Bottom-Up Approach for High Speed SRAM Word-line Buffer Insertion Optimization

Bin Wu and Matthew R. Guthaus
Computer Science and Engineering,
University of California Santa Cruz
Santa Cruz, CA 95064
{bwu8,mrg}@ucsc.edu

***Abstract*—The delay of a square SRAM array is dominated by the bit line delay due to the high capacitance per unit length attached to the bit line. Hence, SRAM arrays are usually longer in the word line direction. However, the word line delay also increases dramatically in a simple naive topology and can be a dominating factor when the word line dimension is much longer than that of the bit line. Therefore, word line optimization is an important part of SRAM delay optimization. Buffer insertion, which is commonly used for long interconnects, can also be used to improve word line delay. This paper proposes an approach to place and size the buffers to reduce word line and overall SRAM delay. The proposed methodology improves the read critical path delay by 15.7%, at the cost of only 5.26% extra area in a 128 Kbit SRAM.**

I. Introduction

SRAM array delay increases at different rates when the array size increases in different dimensions. The SRAM array width is the word line direction while the SRAM array height is the bit line direction. When the width is equal to the height, the SRAM delay is dominated by the bit line delay due to the high capacitance per unit length attached to the bit line. Hence, making the SRAM array wider is preferred because the array delay increases at a lower rate in the array width direction than the array height direction. An optimized width/height configuration exists for a given size SRAM and this configuration depends on the SRAM circuit design.

The word line wire delay and the word line buffer delay are critical components for the array delay optimization. The change of array width affects four delay components: the word line buffer, the word line wire, the bit cell, and the column mux delays. However, the effect on the cell delay and the column mux delay are relatively minor. The cell delay increases slightly because longer wire increases the voltage slew and slows down the access transistor switching speed in 6T cells. The column mux delay increases slightly due to the increased ratio of array width and word size with the word size fixed based on the system requirement. The word line buffer delay increases almost linearly due to the increased load on the output and the word line wire delay increases quadratically due to increase in wire length. Hence, this paper will focus on reducing the word line wire and buffer delay to reduce the delay due to the array width.

Buffer insertion [1], [2] is a widely used method for long interconnect delay optimizations. However, it is mostly done for global signals where the interconnect is very long. Due to the high capacitance per unit length for word lines, such a method also works for word lines. Applying buffer insertion within an array allows utilization of wider sized arrays. For example, it allows fewer banks and simplifies bank level control signals.

Intra-array buffer insertion is compatible with the widely used hierarchical divided word line (DWL) [3]–[8]. Figure 1 shows the schematic of a conventional word line buffer and Figure 2 shows that of a DWL [3]. In a conventional topology, the word line wire, which sequentially connects all cells, has a high capacitance attached per unit length and causes a high rate of delay increase as the word line size increases. A DWL breaks a naive single word line into smaller, parallel hierarchical word lines connected by local buffers to a global word line.

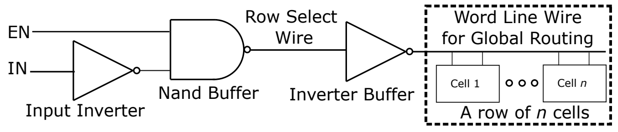

Fig. 1. Conventional memories use a single word line topology which increases the delay quickly as the word line size increases thus is an ideal target for buffer insertion optimization.

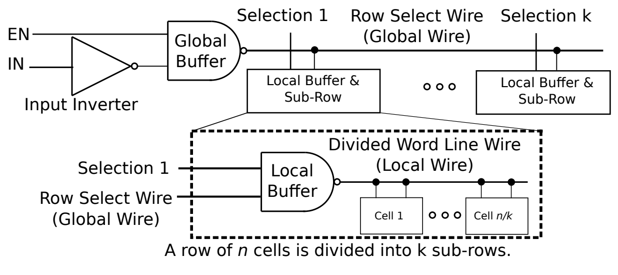
Selection 1
Row Select Wire (Global Wire)
Selection k
EN
IN
Global Buffer
Input Inverter
Local Buffer & Sub-Row
Local Buffer & Sub-Row
Divided Word Line Wire (Local Wire)
Selection 1
Row Select Wire (Global Wire)
Local Buffer
Cell 1
Cell n/k
A row of n cells is divided into k sub-rows.

