Intelligate: Scalable Dynamic Invariant Learning for Power Reduction *

Roni Wiener, Gila Kamhi and Moshe Y. Vardi

Haifa University, Department of Computer Science, Israel.
Intel Corp, Israel.
Rice University, Department of Computer Science, Houston, Texas, USA.

Abstract. In this work we introduce an enhanced methodology to detect dynamic invariants from a power-benchmark simulation trace database. The method is scalable for the application of clock-gating extraction on industrial designs. Our approach focuses upon dynamic simulation data as the main source for detection of opportunities for power reduction. Experimental results demonstrate our ability to learn accurate clock-gating functions from simulation traces and achieve significant power reduction (in the range of 30%-70% of a clock net's power) on industrial micro-processor designs.

1 Introduction

Power consumption has become a major concern for modern microprocessor designs; it affects battery life in the mobile segment, and limits chip frequency in desktops and servers. In this context, a significant design effort is spent on reducing power dissipation, aiming at delivering maximum performance per watt. Power dissipation has a dynamic component, due to the switching of active devices, and a static component, due to the leakage of inactive devices. Since our work targets dynamic power only, further references to "power" in this paper imply the dynamic component.

The clock network is known to be one of the major power consumers, accounting for 30%-40% of the total power of a chip [1]. This can be explained by the large capacitance of the clock net elements, together with their high switching activity. *Clock gating* is one of the most effective and widely used techniques for saving clock power. If a logic block does not perform any useful computation, one can stop the clock of the block, thus saving switching activity and dynamic power [2].

We can classify the existing approaches based on the type of gating conditions. *Unobservability conditions*, or ODCs ("Observable Don't Cares"), were used in [3] to gate state elements that are not observed by their environment. *Stability conditions*, or STCs [4], were proposed in [5] to gate state elements that are stable at the same value. ODCs constitute a natural candidate for clock gating, since they can be computed and expressed as combinational conditions. A scalable ODC-based approach is used in [3]

* Supported in part by NSF grants CCR-0124077, CCR-0311326, CCF-0613889, ANI-0216467, and CCF-0728882, by BSF grant 9800096, and by gift from Intel.

for gating large-scale designs. In contrast, STCs are expressed as sequential conditions (since stability means equality in two successive cycles). For clock gating, one needs to extract combinational STCs, which has proven to be a challenging task [5].

In this paper, we introduce a general framework, called *Intelligate*, with the primary goal of extracting "interesting" dynamic invariants from power-benchmark simulation traces. We demonstrate the usefulness of this framework for STC extraction. We achieve scalability by using a data-mining technique to approximate clock gating from below in a controlled way, enabling the user to navigate the trade-off between acccuracy and computational cost. Our experimental results show that the method scales all the way up to unit-level designs consisting of throusands of sequential elements and finds highly interesing and useful clock-gating conditions, including cross-unit conditions.

1.1 Related Work

Most previous works on automated synthesis of clock-gating conditions are static and extract the conditions from the RTL description of the design. For example, the methods described in [3, 5] identify ODCs or STCs and transform them into clock-gating conditions. Static methods face a trade-off between scalability and accuracy. The FSM-analysis technique of [5] requires extensive logic analysis and does not scale well, which limits its applicability to fairly small design blocks. In contrast, the method of [3] achieves scalability by limiting its analysis to unobservability conditions of a certain kind (those that can be obtained from analysis of steering modules), thereby missing many potentially useful clock-gating opportunities.

Dynamic methods, in contrast to static methods, extract clock-gating conditions from traces describing run-time behavior. Two recent papers describe dynamic approaches to clock-gating synthesis. The method described in [6] analyzes simulation traces for design signals that can be used as clock-gating conditions. This method achieves high scalability, but is limited to very simple clock-gating conditions (sums of literals). A general dynamic framework is proposed in [7], in which clock-gating conditions are to be extracted from simulation traces using machine-learning techniques. The specific learning technique proposed there is precise, which entails that it is not scalable to designs of realistic size.

