A Scalable ODC-Based Algorithm for RTL Insertion of Gated Clocks

Pietro Babighian ‡ Luca Benini * Enrico Macii ‡

‡ Politecnico di Torino
Torino, ITALY 10129

* Università di Bologna
Bologna, ITALY 40136

Abstract

This paper describes a new automatic clock-gating extraction algorithm working at the RT-level. The key features of our approach are: (i) Seamless merging with existing industrial design flows and commercial tools; (ii) High scalability to deal with large circuits; (iii) Improved quality of results with respect to available commercial tools; (iv) Smaller and well-controlled overhead in speed and area. Experimental results, on a set of industrial RTL designs, demonstrate the viability and practical impact of our approach.

1 Introduction

Complex digital circuits contain units that do not perform useful computations at every clock cycle. It is then possible to disable the corresponding logic when it is not in use during a particular clock cycle, with the objective of limiting power consumption. Power optimization techniques based on the principle above belong to the broad class of *dynamic power management* (DPM) methods [1].

Clock gating provides a way to selectively stop the clock whenever the computation to be carried out by a hardware unit at the next clock cycle is redundant. Clock gating can significantly reduce the switching activity both in the logic and the clock net. Thus, it is viewed as one of the most effective logic, RTL and architectural power reduction techniques [2].

Even though clock gating is widely adopted, it can have a significant impact on a complex design flow. Design-for-testability is impacted by gated clocks and several techniques have been proposed to design highly-testable circuits with gated clock [3]. Furthermore, clock tree design is considerably complicated by the presence of gated clocks, that introduce skew and wiring overhead [4]. These diffculties have slowed down the development of commercial tools for automatic clock gating insertion, despite an intense research activity on the topic (refer to [5], Chapter 9, for a recent survey).

Under the pressure of tightening power budgets, computer-aided clock gating tools have recently appeared on the market. All these tools take a very conservative standpoint: They provide good support for specifying gated clocks at the register-transfer level, and for controlling clock gating instantiation and synthesis. However, only very limited support is available for detecting and extracting idle conditions when a designer does not explicitly specify them at the RTL with a suitable HDL construct.

Our work aims at exploiting the expanding commercial tool support for clock gating, to push the envelope with new, more aggressive techniques for automatic clock gating extraction, which: (i) Integrate with available commercial tools, to ensure design flow continuity; (ii) Can be applied to large RTL designs; (iii) Significantly improve quality of results with respect to currently available options. To satisfy these requirements, our approach not only does reduce power, but also minimizes the impact of gated clocks insertion on other metrics, namely, speed and area.

Our emphasis is on *scalable* algorithms for detecting non-trivial redundant clocking in large RTL designs (using approximated *observability don't cares* computation), and for synthesizing idleness-detection logic (i.e., clock activation functions) that can control the gated clocks with minimal speed and area overhead.

The manuscript is organized as follows. Section 2 briefly reviews work related to ours. In Section 3 we present an example that motivates our approach while Section 4 describes our contribution. We present experimental results in Section 5 and Section 6 closes the paper.

2 Related work

The automatic clock gating extraction methods presented in the scientific literature can be coarsely grouped under three classes:

- *FSM-based* approaches, which assume an input specification in form of a finite-state machine [6].
- *Symbolic* approaches, which assume an input specification in form of a gate-level netlist [7].
- *HDL-analysis* tools, which assume an input specification in a hardware description language [8].

The main limitation of the approaches in the first class is that they are not scalable. Designs with a few tens of flip-flops have an unmanageably large FSM description. Symbolic tools have moved the circuit size limit forward, but they require symbolic manipulation of Boolean functions to compute idle conditions, and they are ultimately limited by BDD blowup. HDL-analysis approaches detect specific patterns in the HDL description and they can scale to very large designs. However, they have limited capabilities in extracting clock gating conditions that are not explicitly expressed in form of conditional sequential statements (where the state is updated only if an enable signal

