] L. Benini and G. De Micheli, *Dynamic Power Management: Design Techniques and CAD Tools*, Kluwer Academic Publishers, Norwell, MA, 1997.

[13] A. Chatterjee and R. K. Roy, "Synthesis of low power DSP circuits using activity metrics," in *Proc. Intl. Conf. VLSI Design*, pp. 255–270, Jan. 1994.

[14] G. Lakshminarayana and N. K. Jha, "FACT: A framework for the application of throughput and power optimizing transformations to control-flow intensive behavioral descriptions," in *Proc. Design Automation Conf.*, pp. 102–107, June 1998.

[15] *OpenCAD V 5 Users Manual*, NEC Electronics, Inc., Sept. 1997.