The idea of extracting conditions from traces was first proposed in the context of software invariant synthesis in [8], and then extended to hardware traces in [9]. To get around the computational cost of machine learning, these works do not attempt to synthesize general conditions, but focus on a restricted class of candidate conditions. The difficulty with attempting to learn general conditions stems from the general difficulty of learning Boolean functions [10]. Data mining is an approach to machine learning that trades accuracy for scalability [11]. The *Apriori* algorithm, which is a basic algorithmic building block for Intelligate, is a scalable algorithm for mining data associations in large databases [12].

2 Overview of Intelligate

Intelligate is a framework that aims at dynamic-power reduction through clock gating. It follows the *Learning-from-Examples* approach [7] to identify stability conditions that

are likely to occur and extracts simple Boolean conditions that can be used for the gating of stable registers. Intelligate learns correlations between signals in the design, while using simulation trace database as the main input source. The output consists of *gating pairs*, where a gating pair is a clock-gating function in the form of a DNF expression and a set of registers that can be gated using this expression.

2.1 Data Preparation

Simulation traces are the source of data for inference of clock-gating functions. We can view a trace as two dimensional matrix, where its columns represent the design's signals and its rows represent the simulation cycles. A cell positioned at the i-th row and the j-th column contains the Boolean value of signal j at cycle i. Since our goal is to extract clock-gating conditions to save dynamic power, it is important to choose execution traces of common design scenarios. Extracting gating conditions from traces of rare execution paths is not effective in power reduction of common design executions. Trace data source for Intelligate comes from real-life power-consuming applications (e.g., Powerpoint). These benchmarks represent typical power-consuming scenarios that bring the design to its *thermal design power* (TDP) [13]. By extracting gating functions for these applications, we ensure the detection of dynamic power reduction opportunities that are relevant for typical execution scenarios.

The basic problem that Intelligate addresses is the extraction of STCs for a given TDP benchmark. These conditions determine if the clock will be gated or not. The value of the registers in each simulation cycle is given in the trace matrix M. We say that register j is stable in cycle i if $M[i,j] = M[i+1,j]$. Thus, we can define the STC matrix S, where $S[i,j] = 1$ when $M[i,j] = M[i+1,j]$, and $S[i,j] = 0$ when $M[i,j] \neq M[i+1,j]$. When we focus on a specific register j, the STC matrix S_j can be viewed as a Boolean vector. Intelligate's goal is to find a combinational function f that appproximates S_j. Specifically, we need to have (1) $S_j[i] = 0 \Rightarrow f[i] = 0$ at all cycles i, and (2) $S_j[i] = 1 \Rightarrow f[i] = 1$ for a large fraction of cycles i. Thus, we can split M into two matrices, M^+ and M^- consisting of *positive* and *negative* examples. The positive-example matrix M^+ consists of all rows i where $S_j[i] = 1$ and the negative-example matrix M^- consists of all rows i where $S_j[i] = 0$. We then look for a gating function f that never gates a register that needs to toggle, and does gate often when the register is stable. Note that registers with constant values in a trace yield an empty M^-. For such registers we cannot learn meaningful gating functions, so they are ignored by Intelligate.

The *support* of a function f is defined as the fraction of cycles it satisfies in a given table. Hence the *positive support* of f is the fraction of rows in M^+ where f is high, and the *negative support* of f is the fraction of rows in M^- where f is low. Thus, the goal of the learning algorithm is to find a gating function f that has *zero* negative support and high positive support. Note that it is easy to extract from M^+ and M^- a function with negative support 0 and positive support 1. Such a function, however, is likely to be a highly complex function and would consume more power than it would save. By asking for high positive support, rather than maximal postive support, we are relaxing the accuracy requirement, which makes it possible for us to look for a low-power gating

function. We cannot relax, however, the requirement of zero negative support without compromising the functional correctness of the design.

2.2 Learning

Rather than try to learn general Boolean functions, we aim at learning function in *disjunctive normal form* (DNF). It suffices, therefore, to focus on learning *minterms* (which are conjunctions of literals), since if c_1 and c_2 are gating functions, then so is $c_1 \vee c_2$. The learning algorithm works iteratively in a bottom-up approach trying to find minterms approximating the STC vector. Its inputs are the complementary positive and negative matrices M^+ and M^- described earlier. The algorithm is based on the *Apriori* data-mining algorithm [12]. It relies on a few observations regarding support:
Observations: Let f, g be Boolean functions.