1530-1591/04 $20.00 (c) 2004 IEEE

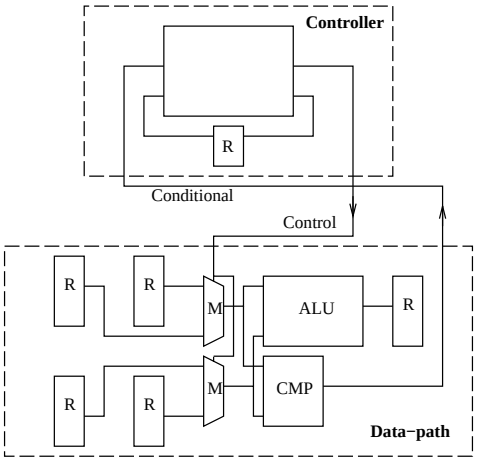

Figure 1: Structure of a generic RTL specification.

is valid). In other words, HDL approaches lack generality in their idleness detection procedures.
A few attempts have been made in addressing the scalability and generality issues. Approximate symbolic approaches have been proposed that could, in principle, trade off accuracy in idleness condition computation with memory usage [9, 10], but until now no results have been reported on circuits with more than a few tens of flip-flops, hence true scalability remains to be proven. On the other hand, HDL techniques for aggressive idleness extraction have been proposed [11], but they have been demonstrated only on small circuits, because the semantic HDL analysis required to extract idleness is even more complex (in the worst case) than BDD-based Boolean manipulation.

2.1 RTL Approaches

A promising approach, which can be viewed as an hybrid between symbolic and HDL-analysis approaches has been proposed by Kapadia et al. [12]. It is a true register-transfer level method. At the RTL the design description is usually partitioned in two main blocks: controller and data-path (Figure 1). Controller and data-path communicate through two types of signals. *Control signals* are outputs of the controller and inputs of the data-path. *Conditional signals* are outputs of the data-path and inputs of the controller.
The data-path usually consists of two types of resources: *computational units* such as adders, comparators, multipliers and *steering logic*, which consists usually of multiplexers and three-state buffers, steered by control signals. Another important component of the data-path is the interconnection structure (busses and global signals). State storage in the data-path is based on *registers*, which are banks of flip-flops. *Register width* is the number of single-bit flip-flops in a register. Registers can be enabled by control signals, hence they can be seen as state-holding steering elements.
To extract idleness conditions from an RTL netlist, Kapadia et al. focus on control signals that drive steering modules, and they observe that steering modules are the source of easily detectable *observability don't care* (ODC) conditions. Intuitively, if a multiplexer is selecting one branch, all other branches become unobservable, and their logic value is irrelevant to the correct operation. Hence, the clock of flip-flops in the fanin cone of a unobservable multiplexor branch can be gated without compromising functionality. Kapadia et al. propose a simple traversal algorithm that computes the ODC for datapath flip-flops (and latches) focusing only on steering modules. Even though all datapath blocks (including arithmetic units) create ODCs on their inputs, their computation is computationally demanding. Hence, it is assumed that their ODCs are null (this is clearly a conservative approximation).
Interestingly enough, the clock gating detection capabilities of current commercial synthesis tools can be viewed as a particular case of ODCs. In fact, by detecting clock gating conditions only when enable signals are present, they just find ODCs generated locally to the registers.
Even though Kapadia et al. demonstrated the good scalability of their approach on a large processor datapath, the technique lacks generality in a synthesis-based ASIC design environment for three reasons:

- When parsing HDL specifications, synthesis tools instantiate a structural RTL netlist (containing generic logic macros) where control signals and data signals are not clearly decoupled.
- Generally control signals are locally transformed through several stages of logic before being fed to steering modules.
- While in flat processor datapaths ODCs remain very simple, more general RTL netlists can have many levels of steering modules and even simplified ODC computation and propagation can become extremely unwieldy.

Our approach addresses all above mentioned limitations, as described in the following sections.

3 Motivating Example

Operand isolation and *clock gating* strategies represent two of the most popular techniques for dynamic power reduction at the RT level. In particular, *operand isolation* reduces power dissipation by inserting isolation logic along with an activation signal in order to hold data-path operators inputs stable whenever their output is not used. On the contrary, *clock gating* strategy saves power by gating register banks when their enable signals are not active. However, there are some corner cases in which these two methodologies are not performable by modern computer-aided optimization tools. Consider the logic network shown in Figure 2. Block L represents a cloud of elementary logic gates driven by a register bank the enable of which is always active. In such a situation neither clock gating nor operand isolation can be adopted. In particular, clock gating logic will never stop clock signal since the registers are

always turned on. In this case, clock gating logic insertion has an opposite effect providing an extra power dissipation. On the other hand, commercial tools perform operand isolation only for arithmetic operators such as ALUs, adders and multipliers represented as macros whithin the design. Hence, if block L uses simple logic gates to implement an arithmetic operator the operator itself will never be isolated.