Fig. 2. DWL is a high-speed variant word line topology and is compatible with buffer insertion.

The DWL with buffer insertion is a hard problem and applying buffer insertion to a naive single word line topology is an important subproblem of it. For a given row size, the best topology of a DWL is not trivial as the segments are not the same size to achieve the smallest delay [9]. Hence, different topologies must be iterated over for a DWL buffer insertion optimization while a standard buffer insertion requires a fixed topology. However, applying buffer insertion to a single word line topology is easier as the topology is fixed and is similar to a subproblem of a DWL buffer insertion optimization.

Hence, this paper focuses on buffer insertion of a single word line topology as the first step. *The novelty of this paper is to apply the Van Ginneken algorithm [10] to buffer insertion on array word lines.* We have built SPICE models for both gates and wires along with an implementation of the Van Ginneken algorithm to size and place buffers in a single word line topology. The delay data is verified with SPICE simulations and the area cost is verified with an open source memory compiler (OpenRAM) [11]. The rest of this paper is organized as follows: Section II presents the background, Section III presents the proposed methodology, Section IV analyzes delay and area impacts, Section V shows the simulation setup, and Section VI shows the simulation results.

II. Background

Figure 3 shows the schematic of an SRAM array. The array is n cells wide and m cells high with a word size of w bits. Synchronization signals are given for word line drivers and sense amplifiers. The read operation is slower than the write operation, thus it is more important and decides the main timing restriction of the SRAM clock.

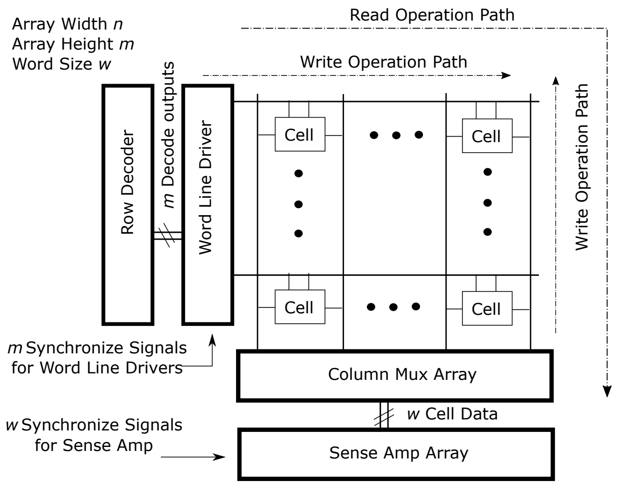

Fig. 3. The width and height of an array only impact the timing restriction of the read and write critical paths.

Table I summarizes the delay increase due to the array width and array height and describes the array delay change assuming only width or height changes while the other one remains the same. For read operations, three important delay components are not impacted by the array width: decoder delay, sense amplifier delay and bit line delay. First, even though the array width affects the row decoder delay, it does not significantly impact the read path delay as row decoder outputs need to be stable before the synchronization signals arrive at word line drivers. Second, the sense amplifier delay is also not impacted due to bit line isolation during read operations. Sense amplifiers are enabled after the sense enable signal arrives and after that they only depends on the sense amplifier design rather than the array size. Third, the bit lines are not dependent on the array width.

TABLE I
Array width and height have different effects on the SRAM delay break down.

Delay	Decoder	Word Line		6T	Bit	Column	Sense
		Buffer	Wire	Cell	Line	Mux	Amp
Width	n	y	y	y	n	y	n
Height	y	n	n	y	y	n	n

The word line buffer delay and the word line wire delay have significant impacts on the read operation. The delay over the bit line wire is not significant as the signal does not change fast enough. The cell delay is a significant component because a 6T cell is a weak driver compared to the high capacitance attached to the bit line per unit length. However, the delay sum of the word line and the word buffer can be comparable to the bit line delay, but this depends on the ratio of the word line and the bit line.