1. $support(f) \leq support(f \wedge g)$
2. $support(f \vee g) \geq support(f)$
3. If $support(f) = support(g) = 0$, then $support(f \vee g) = 0$.
4. If $support(f) = 1$, then $support(f \wedge g) = support(g)$.

The learning algorithm maintains zero negative support, which guarantees logic correctness. At the same time, the algorithm tries to maximize positive support. Positive support allows measurement of gating-function quality from a dynamic-power perspective. High positive support means that a high percentage of the gating opportunities were utilized, whereas low positive support means that most of the gating opportunities were missed by the function. A threshold for positive support is determined by the user according to the desired power scheme. Minterms with positive support below the specified positive support threshold are redundant, because they are not power beneficial from the user's point of view. The observations above tell us that we can stop "growing" a minterm once the positive support is below threshold. The algorithm is described in detail in Section 3.

2.3 Verification

The output of the learning algorithm consists of a group of gating pairs. A gating pair (m, r) consists of a minterm m and a group of registers r that m can gate. Before a clock-gating functionality can be integrated to the circuit, it is vital to verify the correctness of all gating pairs. The temporal assertion $always(m \Rightarrow (r = next(r)))$, must be verified as correct, as the learning algorithm guarantees that this implication holds only with respect to the input trace.In order to increase our confidance in the assertions, we check that the implication holds with respect to all traces in the trace database, but that does not guarantee correctness. A more extensive verification process is accomplished in two steps. First, dynamic verification (e.g., randomized constrained simulation) is used to filter out the erroneous minterms. Following initial filtering, formal verification is applied on the remaining minterms. Since Intelligate may extract global relations in the design and uses them to halt portions of the clock network, the formal-verification task required to verify the correctness of the gating-conditions would in most cases

surpass the capacity of the the state-of-the-art formal verification tools. In these cases, human experts (e.g., designers of the design at hand) are consulted in order to certify the gating functions, based upon their design knowledge.

2.4 DNF Composition

After the verification stage, minterms and their corresponding groups of registers can be combined to form a DNF expression. Each minterm can be implemented as a clock-gating function by itself, since it has enough positive support and it gates sufficient number of registers to be power efficient. According to the observation above, disjunctions between minterms can increase the overall support, therefore a DNF expression can be composed by building a disjunction between the gating minterms.

3 Intelligate Learning and Grouping Algorithms

The pseudo-code describing the learning process is shown in Algorithm 1.

Algorithm 1 Minterms learning algorithm

Input: A register to gate, M^+, M^-, positive support threshold PST, and Minterms' maximal size k.
Output: A group of gating pairs.
1: $G = \emptyset, C = \emptyset, R = \emptyset$
2: C_1 = All non-constant signals in the design and their negations.
3: $G = \{c \mid c \in C_1 \wedge NegSup(c) = 0 \wedge PosSup(c) \geq PST\}$
4: $C_1 = \{c \mid c \in C_1 \wedge c \notin G \wedge PosSup(c) \geq PST \wedge NegSup(c) < 1\}$
5: **for** $i = 2 \rightarrow k$ **do**
6: C_i = Generate candidates of size i (see Algorithm 2).
7: $G = G \cup \{c \mid c \in C_i \wedge NegSup(c) = 0 \wedge PosSup(c) \geq PST\}$
8: $C_i = \{c \mid c \in C_i \wedge c \notin G \wedge PosSup(c) \geq PST\}$
9: **if** $C_i = \emptyset$ **then**
10: Exit loop
11: **end if**
12: **end for**
13: **for all** $g_i \in G$ **do**
14: R_i = Group registers gated by g_i (see grouping section).
15: **end for**
16: Return all gating pairs (g_i, R_i)

3.1 Generating and Pruning Literals

First, all non-constant signals in the design and their negations are considered as potential clock-gating candidates. As a second step, the pruning of unsuitable signals is accomplished; only signals with zero negative support and positive support above

threshold can be used as clock-gating signals (Step 3). Signals with positive support above threshold and non-zero negative support can participate in larger minterms only if their negative support is less than 1 (Step 4). This filtering process is repeated in the second part of the algorithm for larger minterms. Minterms with zero negative support and positive support above threshold are ponential clock-gating functions (Step 7), and minterms with positive support above threshold can participate in larger minterms (Step 8). (Note that Step 4 and Observation 1 guarantees that all minterms in C_i have negative support below 1.) By Observation 4 in Section 2, minterms with negative support 1 are redundant when they are subsets of larger minterms, so they can be discarded. At the end of the second loop, all minterms in G are potential clock-gating functions. For each such gating function we find the group of registers that be gated by that function.

