Optimizing Leakage Power using Machine Learning

Shuhan Bao
sbao@nvidia.com

December 10, 2010

1 Abstract

As transistor technology nodes continue to scale into deep sub-micron processes, leakage power is becoming an increasingly large portion of the total power. This has been true for many years now, ever since deep sub-micron processes became available. In addition, more recently, computing is becoming increasingly mobile, where minimal power is of paramount importance. As a result, companies are becoming increasingly aware of the need to optimize leakage power to the greatest extent possible. One of the most popular approach to leakage reduction has been to use gates with higher threshold voltages, which reduces the leakage of the gate exponentially, at the cost of increasing the delay through the gate. By swapping gates in paths with significant positive timing slack to their high Vt equivalents, it is possible to greatly reduce the leakage of a design. In this paper, I will discuss an algorithm for solving this optimization problem. By using machine learning to provide a model for the design and standard cell library, we can model the salient variable (rather than an indicator of that variable) and dramatically increase the set of potential swaps we consider, at the cost of adding some error to the timing model.

2 Introduction

Of the methods for reducing the leakage power of a chip, using libraries with multiple Vt classes is amongst the most popular. This method assumes a standard cell library consisting of two or more Vt equivalents for at least most if not all cell types in the library. For every cell type, there would exist a variant of that cell with minimal Vt and equivalently minimal delay. We can use this knowledge to define an optimal achievable timing slack of the design, S_{opt}, where each $S_{opt}(i)$ is the timing slack through cell i, under the condition that all cells in the design are of the minimal Vt (and thus fastest) type. Additionally, we can define a leakage term $L(i, V_t)$, is the leakage of cell i, given that we have assigned its threshold voltage class to $V_t(i)$. The optimization problem is then to choose an array V_t of threshold voltages over i which satisfies

$$\min \sum_{i=1}^{n} L(i; V_t(i))$$

$$\text{s.t. } S(i, V_t) = S_{opt}(i) \vee S(i, V_t) \geq 0$$

In other words, we wish to minimize the leakage power of the design, while maintaining the performance of the design. Alternatively, this constraint can be expressed as requiring that all paths either meet timing $S(i, V_t) > 0$, or consist entirely of the leakiest, but fastest cells.

Current algorithms to do this will heuristically "swap" some subset of V_t based on analysis of the potential power savings and the difference in the cell delay as calculated by the library model. In general, this method requires multiple iterations as crucially, the library only gives a model for the cell delay, but not the quantity $S(i, V_t)$, the path slack. In current deep sub-micron technologies, it is not guaranteed that cell delay changes are perfectly indicative of path delay changes, as changes to the slew propagation will cause downstream cells to speed up or slow down (figure 1). As a result, if optimization proceeds solely on the basis of cell delay, it will be suboptimal for situations in which the path delay differs significantly from the cell delay. Additionally, after each iteration, the full timer must be

updated with the new cell types to determine the new $S(i, V_t)$.

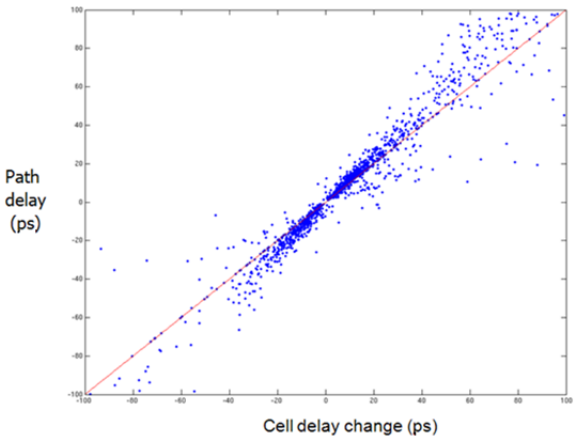

Figure 1: Path delay vs cell delay. x=y line displayed

The purpose of this work is to use machine learning to generate a model for the path slack for each potential swap, which can then be evaluated on new inputs very quickly, in order to allow for optimization algorithms to work on the path delay change rather than the cell delay change. If the error of such a model is minimized, it would be possible to very quickly compute the effect of changing $V_t(i)$ on $S(i, V_t)$ for all cells in the design, and use these values to find an optimal V_t configuration.