The delay of the naive single buffer topology increases quadratically as the word line size increases. The capacitance attached to the word line per 6T cell consists of two parts: the input capacitance of two access transistors and the capacitance of a wire that is the width of the cell. The 6T cell input gate capacitance is much higher than the wire capacitance in this scenario. Hence, the word line capacitance per 6T cell is much higher than that of a simple wire. Due to the adjacent connection of the cells, the signal needs to go through this slow word line wire, which is as wide as the array, to reach the farthest bit cell.

Inserting buffers breaks the long wire into smaller segments with less total delay. However, the placement and sizing of the inserted buffers are complicated as both gate and wire delays need to be considered for the buffer insertion problem. The gate delays dominate for small size rows, but the wire delay dominates for large size rows causing extremely long row access times. The gate delay depends on three factors: input slew, gate capacitance, and gate type. Wire delays use simple Elmore delay models. The wire capacitance impact on the driving gate is complicated and is a signal slew dependent.

The Van Ginneken algorithm finds the location and size of the inserted buffers. During the first stage, the Van Ginneken algorithm builds solutions from the bottom-up in the routing tree to find non-dominated candidate solutions. During the second stage, the algorithm chooses the best solution at the root and inserts the buffers based on the bottom-up candidate solutions.

III. Proposed Design

This paper proposes a buffered single word line topology as shown in Figure 4. The SRAM row is n cells wide and buffers are inserted on the word line wire to optimize speed. Buffers can be inserted between any two 6T cells in the row and the same insertion pattern is used for all rows in the array.

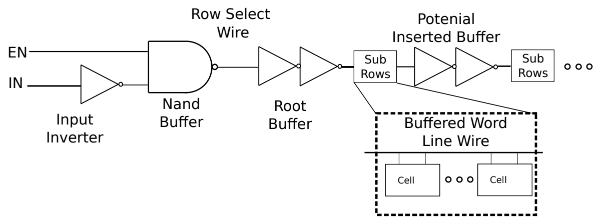

Fig. 4. Inserting buffers in a single word line topology can improve word line delay at the expense of some array area overhead.

A. Word Line Topology and Solution Space

The two access transistors of a 6T cell are represented as a single node in the word line topology. Hence, a row is a tree with n nodes and each node only has one child node. The root node is defined as the closet cell access point to the decode output. The wire between two nodes is represented as a connection and there are $n-1$ connections in the tree.

A solution describes the status of a node under one buffer insertion scenario. A solution contains the input capacitance at the current node, the timing information, buffer size (if any), and the next downstream sub-solution used. The inserted buffer size is the size of the buffer inserted before this node if such a buffer exists and is zero if there is no buffer. The number of available buffer library cells for insertion is k.

B. Inputs and Outputs of the Algorithm

The inputs to the algorithm include a graph representing the interconnect topology with a node for each of the n cells in the row. The nodes are in a vertex set and the connecting wires are in an edge set. Each of the n nodes in the vertex set V is represented by a vertex v_i with i increasing away from the word line NAND buffer. Each node v_i has an internal capacitance that is $2 \times C_{ug}$, where C_{ug} is the gate input capacitance of a minimized gate.

An edge, denoted as e_i, connects vertex v_i to the next downstream vertex v_{i+1} and is in the edge set E. The length of each edge that connects two adjacent 6T cell is defined as a unit wire and has a resistance of unit wire resistance R_{uw} and a capacitance of unit wire capacitance C_{uw}. Inserting a buffer between two cells increases the distance of the connecting edge, but the delay/capacitance impact of the wire length increase is negligible compared to the word line and is thus ignored. All e_i are set to have the resistance of R_{uw} and capacitance of C_{uw}.

The inputs also include a set B of k possible buffer sizes. The input capacitance of a minimized buffer is $(1+R_{pn}) \times C_{ug}$ where R_{pn} is the PMOS/NMOS ratio and the buffer input capacitance is proportional to its size.