Note that the learning algorithm is applicable only to non-constant registers and signals. In our experiment, this straightforward filtering played a major role in reducing run time. At the same time, this leaves open the question of finding gating conditions for the large number of constant registers.

3.2 Candidates Generation

Candidate generation for minterms of size two and larger is described in Algorithm 2. The procedure is based on the candidates generated in the previous iteration and the positive and negative matrices M^+ and M^-.

Algorithm 2 K-candidates generation

Input: Candidates of size $i-1$ (C_{i-1}), M^+, M^-.
Output: Candidates of size i (C_i).

{Increase minterm size}
1: $C_i = \emptyset$
2: L = all literals $l \in$ some minterm $c_{i-1} \in C_i$).
3: **for all** minterms $c_{i-1} \in C_{i-1}$ **do**
4: **for all** literal $l \in L$ **do**
5: **if** $l \notin c_{i-1} \wedge \bar{l} \notin c_{i-1}$ **then**
6: $C_i = C_i \cup (c_{i-1} \wedge l)$
7: **end if**
8: **end for**
9: **end for**

{Filter infrequent candidates}
10: **for all** $c_j \in C_i$ **do**
11: **for all** c'_j subset of c_j of size $i-1$ **do**
12: **if** $c'_j \notin C_{i-1}$ **then**
13: Delete c_j from C_i
14: **else if** $NegSup(c_j) \geq NegSup(c'_j)$ **then**
15: Delete c_j from C_i
16: **end if**
17: **end for**
18: **end for**
19: Return C_i

In Steps 3-9, candidate minterms are generated by conjunctions of un-pruned literals to every candidate minterm from the previous iteration. In Step 4, L denotes all the literals from the previous iteration, and every literal is conjoined with each minterm from the previous iteration, thereby increasing its size by one. The second part of the algorithm filters away many of these candidate minterms. We delete a candidate minterm c_j if it has a subminterm c'_j that is not a candidate minterm of size $i-1$ or whose negative support is not smaller than that of c_j.

The rationale for the deletion is as follows. Let c_j be a candidate minterm in Line 10, and let c'_j be a subminterm of c_i. If $c'_j \notin C_i$, then either c'_j has a positive support below threshold, in which case the same holds for c_j, or c'_j has negative support 1, in which case we might as well delete it from c_j and consider the small candidate minterm $c_j - c'_j$, but this smaller minterm must have been considered at an earlier iteration. If $NegSup(c_j) \geq NegSup(c'_j)$, then enlarging c'_j to c_j did not decrease $c'_i s$ negative support but may have decreased its positive support, which means that we have gained nothing by this enlargement.

3.3 Grouping

The grouping stage is carried out after the candidates generation and pruning iterations are finished. The output of this stage is the group G of clock-gating minterms. All the minterms in the G have above-theshold positive support and zero negative support. This was all done with respect to one register. Using the minterms to gate only this one register would probably increase the design's dynamic power consumption, since the implementation of the gating function will consume more power than the power saved by gating the clock. Thus, minterms that gate only small number of registers are not power beneficial. In order to overcome this inefficiency, a group of registers is matched with each gating minterm; for each $g \in G$ we find all registers where the traces satisfy the condition $always\ g \Rightarrow (r = next(r))$.