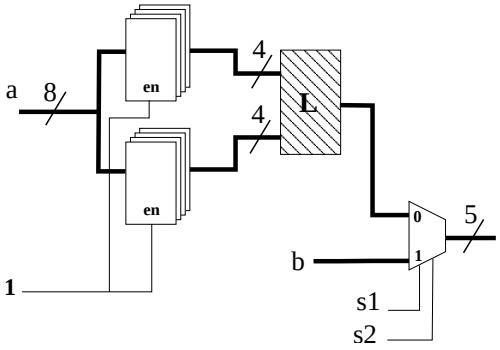

Figure 2: Motivating example.

4 ODC-Based Gated-Clock Extraction

The goal of our technique is to condition the activation of a generic register by extracting a signal from the logic description of the circuit based on the ODC conditions of the register to be isolated.

4.1 ODC Computation

We compute a safe approximation of the actual ODCs by assuming that computational units are fully observable and that they do not reduce the observability of their inputs. Only steering modules reduce the observability of their inputs, and they are the ones that need to be analyzed for extracting idle conditions.

Example 1 *Consider the steering modules shown in Figure 3. All but one data inputs of the MUX are unobservable at any given time (a). The three-state driver makes its input unobservable if its enable signal is de-asserted (b). The same holds for the enabled register (c). In symbols, the ODCs are:* $ODC_{Data_0} = S_0'$ *and* $ODC_{Data_1} = S_1'$ *for the MUX;* $ODC_{Data} = En'$ *for the three-state driver and the register.*

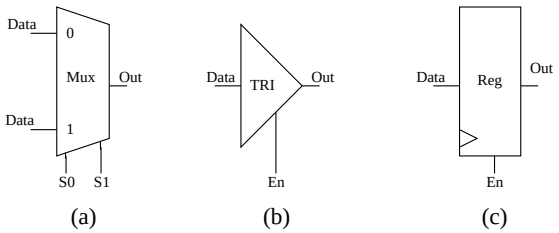

Figure 3: Steering Modules in a RTL Description.

It is easy to compute the observability loss caused by steering modules and, more importantly, the logic for idleness detection is compact (simple functions of control inputs of the steering modules). Given a steering module, the ODC of its output is the intersection of the ODCs of all its fanout stems. The ODC of one of its inputs is thus logic sum of the ODC of its output and the additional ODC condition produced by the steering module itself. If no information on external *don't cares* is available, the ODCs of the primary outputs are assumed to be empty. The presence of a computational unit is completely transparent. The ODCs of all its inputs is assumed to be the same as the ODC of its output. These simple rules are applied for each steering level moving backward in the network, as described in the pseudo-code of Figure 4 and shown in Figure 5.

```
BackTrav (StartLevel, Start, DrivingList) {
  if (Start = True) {
     level = 0;
     /* all ODCs for PIs are inizialized to 0 */
     /* (always observable) */
     foreach InputPin {
        ODC(InputPin) = 0;
     }
     /* unknown ODCs of POs are fixed to 0 */
     foreach Primary Output {
        if (ODC(Primary Output) is not given)
              ODC(Primary Output) = 0;
     }
     /* Level 0 (FFs) is created and preliminary ODCs */
     /* of their inputs are computed */
     foreach FF {
        Level(FF) = level;
        /* FF is a generic flip-flop with N fanout */
        /* stems FF_Out_1, FF_Out_2, ...,FF_Out_N  */
        ODC(FF_Output) &= ODC(FF_Out_i);
        /* OR-ing ODC(FF_Output) with the ODC of the module */
        Compute ODC(FF_Input);
     }
     Get DrivingList(FF_list);
  } else {
      /* starting topological levelization */
      foreach cell_1 in DrivingList {
          Get SuccessorList;
          foreach cell_2 in SuccessorList {
              if (Level(cell_2) does not exist) {
                   List = remove_from_driving(cell_1);
                   exit foreach;
              }
          }
      }
      foreach cell in List {
          Level(cell) = level + 1;
          level = level + 1;
          if (Level(cell) <= StartLevel) {
              ODC(Output) &= ODC(Outi);
              /* ODCs of Steering module data inputs */
              /* are created */
              if (cell is SteeringModule) {
                  foreach SteerDataInput {
                       Compute ODC(SteerDataInput);
                  }
              } elsif (cell is Computational) {
                 /* ODCs of computational modules inputs are */
                 /* created by propagation                   */
                 ODC({InputPin_i}) = ODC(Output);
              }
          }
      }
      Get DrivingList(List);
  }
  /* algorithm stops if PIs and FFs outputs are reached */
  if (DrivingList is empty OR DrivingList contains only FFs) {
      /* updating of FFs inputs ODCs */
      foreach FF {
          ODC(FF_Output) &= ODC(FF_Out_i);
          Compute ODC(FF_Input);
      }
  } else {
      BackTrav (StartLevel, False, DrivingList);
  }
}
```