As in Figure 4, both the input inverter and NAND buffer are gates before the root node. Their sizes are fixed for a given row size.

The output of the algorithm is an optimal list of sub-solutions s_i at every node v_i. The algorithm starts with each s_i is empty and builds candidate solutions as the algorithm processes.

C. Algorithm Overview

Algorithm 1 shows the pseudocode of the algorithm. There are two major stages: bottom-up (lines 1-17) and top-down stage (lines 18-23). The first, bottom-up stage goes from the leaf node s_n, which is the furthest cell in the row, and traverses in the upstream direction until the root node s_0 to build solution list s_i for each node v_i.n Once the root node is met, the second, top-down stage builds the result by selecting the optimal solution in solution list s_0 at the root node v_0 and recursively selecting each sub-solution used to construct that solution in downstream direction.

Algorithm 1 The Van Ginneken algorithm applied to a single word line topology.

```
Input: Topology ($V$, $E$), Buffer library ($B$)
Output: Optimal delay buffer solution
 1: Create leaf node solutions $s_n$ with and without buffers.
// Create unbuffered solutions in bottom-up manner
 2: for i = $n-1$ to 0 do
 3:     for $s \in s_{i+1}$ do
 4:         Create new solution $s'$ in $s_i$
 5:         Remember sub-solution $s$ in $s'$
 6:         Set delay of $s'$ to delay of $e_i$ plus $s$
 7:         Set capacitance of $s'$ to cap of $e_i$ plus $s$
// Create buffered solutions from unbuffered solutions
 8:     for $s \in s_i$ do
 9:         Create an empty buffered solution list $sbuf$
10:         for $b \in B$ do
11:             Create new solution $s'$ in $sbuf$
12:             Remember sub-solution $s$ in $s'$
13:             Set buffer size of $s'$ to $b$
14:             Set delay of $s'$ to delay of $b$ with capacitance load
                of $s$
15:             Set capacitance of $s'$ to input capacitance of $b$
16:         Add buffered solution list $sbuf$ to $s_i$
// Remove inferior solutions
17:     Prune $s_i$
// Find optimal solution with root word line driver
18: for $s \in s_0$ do
19:     Calculate delay with root word line driver
20:     Remember fastest solution, $s_{min}$
// Construct optimal buffer placement in top-down manner
21: while $s$ is defined do
22:     Add buffer $b$ if set in $s$
23:     Set $s$ to its sub-solution
```

D. Bottom-Up Stage

There are three steps in the bottom-up stage: propagating, buffer insertion, and pruning.

The first step creates the unbuffered solution set by propagating the downstream solution list s_{i+1} to the current solution list s_i (lines 3-7 in Algorithm 1). For a solution s in s_i, the corresponding propagated solution s' is calculated based on s and the connecting edge e_i with a capacitance of C_{uw} and a resistance of R_{uw}. The delay of s' is the sum of e_i propagation delay and s delay, and the capacitance of s' is the downstream capacitance. The size of s_i for a node v_i is the same as the size of s_{i+1} for node v_{i+1} when the propagating step is finished.

The second step inserts buffers for each previous unbuffered solution (lines 8-16 in Algorithm 1). Because there are k buffer sizes, each unbuffered solution will generate k buffered solutions for a total of $k \times |s_{i-1}|$ buffered solutions. There are $(k+1) \times |s_{i-1}|$ total buffered and unbuffered solutions before pruning.

A large buffer library (big k) significantly increases the runtime. On the other hand, the size of library should be sufficient so that there are buffers small and large enough to get an optimal solution. Hence, the provided buffer size range is large enough so that the upper limit does not limit the speed.

The third step prunes the inferior solutions (line 17 in Algorithm 1). An optimal solution, however, is not simply the fastest solution, because the upstream solution is not yet known. A seemingly fast sub-solution in s_i may have a big input capacitance that causes significant wire propagation delay in the upstream solution yielding an overall slow solution in s_{i-1}. On the other hand, a seemingly slow sub-solution in s_i may have a small input capacitance yet not require further buffering upstream and produce a fast solution in s_{i-1}. Therefore, a sub-solution is inferior to another solution only if both its input capacitance and delay are bigger than another sub-solution.