4 Experimental Results

We conducted our experiments on an industrial design from a high-performance microprocessor design. The design had already been partially clock-gated as an outcome of design reuse from previous generations. The design we studied covers different design styles (control and data path). The source for the power trace data came from sample power applications' (benchmarks) simulations carried out during the design's power evaluation. These power tests are known as TDP (Thermal Design Power) benchmarks. We had seven TDP benchmarks at hand to work with. Each benchmark test represented a simulation trace of a real application execution such as zipping of files or/and reading data through the computer's infrared port. The learning procedure was applied to one benchmark, and then the rest of the benchmark suite was used to verify the logic correctness of the inferred clock-gating conditions through dynamic verification and to test average power reduction over all TDP benchmarks. The extracted conditions were not formally verified; instead a human expert assisted in the ratification of these conditions. As success criterion for the power efficiency of an extracted gating pair, we computed

the percentage of reduction in switching activity (SA) of the clock network elements driving the gated registers group, as dynamic power consumption scales linearly with switching activity.

4.1 Results

We applied Intelligate on a unit-level design, which consists of a set of functional logic blocks. First, Intelligate was applied to a single functional logic block (block A) (see Figure 1) with 6998 signals and 466 registers. The analysis was completed in 1 second. The extracted conditiond yielded clock power reduction in between 26% and 41%. (We report the four largest register groups.) An analysis of the inferred minterms revealed that some of the gating signals are among the block's input signals, meaning that the gating logic is implemented outside block A.

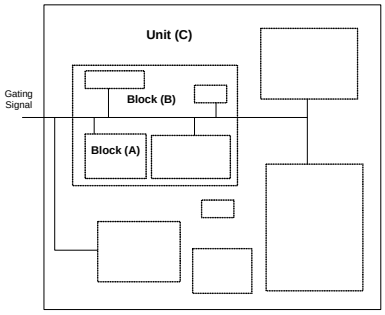

Fig. 1. Cross Block Gating

In order to reveal the cross-block gating, we applied Intelligate to a a group of four functional logic blocks, which we refer to as block B, with 28,000 signals and 3,810 registers. We started with the same register as in the first experiment. Running time now was 26 seconds, and clock power reduction was around 40%. (Here we found three signals gating the same group of 114 registers.) Again, the gating signals found were input signals of this set of blocks as well. This suggests that the gating condition is an architectural signal and therefore, probably, applicable to an even larger group of registers.

We then applied Intelligate to an entire functional unit, containing more than 30 functional logic blocks (unit C). This unit contains 2,664,269 signals and 55,841 registers. The analysis now took 245 seconds, but we found a single signal that can gate 1,603 registers, yield a clock power reduction up to 72% to some of the clock signals in the group (registers that are already partly gated have lower power reduction).

The results, summarized in Table 1, indicate that Intelligate is highly scalable and applicable to designs of large size. The experiments prove Intelligate's ability to detect cross-block and cross-unit clock-gating conditions. Such conditions typically cannot be

detected using static methods, since these require logic analysis, which become infeasible for large designs.

Table 1. Inferred minterms analysis

Minterms size	Gated group size	Power reduction							
		TDP 1	TDP 2	TDP 3	TDP 4	TDP 5	TDP 6	TDP 7	Average
Block A - 6998 signals, 466 registers, threshold: 60%, run time: 1 sec									
1	38	36%	56%	18%	56%	40%	72%	15%	41%
2	20	28%	48%	10%	22%	28%	48%	7%	27%
2	18	25%	35%	11%	30%	27%	48%	7%	26%
2	10	24%	43%	21%	29%	30%	48%	16%	30%
Block B - 28,000 signals, 3810 registers, threshold: 60% run time: 26 sec									
1	114	36%	56%	18%	56%	40%	72%	15%	41%
1	114	32%	36%	18%	56%	38%	72%	14%	38%
1	114	34%	50%	18%	56%	38%	72%	14%	40%
Unit C - 2,664,269 signals, 55841 registers, threshold: 60% run time: 245 sec									
1	1603	36%	56%	18%	56%	40%	72%	15%	41%

The minterms column lists the inferred minterms size, and the number of registers each minterm gates is listed in the second column. Minterms power reduction from the clock nets is listed in the power reduction columns

Intelligate is highly scalable due to three major reasons. First, since its data source is a suite of TDP traces, parts of the design that are not active are ignored. As mentioned earlier, only toggling signals are considered. Our experiments showed that on average only about 5% of the design's signals toggle in the power benchmarks. For example, out of the 6998 signals of block A, only 159 toggle. In block B, only 1601 signals out of 28,000 toggled. In unit C, 178,778 signals out of 2,664,269 signals toggle. Second, the use of the bottom-up Apriori approach contributes to the scalability of Intelligate by smartly increasing minterm size without exploring redundant minterms. Finally, the Apriori approach allows the user to control the threshold, trading off accurary for scalability, which enables Intelligate to analyze very large designs.