Figure 4: Pseudo-Code of ODC Computation Algorithm.

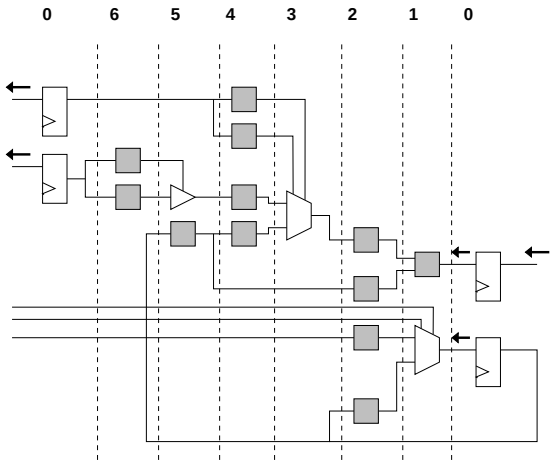

Figure 5: Levelization based on a backward traversal of a generic data-path.

Example 2 *Figure 6 shows an example RTL netlist (disregard the enable signals on the input flip-flops, for the time being). We assume that the ODC for the output is empty. The ODCs for all internal nets B1, B2, ..., B6, are shown in boldface. Consider net B2. It has two fanouts to two computational units. The ODC of the input of a computational unit is the ODC of the output (remember that ODCs introduced by computational units are neglected). Hence, the ODC of B2 is the intersection of the ODCs of the outputs of the computational units:* $ODC_{B2} = (En' + S_1')(En' + S_0') = En' + S_1'S_0' = En'$, *because* S_1 *and* S_0 *cannot be zero at the same time for correct multiplexing.*

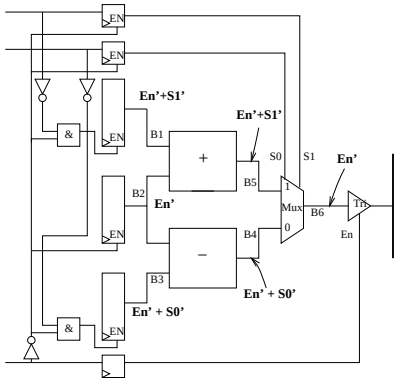

Figure 6: Data-Path DPM Based on External Idleness.

Even though ODC computation is simple for a single steering module, ODC expressions can become quite large if the netlist has many levels of steering modules and many fanout points. To further improve scalability, we implemented two techniques: First, we introduce extra variables to uniquely identify ODC subexpressions when they grow larger than a few tens of terms, and we build new expressions using the extra variables (this is equivalent to describe the ODC with a multi-level netlist). Second, we give the possibility of traversing only a limited number of levels of the netlist. In other words, we can start our backward traversal for ODC construction a specified number of levels L away from the flip-flops. At the starting level, we assume empty ODCs.

4.2 Gating Logic Instantiation

The main difficulty in this step is that ODC conditions masking flip-flops in clock cycle k should be used to gate their clock in cycle $k-1$. In other words, the clock gating logic should be active in the clock immediately before the flip-flop becomes unobservable. Unfortunately, the control signals at the inputs of the ODC functions are generated one clock cycle too late.