To implement pruning, a solution list is sorted by either delay or capacitance and solutions are compared in order to avoid $O(|s_i|^2)$ comparisons. For example, if the list is sorted by the delay from low to high, a solution in the s_i is only compared to slower solutions after it in sorted order. If one of these compared solutions has bigger capacitance then it is inferior. After pruning, the algorithm removes all inferior solutions and in practice keeps roughly a linear number of pruned solutions.

E. Top-Down Stage

Once all nodes have their pruned solution lists, the top-down stage starts at the solution list s_0 of the root node v_0. The root node is driven by the output of the global NAND buffer with a fixed size. Hence, each solution in s_0 is used to calculate a total row delay and the fastest solution is used (lines 18-20 in Algorithm 1).

Every solution during the bottom-up stage kept track of the buffered sub-solution it used. The top-down algorithm proceeds by adding each buffered sub-solution to the final result. Once there is no next buffered sub-solution, the location and size of all inserted buffers are in the result (lines 21-23 in Algorithm 1).

F. Delay Models and Accuracy

Signal slew causes two sources of mismatch. First, the effective capacitance at the output of a driving buffer is hard to calculate because it depends on the signal slew. This unknown signal slew causes effective capacitance mismatch in line 6. Second, the delay of buffers are pre-measured for fast run-time. A look-up table takes input slew and load capacitance as input keys and uses linear interpolation to find the delay for given inputs. Hence, the unknown signal slew also causes buffer delay mismatch in line 11.

IV. Timing and Area

The proposed design reduces the read path delay at the cost of extra array area. By inserting buffers in the word line, the original long wire is broken into smaller wires with smaller total delay. Even though there are more buffers compared to the single buffer word line topology, the access speed is faster.

The extra area is from the inserted buffers and extra spacing to adjacent bit cells. Having more, smaller buffers costs more area than fewer, larger buffers due to the extra spacing. The buffer and extra space does not significantly change the word line interconnect delay because the buffer widths are relatively narrow compared to the word line as a whole.

The auxiliary circuits of an SRAM are created to match the size of the array including the buffers. The proposed method increases the width of the array thus requires the sense amp array and column mux array to have additional spacing. The proposed work uses the same sense amp array and column mux array as the regular array but with additional spacing.

V. Experimental Setup

We use FreePDK45 design rules [12] and local/intermediate interconnect model parameters from the Predictive Technology Model [13]. A wire that is the width of a 6T cell ($0.7um$) has the following parameters: the unit wire resistance (R_{uw}) is 1.0269Ω and the unit wire capacitance (C_{uw}) is 0.0987fF, assuming $0.075\mu m$ wire width and $0.01\mu m$ wire spacing. The PMOS to NMOS ratio (R_{pn}) is 2 and unit transistor gate capacitance (C_{ug}) is 0.119fF.

A. Baseline and Proposed Design

As mentioned in Section III-B, the size of both the input inverter and NAND buffer are fixed for both the baseline and proposed method. The baseline has one buffer after the NAND buffer and the size of this buffer is the fastest result considering all possible library buffers. Our proposed method inserts buffers based on the results of the Van Ginneken algorithm.

B. Array Size Setup

The comparison considers different array width and height combinations with a constant total size. The array width must be a power-of-two multiple of the word size, which is 64 bits in our simulation, and the maximum array width is 1024 bits as the maximum column mux input is 16. The bit line size is determined from the word line size and the fixed total memory size. For a given memory size, each design uses its fastest combination of array width and array height.

C. *Buffer Setup*

The buffer library uses two equal sized inverters for each buffer and has sizes: 1, 2, 4, 8, 16, 32, 64, and 128 times a minimum-sized inverter. The upper limited is set to be high enough so that the upper limit is not used and therefore does not restrict the potential solutions.

The candidate buffer placement locations are determined by the intervals of the bit cells. Inserting a buffer in the array increases the width by a buffer plus extra spacing which is a technology dependent value (0.56um in our technology).