5 Conclusions

In this paper we introduce a novel methodology and framework, called Intelligate, with the primary goal of extraction of "interesting" invariants for power optimization. Using a simulation trace data, we applied machine-learning techniques, based on learning from examples, to extract clock-gating conditions. Intelligate, in comparison to previous methods for dynamic invariant extraction, has specialized pruning capabilities that enables it to zero-in to relevant design areas and execution scenarios, while ignoring the rest (e.g., filtering out non-toggling signals). We demonstrated the robustness and scalability of Intelligate for the detection of power-saving opportunities on real-life unit-level (cross-block) micro-processor design, which is orders-of-magnitude larger than what is feasible by previous techniques.

References

1. L. Benini, G. De Micheli. *Dynamic Power Management: Design Techniques and CAD Tools*. Kluwer, 1997.
2. M. Pedram, J. Rabaey (eds.). *Power-Aware Design Mathodologies*. Kluwer, 2002.
3. P. Babighian, L. Benini, G. De Micheli. *A Scalable ODC-Based Algorithm for RTL Insertion of Gated Clocks*. In Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1 (February 16 - 20, 2004). Design, Automation, and Test in Europe. IEEE Computer Society, Washington, DC, 10500.
4. R.Fraer, G.Kamhi, M.Mhameed *A New Paradigm for Synthesis and Propagation of Clock Gating Conditions.* In the proceedings of Design Automation Conference, 2008, Anaheim, USA
5. L. Benini, G. De Micheli, *Automatic Synthesis of Low-Power Gated-Clock Finite-State Machines*. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 15(6), Jun. 1996
6. A. P. Hurst, *Fast Synthesis of Clock Gates from Existing Logic*. Proc. 16th Int'l Workshop on Logic and Synthesis, 2007.
7. P. Babighian, G. Kamhi, M .Y. Vardi. *PowerQuest: Trace Driven Data Mining for Power Optimization.* Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE '07.
8. M.D. Ernst, A. Czeisler, W.G. Griswold, D. Notkin. *Quickly Detecting Relevant Program Invariants.* 22nd International Conference on Software Engineering (ICSE '00) pp. 449.
9. Hangal, S., Chandra, N., Narayanan, S., and Chakravorty, S. *IODINE: A Tool to Automatically Infer Dynamic Invariants for Hardware Designs.* In Proceedings of the 42nd Annual Conference on Design Automation (San Diego, California, USA, June 13 - 17, 2005). DAC '05. ACM Press, New York, NY.
10. M.J. Kearns and U.V. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, 1994.
11. D. Larose. *Data Mining Methods and Models*. Wiley-IEEE Press, February 2006.
12. R. Agrawal and T. Imielinski and A. N. Swami. *Mining Association Rules between Sets of Items in Large Databases.* Proc. 1993 ACM SIGMOD Int'l Conf. on Management of Data, ACM Press, pp.207-216, 1993.
13. http://en.wikipedia.org/wiki/Thermal_Design_Power
14. T. Mudge. *Power: A First-Class Architectural Design Constraint.* IEEE Computing, 34(5), Apr. 2001.
15. V. Tiwari, S. Malik, P. Ashar. *Guarded Evaluation: Pushing Power Management to Logic Synthesis/Design.* IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 17(10), Oct. 1998.
16. L. Benini, G. De Micheli, E. Macii, M. Poncino, R. Scarsi. *Symbolic Synthesis of Clock-Gating Logic for Power Optimization of Synchronous Controllers.* ACM Trans. on Design Automation of Electronic Systems, 4(4), 1999.
17. W. Qing, M. Pedram, W. Xunwei. *Clock-gating and its application to low power design of sequential circuits.* IEEE Trans. on Circuits and Systems I: Fundamental Theory and Applications, 47 (3), Mar. 2000.