If the control signals are available directly as outputs of flip-flops, as shown in Figure 6, the instantiation of the clock gating logic is straight-forward. Logic gates implementing the ODC expressions are instantiated and their inputs are connected to the inputs of the flip-flops. If the control inputs of the steering modules are generated by additional logic, then the entire cone of logic between flip-flops and control signals should be duplicated and connected at the input of the flip-flops, and ODC computation gates should then be connected at the outputs of the duplicated cones. Clearly, this is a non-negligible overhead.

To tackle this problem, in [13] an *observability-based operand isolation* strategy was proposed. In particular, power savings are achieved by insertion of banks of latches and combinational logic gates in order to stop input transitions of a set of selected modules during redundant computation. For each of these modules an activation signal is extracted on the basis of its observability condition. Next, the extracted signals are used to condition the isolation banks in order to block unused computation of the modules themselves.

We notice, however, that also in this case *precomputation* of the ODCs of register outputs represents a difficult problem; generation of the observability conditions is then performed by assuming that the outputs of sequential elements are always observable.

On the contrary, our algorithm is able to perform advanced clock gating extraction by taking into account a larger set of idleness conditions, leading to a lower power solution than for the case of operand isolation. In particular, our strategy is based on the fact that logic duplication overhead can be greatly reduced by observing that the cones of logic leading to the control signals do not need to be duplicated if a *retiming transformation* is applied. If the flip-flops at the inputs of the cones of logic are moved via retiming to the cone's outputs, the circuit remains functionally equivalent to the original one, but the control signals are now directly available at the flip-flops inputs and outputs.

To better appreciate the advantages of our technique with respect to operand isolation, consider the ciruit depicted in Figure7(a). By applying operand isolation, two banks of *latches* are inserted at the inputs of the *adder* and the *subtractor*. Power savings are assured by the fact that isolation banks are controlled by the two activation signals at the input and output of the inverter A (see Figure 7(b)).

However, the area and delay overheads due to the presence of the 16 blocking *latches* is not negligible. After synthesizing the circuit, we have obtained an area overhead of 35% and a delay increase on the critical path of about 14%.

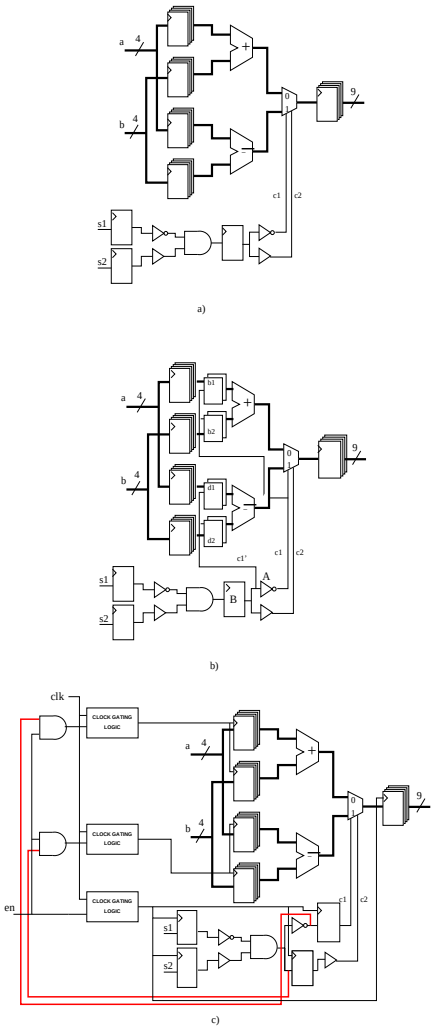

Figure 7: Operand Isolation vs. ODC-Based Clock Gating.

Generally speaking, if the activation signal *c1* arrives at the inputs of the adder isolation bank after the arrival of the data input *b1, b2*, then the delay on the critical path is negatively affected and the power dissipation itself is increased. Conversely, if the activation signal becomes stable before the arrival of *b1, b2*, then the control signal *c1* is available directly as output of flip-flop B. Hence, in this case, no *precomputation* is needed in order to succesfully apply ODC-based clock gating.