VI. Simulation Results

Figure 5 shows the bank delay trend. The proposed method starts to insert buffers and reduce the bank delay when the word line size is bigger than 512 bits. However, for a small bank, the optimized array setup is not wide enough for the buffer insertion to be effective. The rest of the section reviews the fastest dimensions for a given sized bank in detail.

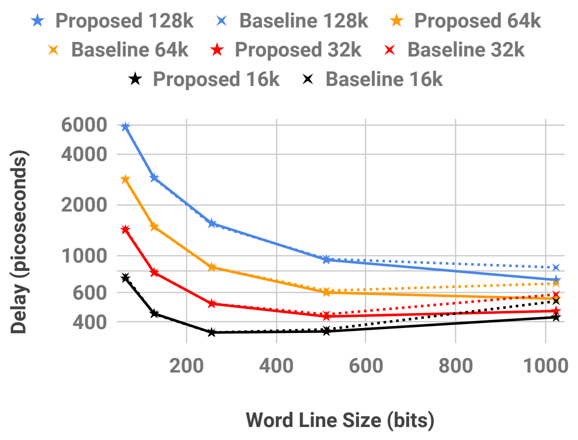

Fig. 5. Buffer insertion starts to reduce bank delay when the word line size is around 512 bits.

Figure 6 shows the topology schematic of algorithm generated buffer placement for a given array width. Buffer insertion is not used when the row size is smaller than 512 bits. For a given array size, the best array width and height can change when the word line topology changes. Table II shows the buffer sizing and placement results of the fastest array width and height for the baseline design and our proposed solution. The proposed solution uses the same setup as the baseline when the array size is smaller than 16 Kbit as the interconnect delay and capacitance are not significant. The proposed solution starts to use extra buffers from an array size of 32 Kbit and inserts more, bigger buffers as the array size increases.

Table III shows the read delay, which is the optimization goal, for the baseline and the proposed solutions. The proposed solution has more read delay improvement as the array size

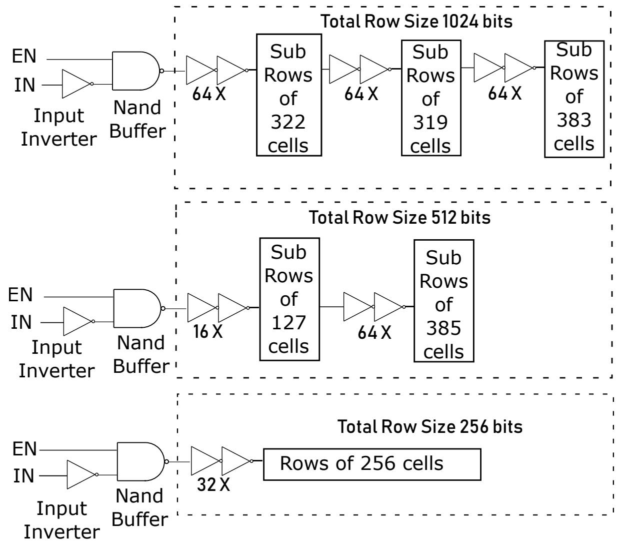

Fig. 6. Optimal buffer insertion solutions show that buffers are needed when the row size exceeds 512 bits.

TABLE II
Optimized Array Results.

	Total Size	Array Width	Array Height	Buffer Num	Location & Size
Proposed	128k	1024	128	3	64x at 0,323,640
Baseline	128k	1024	128	1	64x at 0
Proposed	64k	512	128	3	16x at 0, 64x at 128
Baseline	64k	512	128	1	32x at 0
Proposed	32k	512	64	2	16x at 0, 64x at 128
Baseline	32k	512	64	1	32x at 0
Proposed	16k	256	64	1	32x at 0
Baseline	16k	256	64	1	32x at 0

increases. Such improvement is shown starting at the bank size of 32 Kbit. For a 128 Kbit array, the speed improvement is 15.7%. The intermediate 32 Kbit and 64 Kbit data points show slight fluctuations in delay improvement as the algorithm does not consider the effect of signal slew and the interconnect shielding due to the nature of the bottom-up approach. However, both data points still show slight delay improvement in the final simulation which considers those effects ignored by the algorithm model.