3 Methods

In this section, I will describe the methodology for implementing this algorithm. In order to implement the algorithm, as described, it would be necessary to create a model for the slack difference

$$\Delta S = S(i, V_t) - S(i, V_t')$$

observed when swapping cell i. In practice, it makes more sense to create a model for changing the threshold voltage of a standard cell library cell, then applying that model for all cells in the design which are mapped to that library cell.

3.1 Training

In order to gather the training data, an existing EDA tool was used to load a relatively small block of a next-generation GPU chip. Then for each cell in the design, the V_t was swept across all possible values for which an equivalent cell existed in the library, and the subsequent change in path slack was recorded, along with relevant input features. For the most part, in the library that was tested, 3 V_t settings were available for each standard cell type. Based on

Since the goal of the model is to output a continuous valued function, a linear regression minimizing square error on the slack delta was tried. The model parameters are determined by solving the closed formed equation for the parameters, as gradient descent was found to converge very slowly. Despite the potential complexities of analyzing a full path as opposed to a single cell, the linear regression was in fact able to achieve a reasonably good training error. This will be discussed further in the results section.

Features for the linear regression were selected primarily on the basis of first order contributors to transistor and wire delays. Additionally, the amount of downstream logic was included in order to represent the effect of slew propagation along the path.

3.2 Leakage Optimization Algorithm

Once the models for each swap are generated, it is possible to load any design that is bound to the library which was trained on, and apply the leakage optimization algorithm. Two algorithms were tried for the leakage optimization. Both are suboptimal in that both lack complete information that would typically be held in a timer graph. In favor of time constraints, building such a complete view of the design connectivity was decided against. Additionally, both methods use a heuristic method to obtain some connectivity information, which simply assumes that cells with the same slack value are likely to be on the same path. This generally holds if the number of possible slack values is large compared to the number of paths in the design. For the cases where it does not hold, the "merged" paths will diverge after any change along either of the paths. In

general, this assumption is neither always true, nor is it complete, but in practice is a very simple to implement method that improves the performance of both algorithms dramatically over having no connectivity information whatsoever.

In algorithm 1, first, the $S(i, V_t)$, and $x(i)$ is dumped for all cells in the design, where $x(i)$ is the input feature vector for cell i. For each potential single element change in $V_t(i) \rightarrow V_t'(i)$, a reward function is then calculated:

$$R(i, V_t'(i)) = (1 - C) \log (1 + |\Delta S(i, V_t \rightarrow V_t')|)$$

$$-C \log (1 + |\Delta L(i, V_t \rightarrow V_t')|)$$

Where the model learned previously is used to estimate S(i,Vt'(i)). $C \in [0, 1]$ is a parameter to control the relative weighting of the leakage versus the slack minimization as the deltas can be orders of magnitude apart. For simplicity, the reward function above is only valid for swaps to lower threshold. The converse higher threshold swap would simply be $-1 \times R(i, V_t'(i))$. This reward function is chosen on the basis that the best swaps are those with the best ratio of small timing degradation in exchange for large leakage reduction. In this method, C needs to be varied from favoring leakage reduction towards the beginning, to favoring timing fixing towards the end, as the final design is expected to achieve S_{opt} or close to S_{opt}.

In method 2, we evaluate each "path" as described above. First, swap all cells to their lowest V_t equivalent. This gives a value for S_{opt}. In subsequent iterations, for each path, we have a set of cells P, and some slack value $S(i, i \in P) = C$ is constant over all cells along the path. Given this simple "path", we can solve the 0-1 knapsack packing using dynamic programming. First compute a set of "weights" for each swap as defined by the amount the slack would degrade if the swap was executed. We do this using the model learned previously. Second, the "value" of the swap is simply the beneficial change in leakage. The total capacity of the sack is just C. Solving this problem will yield the optimal leakage configuration for any single path. A similar problem can be solved for paths with negative slack, to find the set of changes to V_t that minimize leakage degradation, but set $S(i, i \in P) = S_{opt}(i, i \in P)$ or greater than 0.