By using our ODC-based strategy (see Figure 7(c)), very little additional logic is inserted. Area overhead is just 5.2%, while the delay increases by around 3.8%. Furthermore, our strategy not only saves the same amount of power as operand isolation, but it also reduces the power dissipated by registers and clock tree (i.e., 22% power savings over the implementation using operand isolation).

There are three potential pitfalls in applying this technique. The first is when the cones of logic leading to the control inputs have a large number of fanouts. In this case, retiming can cause a large increment in the number of registers. The second problem can arise when timing constraints are very tight and retiming moves logic on the critical path. The third is due to the fact that if the moved registers are driven by primary inputs, then power dissipation of the original fanin cones is not reduced. In this case, ODC-based clock gating is performed by assuming flip-flops outputs to be always observable, in order to avoid *precomputation*. In our implementation, the increment in the number of flip-flops is controlled, and it is rejected if it exceeds a user-defined threshold (10% in our experiments). If the number of all control flip-flops is in excess after *retiming*, a repositioning of the moved ones is performed until reaching a value just lower than the threshold.

Delay issues are also taken into account. *Retiming* strongly affects the delay of the circuit and estimation of the slack reduction is performed by making use of the timing engine of a synthesis system. In our implementation, we have included a timing check that rejects the retiming move if the critical path is increased by more than a user-specified slack value.

5 Experimental Results

The proposed ODC-based clock-gating algorithms have been implemented within the Synopsys environment, using the *dc-tcl* interface. This choice is motivated by two reasons. First, we can leverage on the HDL parser of DesignCompiler, and operate directly on the structural RTL implementation (known as *gtech* description), in a HDL-independent fashion. Second, we can exploit all Power-Compiler facilities for instantiation of clock-gating circuits, namely: (i) replacement of enabled flip-flops with simpler library cells, if available, (ii) automatic instantiation of clock-gating distribution cells and gated clock rebuffering, (iii) design for testability support, (iv) timing analysis support. Finally, we can exploit Design Compiler and Power Compiler reporting capabilites to compute power, speed and area of our optimized designs.

The main disadvantage of this choice is that the dc-tcl interface can be very slow if it is not tuned with care. In general, a fully compiled, autonomous implementation could be significantly faster than dc-tcl, but at the price of a enormously increased engineering effort to build a network manipulation infrastructure at the same level of Design Compiler. Clearly, the advantages overcome the limitations.

The benchmarks we have used are industrial designs, provided us by STMicroelectronics, Motorola and Infineon Technologies. In all cases, computation time was less than 4 hours on a SUN Ultra Workstation with 512MB of main memory.

Table 1 reports cells number, area, delay and power dissipation for three different implementations. All results were obtained by making use of Design Compiler and Power Compiler. In Table 2 we compare our results to the designs

Bench	*No Clock Gating*				*Traditional Clock Gating*				*ODC-based Clock Gating*			
	Cells	Area	Delay [ns]	Power [mW]	Cells	Area	Delay [ns]	Power [mW]	Cells	Area	Delay [ns]	Power [mW]
Industr1	503	37,915	2.17	8.33	519	36,405	2.50	5.36	551	39,213	2.52	4.47
Industr2	2,811	338,991	25.2	28.72	2,974	330,336	25.62	18.68	3,305	330,054	25.64	16.43
Industr3	309	45,822	4.27	9.0	341	44,991	4.60	5.61	383	46,165	4.66	4.72
Industr4	431	36,215	2.25	8.54	437	33,723	2.58	5.40	460	35,388	2.58	4.78
Industr5	216	15,631	1.72	8.84	226	15,046	1.95	6.37	235	15,137	1.99	4.32
Industr6	15,270	842,628	4.29	886.64	16,031	829,115	4.62	664.98	16,643	841,032	4.65	572
Industr7	18,742	1,291,464	2.30	1123.41	20,098	1,126,391	2.63	876.41	20,261	1,201,912	2.66	722.61
Industr8	95,032	3,959,512	6.86	221.9	96,724	3,623,584	7.19	151.02	99,524	3,826,504	7.25	129.29