TABLE III
Read Delay Results.

	Total Size	Bank Delay (ps)	Improve Percent
Proposed	128k	713	15.7%
Baseline	128k	846	
Proposed	64k	600	2.43%
Baseline	64k	615	
Proposed	32k	432	2.92%
Baseline	32k	445	
Proposed	16k	347	0%
Baseline	16k	347	

Table IV shows the area of both the baseline solutions and the proposed solutions. The proposed solution area is slightly larger than the baseline design. The fluctuation of the area penalty is due to the same model inconsistencies as the delay improvement.

TABLE IV
BANK AREA RESULTS.

	Total Size	Array Width (um)	Array Height (um)	Area Penalty
Proposed	128k	784.18	194.88	5.26%
Baseline	128k	744.96	194.88	
Proposed	64k	396.96	193.16	4.56%
Baseline	64k	379.63	193.16	
Proposed	32k	395.30	107.08	5.23%
Baseline	32k	375.62	107.08	
Proposed	16k	197.42	102.5525	0%
Baseline	16k	197.42	102.5525	

VII. CONCLUSION

The proposed word line buffering methodology significantly reduces word line delay and bank delay with only minor area cost when the word line size is big enough. This method thus expands the permissible bank size range given performance requirements and can lead to increased flexibility of system-level memory design.

REFERENCES

[1] S. Dhar and M. A. Franklin, “Optimum buffer circuits for driving long uniform lines,” *JSSC*, pp. 32–40, 1991.

[2] V. Adler and E. G. Friedman, “Repeater design to reduce delay and power in resistive interconnect,” *TCAS II*, vol. 45, no. 5, pp. 607–616, May 1998.

[3] M. Yoshimoto, K. Anami, H. Shinohara, T. Yoshihara, H. Takagi, S. Nagao, S. Kayano, and T. Nakano, “A divided word-line structure in the static RAM and its application to a 64K full CMOS RAM,” *JSSC*, vol. SC-18, no. 5, pp. 479–485, 1983.

[4] T. Hirose, “A 20 ns 4-Mb CMOS SRAM with hierarchical word decoding architecture,” *JSSC*, vol. 25, no. 5, pp. 1068–1073, 1990.

[5] T. Chen, D. Lauderback, and G. Sunada, “Optimization of the number of levels of hierarchy in large scale hierarchy memory systems,” *ISCAS*, vol. 5, pp. 2104–2107, 1992.

[6] A. J. Bhavnagarwala, S. Kosonocky, and J. D. Meindlt, “Interconnect-centric array architectures for minimum SRAM access time,” *ICCD*, pp. 400–405, 2001.

[7] B. Rooseleer, S. Cosemans, and W. Dehaene, “A 65 nm, 850 MHz, 256 kbit, 4.3 pJ/access, ultra low leakage power memory using dynamic cell stability and a dual swing data link,” *JSSC*, vol. 47, pp. 1784–1796, 2012.

[8] B. Rooseleer and W. Dehaene, “A 40 nm, 454 MHz 114 fJ/bit area-efficient SRAM memory with integrated charge pump,” *ESSCIRC*, pp. 201–204, 2013.

[9] B. Wu, J. E. Stine, and M. R. Guthaus, “Fast and area-efficient SRAM word-line optimization,” *ISCAS*, 2019.

[10] L. P. P. P. van Ginneken, “Buffer placement in distributed rc-tree networks for minimal elmore delay,” *ISCAS*, p. 865–868, 1990.

[11] M. R. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, and M. Sarwar, “OpenRAM: An open-source memory compiler,” *ICCAD*, pp. 1–6, 2016.

[12] “Freepdk45,” https://www.eda.ncsu.edu/wiki/FreePDK45:Contents.

[13] “Predictive technology model,” http://ptm.asu.edu/.