In both algorithms, we execute the algorithms across the entire design. After each pass, it is necessary to re-evaluate the timing to verify both that the estimated slack agrees with the actual calculated slack. It is necessary to recompute the timing due to two sources of error. First, there is the potential test error of the learned slack prediction model. Second, there is potential error associated with having an incomplete connectivity picture when doing the optimization. The algorithm may believe two cells to be on independent paths and individually act on both cells, when in fact they are on the same path, and should have been treated together. After re-evaluating the timing, the algorithm will either stop if it has converged, or dump the state data and iterate again.

4 Results

We can view the results in three sections: first, an analysis of the training error, second, an analysis of the test error, and finally, a report of the actual results of running the algorithm on real designs.

4.1 Training Error

The primary contributors to training error seemed to be complex cells and flops. Initially this effect was even more pronounced, until separate models were built for each pathway through the cell (for example, a MUX has 3 pathways from S to Z, I0 to Z, and I1 to Z). In particular, before building path specific models of the delays, the error for flops was enormous, as the effect of swapping is dramatically different for the delay of the Q pin (which acts as a driver) and the setup time of the D pin (which is only serving to capture the signal). One explanation of the remaining error in complex cells and flops can be given in terms of conditional arcs, which cause the delay to differ depending on the state of other inputs to the cell. Since modeling this would require a much more complex model of the design, it was decided to accept this error. From the table below, the overall training error is seen to be approximately 3.5ps across all models built.

Type	Count	Error (ps)
Buffer/Inverter	59	0.9065
Complex	526	3.6155
Sequential	40	5.3318
Total	625	3.4696

Table 1. Training error for a typical library

Based on the observed characteristics of the training error, it was decided that more complex model would not help the accuracy of the model. Figure 2 shows the data for a worst case training error cell type, in this case for a complex XOR gate. In the figure, capaci-

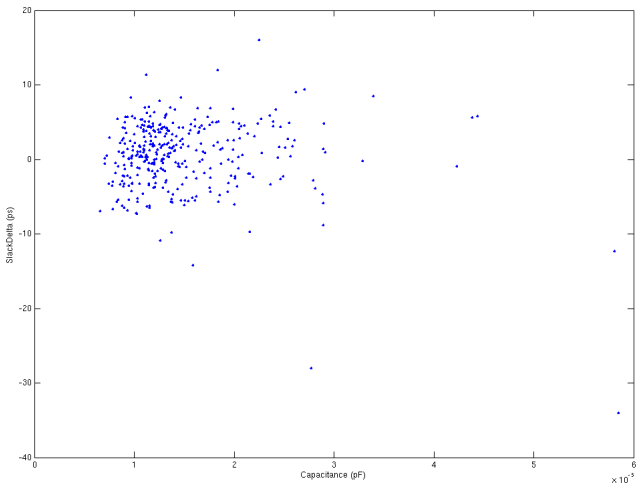

Figure 2: Worst Case Training Error Example

tance is chosen to be plotted against the slack delta, but the same type of graph can be seen for all of the input features. Primarily, work was done to find an input feature that showed reasonable correlation with this output data, but there is no conclusion on this issue yet.

4.2 Test Error

Figure 3 summarizes the test error of the algorithm when executed on a standard 28nm design. The data is gathered by actually running algorithm 2 on a design, and at each iteration comparing the algorithm's expected value of the new slack against the actual observed slack value. Of note is that while training occurred one change per "iteration", the actual execution