Table 1: Results: Traditional Clock Gating vs. ODC-Based Clock Gating

Bench	*Traditional Clock Gating*			*ODC-based Clock Gating*		
	Δ *Area* [%]	Δ *Delay* [%]	Δ *Power* [%]	Δ *Area* [%]	Δ *Delay* [%]	Δ *Power* [%]
Industr1	1.4	4.1	35.7	9.1	3.3	46.3
Industr2	0.4	2.6	34.9	0.3	2.6	43.0
Industr3	5.2	1.8	37.7	7.8	0.7	47.6
Industr4	1.5	6.9	36.8	6.5	2.3	44.0
Industr5	3.5	3.7	27.9	4.2	3.2	51.1
Industr6	3.3	1.6	25.0	4.7	0.2	35.5
Industr7	3.2	12.8	22.0	10.1	7.0	35.7
Industr8	1.8	3.4	31.9	7.5	3.4	41.7

Table 2: Comparision between Traditional Clock Gating and ODC-Based Clock Gating

implemented without clock gating. By performing ODC-based Clock Gating strategy we observe that power savings are averaging around 43% over the initial implementations demonstrating the effectiveness of this technique. Even in those cases where a Traditional Clock Gating strategy is not performable our approach can still assure a large reduction of power dissipation thanks to the presence of the ODC logic alone. Infact, from Table 2 we can also notice that a power reduction of up to 23% can be found over the Traditional Clock Gating while speed reductions are minimal and area overhead is always very tightly controlled.

6 Conclusion

We have presented an automatic clock gating extraction algorithm that operates at the RTL. It is characterized by fast run times and scalability to large designs. It substantially reduces power consumption with respect to existing commercial tools, at a minor cost in terms of area and speed, and it seamlessly integrates with existing industrial design flows.

References

[1] L. Benini, G. De Micheli, *Dynamic Power Management: Design Techniques and CAD Tools*, Kluwer 1997.

[2] M. Pedram, J. Rabaey editors, *Power Aware Design Methodologies*, Kluwer 2002.

[3] M. Favalli, L. Benini, G. De Micheli, "Design for testability of gated-clock FSMs," *European Design and Test Conference* pp. 589–596, March 1996.

[4] D. Garrett, M. Stan, A. Dean, "Challenges in clock gating for a low power ASIC methodology," *IEEE International Symposium on Low-Power Electronics and Design*, pp. 176–181, Aug. 1999.

[5] S. Hassoun, T. Sasao (eds.), *Logic Syntesis and Verification* Kluwer 2001.

[6] L. Benini and G. De Micheli, "Automatic synthesis of low-power gated-clock finite-state machines," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 6, pp. 630–643, June. 1996.

[7] M. Alidina, J. Monteiro, S. Devadas, A. Ghosh, M. Papaefthymiou, "Precomputation-Based Sequential Logic Optimization for Low Power," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 2, No. 4, pp. 426-436, December 1994.

[8] M. Onishi, A. Yamada, H. Noda, T. Kambe, "A Method of Redundant Clocking Detection and Power Reduction at RT Level Design," *ISLPED-97: IEEE International Symposium on Low Power Electronics and Design*, pp. 131-136, Monterey, CA, August 1997.

[9] F. Theeuven and E. Seelen, "Power reduction through clock gating by symbolic manipulation," in *Symposium on Logic and Architecture Design*, pp. 184–191, Dec. 1996.

[10] L. Benini, G. De Micheli, E. Macii, M. Poncino, R. Scarsi, "Symbolic Synthesis of Clock-Gating Logic for Power Optimization of Synchronous Controllers", *ACM Transactions on Design Automation of Electronic Systems*, Vol. 4, No. 4, pp. 351-375, October 1999.

[11] G. Lakshminarayana, A. Raghunathan, K. S. Khouri, N. K. Jha, S. Dey, "Common-Case Computation: A High-Level Technique for Power and Performance Optimization," *DAC-36: ACM/IEEE Design Automation Conference*, pp. 56-61, New Orleans, LA, June 1999.

[12] H. Kapadia, L. Benini, G. De Micheli, "Reducing Switching Activity on Datapath Busses with Control-Signal Gating," *IEEE Journal of Solid-State Circuits*, Vol. 34, No. 3, pp. 404-414, March 1999.

[13] M. Munch, B. Wurth, R. Mehra, J. Sproch, N. Wehn, "Automating RT-Level Operand Isolation to Minimize Power Consumption in Datapaths," *DATE-00: IEEE Design Automation and Test in Europe*, pp. 624-631, March 2000.