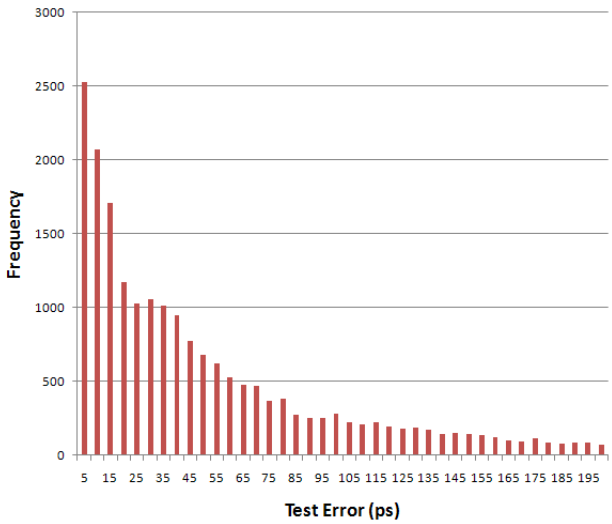

Figure 3: Test error histogram

of the algorithm may swap typically on the order of ten thousands cells at once, without guarantees on their independence. We can see that test error is generally much greater than training error, and that the error is spread over a much wider range. Observations of greater error can be attributed to two factors. First, one of the input features selected for determining the slack differential is the input slew to the cell. For a single swap, this value will be invariant, but in practice when many cells along a path may be swapped at once, the input slew of the downstream cells will change, rendering the model estimate less accurate. The second factor stems from the fact that as described above, the algorithm did not build a complete graph of the cell connectivity. As a result, it may be unaware that is swapping cells along the same path, and independently calculate the expected slack for each path. In this case, the final observed slack will depend on both changes, while the expected values for both only depend on the single change in V_t.

4.3 Leakage Recovery Results

The table below summarizes the actual result of running the algorithm on some select designs. Unfortunately, due to the enormous size of most interesting designs at NVIDIA, it was not possible to gather more data points

Design (names changed)	Cell count	Initial Leakage (mW)	Algorithm 1	Algorithm 2	Current flow
Design 1	113433	7.3	4.9	6.4	5.9
Design 2	427667	45.73	35.3	40.5	38.4
Design 3	464617	36.82	33.1	37.1	33.45
Design 4	68490	48	40.9	41.1	N/A

Figure 4: Leakage results after various optimization methods

within the time frame of the project. We can see from these results that actually method 1 tends to outperform method 2, despite the fact that method 1 is only a heuristic approximation to the packing problem that method 2 solves. This can be understood if we account for the error in the models. Because algorithm 2 tries to optimize the leakage to fit exactly within the slack budget, it's possible that it may add fairly suboptimal swaps (i.e. bad leakage to timing degradation ratio) to fill in what it perceives to be a small shortfall in the slack budget. If the error is sufficiently large, however, this swap may turn out to block a much more useful swap. Unless the timing is violating, algorithm 2 cannot undo previous swaps that it did even based on new information, as this would cause it to take much longer to converge. Algorithm 1 on the other hand is always selecting the best swaps from an infinite budget perspective, and so in the presence of greater errors, it will perform better. The biggest problem observed while running these trials was that as the algorithm proceeded, even though the achieved leakage number was very good compared to the expectation set by the existing leakage optimization flow, the algorithm had a lot of difficulty reconverging on the timing. As noted in the discussion on test error, large errors can be attributable to the algorithm working on seemingly independent, but actually identical data pathways. It often required many iterations and some guidance (reducing the number of swaps at each iteration for example) to make the algorithm converge in the presence of these errors. In many cases where the design was too large, the runtime needed to converge on both a good leakage number and good timing was prohibitive.

5 Conclusions and Future Work

Overall, this was a fairly basic trial to see if a machine learning based algorithm could enable a better leakage optimization algorithm. For certain, there are cases where the algorithm is doing a good job, especially on simple buffer chains, the algorithm performs very well to find an optimum setting for V_t. However, there are still two large sources of error in the method that need to be worked out, primarily, the training error associated with complex cells, and the lack of a complete timer graph. As the algorithm can provide very good leakage numbers when given enough iterations to converge, resolving these two issues with both improve the convergence time and improve the overall achievable leakage. Future work will focus on selecting a more complete set of input features, and generating better models of the design as a whole, rather acting on individual cells in fairly independent fashion.